Objectives

Implement a data type named MegaInt that models arbitrary large integers Implement a MegaInt calculator

Experience operator overloading in a realistic setting

Get continued practice using the the STL sequential container classes, as well as algo-rithms, iterators, C++ strings, and I/O facilities

Practice programming

Your Task

Write an interactive C++ program that simulates a calculator that works with arbitrary large integer values.

Your program should start the calculator as follows:

1 int main() 2 {

3 MegaCalc mc; // Create a calculator, 4 mc.run(); // use it,

5 return EXIT\_SUCCESS; // done, report success 6 } // not so fast, mc’s destructor may throws!

The mc object starts the calculator at line 4; it accepts input from the keyboard, performs the desired operation, and displays the resulting value on the screen. The displayed value is called the called the accumulator. The calculator always uses the accumulator as one of the operands in the next operation. Specically, if the next operation is a binary operation then the accumulator is used as the left operand in that operation, and if the next operation is a unary operation then the accumulator is used as the only operand in that operation.

Every input line must begin with a single character command. If the command is a binary operator, then it must be followed by an unsigned or a signed integer. For unary commands, the input line must begin and end with the command. Tabs and spaces can appear anywhere in a command line, even within the input number. The command (q) ends the calculator run. Finally, an empty command line repeats the previous command.

Here is the output produced by a sample run of the program:

1

1 Welcome to MegaCalculator 2 =========================

3

4 Accumulator: +0

5 Enter input: = + 1 2 3

6

7 Accumulator: 8 Enter input:

9

10 Accumulator: 11 Enter input:

12

13 Accumulator: 14 Enter input:

15

16 Accumulator: 17 Enter input: 18 > +7

19 > +22 20 > +11 21 > +34 22 > +17 23 > +52 24 > +26 25 > +13 26 > +40 27 > +20 28 > +10 29 > +5 30 > +16 31 > +8 32 > +4 33 > +2 34 > +1

35

+123

\*-123456789012345678901234567890

-15185185048518518504851851850470 /-123

+123456789012345678901234567890 =7

+7 h

36 length of the hailstone(+7) sequence: +17 37

38 Accumulator: +7 39 Enter input: -2

40

41 Accumulator: +5 42 Enter input: f 43 +1! = +1

44 +2! = +2 45 +3! = +6 46 +4! = +24

47 +5! = +120

48

49 Accumulator: +5

50 Enter input: \* 111 222 333 444 555 666 777 888 999

51

52 Accumulator: 53 Enter input:

54

55 Accumulator: 56 Enter input:

+556111667222778333889444995 /5

+111222333444555666777888999 q 2

Command list

Input Command Operand Operation Description

+i + i a ++ i + +i a + i + i a

i i a

+i +i a

i i a i i a +i +i a i i a =i = i a =+i = +i a = i = i a %i % i a %+i % +i a % i % i a = i = i a = +i = +i a = i = i a c c a n n a

f f a

a+i add a+i add a+( i) add

a i subtract a i subtract a ( i) subtract ai multiply ai multiply a( i) multiply a=i divide a=i divide a=( i) divide a%i modulus a%i modulus a%( i) modulus i reset a

i reset a ( i) reset a 0 clear a

a negate a

a! factorial of a

h h hailstone( a ) prints hailstone(a), returns its length. Click [here](http://mathworld.wolfram.com/HailstoneNumber.html) for info.

q q quit

‘a’ denotes the accumulator value

‘i’ denotes the input number, which may or may not be signed

input lines like ?+i are eectively the same as ?i, where ? denotes a binary command.

Programming Requirements

a) No dynamic storage allocation; that is, no use of the new or delete operators.

3

b) No C-style raw arrays.

c) Must use the STL deque sequence container to store the digits of the numbers.

Here is why: the STL oers ve sequence container class templates, namely, array, vector, deque, forward list, and list .

The array class template is of no use in this assignment because it allows only xed sized storage, whereas our mega numbers can have arbitrary large number of digits.

You will recall from elementary school pencil and paper methods that the sum, dier-ence, and the product of two integers are formed from right to left but the quotient of the division from left to right. Therefore, ruling out forward list, you look for containers that not only let you scan the digits of a number in both directions but also let you insert and delete digits at both ends of that number eciently. That require-ment rules out vector, leaving you with deque and list to choose from. Like vector, deque provides operator[] overloads but list does not, meaning that deque objects are somewhat easier to use than those of list . Although you know that list allows ecient insertion and deletion in the middle of the container and deque does not, you decide that you have no use for that feature because you only need to push and pop digits at the ends of a number and never in the middle. That will leave you with deque as an ideal container class for storing and manipulating the digits of your numbers.

d) WriteaclassnamedMegaInttomodelarbitrarylargeintegersusingthesign-magnitude notation. Equip MegaInt with the same operators supported by the built-in data types such as int. Your implementation does not have to be highly optimized, but it should not be too inecient either. For example, you might not want to pack a storage unit with multiple digits, but you don’t want to waste storage either. Thus, use a char to store a single digit instead of a long, or an int. To facilitate grading of your work, name your container object as follows:

deque<char> mega\_int; // stores a finite but arbitrary large integer

Obviously, storing digits in chars necessitates conversion between char and int. For example:

int i7 { 7 }; char c0 { ’0’ };

//char c7 = char{ i7 + c0 }; // ’int’ to ’char’ requires a narrowing conversion

char c7 = static\_cast<char>( i7 + c0 ); // reassure compiler that we know we are narrowing mega\_int.push\_back(c7); // store digit 7 as character ’7’

// ...

int digit = mega\_int[0] - int{ c0 }; // convert from char ’7’ back to int 7 // ...

// hide the back and forth conversion mess into ypur operator[] overloads // or other private/protected member functions.

e) Using the dependency relationship shown below implement a class named MegaCalc to provide and manage the user interface:

4

|  |  |  |
| --- | --- | --- |
| MegaCalc |  | MegaInt |
|  |

MegaCalc simply accepts user input from the console, determines the command to be performed, and then delegates the command directly to MegaInt for answers. If MegaCalc oers services such as fatorial(n) and hailstone(n) that MegaInt does not provide, then it implements them itself. For example:

A static member of class MegaCalc

MegaInt MegaCalc::factorial(const MegaInt& n) {

const MegaInt one{"1"}; // similar to int int m{1} MegaInt mega\_fact{ one }; // similar to int mega\_fact {1}

for (MegaInt mega\_k = one; mega\_k <= n; ++mega\_k) // uses op=, op<=, op++ overloads {

mega\_fact \*= mega\_k; // op\*=

cout << mega\_k << "! = " << mega\_fact << endl; // op<< }

return mega\_fact; // uses copy ctor }

Thus, MegaCalc includes a run() member function to run the calculator (as suggested by the driver code on page 1), and two static functions that implement factorial(n) and hailstone(n). Fill free to provide your own helper member functions but include them as private or protected members.

Suggestions

a) To get a feel for the operations you intend to implement, practice them using pencil and paper.

b) Implement addition and subtraction operations rst. Test.

c) To get all four basic arithmetic operations up and running quickly, implement multi-plication in terms addition (see algorithm M1) and division in terms subtraction (see algorithm D1). Test. Once you have implemented and test all the operators, implement M2) and D2).

d) Test your addition and multiplication together by computing 50!, which is:

30414093201713378043612608166064768844377641568960512000000000000

e) As an ultimate test, count the length of the sequence generated by hailstone(n), where n = 50!. For example, the length of the sequence "3, 10, 5, 16, 8, 4, 2, 1" generated by hailstone(3) is 8. Note that hailstone(n) starts at n and ends at 1.

5

Basic Arithmetic with Large Decimal Integers

1 Abstract

This note presents an introduction to basic arithmetic operations on arbitrary large decimal integers. Using the sign{magnitude representation of integers, it describes algorithms for addition and subtraction of integers. It also provides information that can be useful in designing algorithms for multiplication, division, and modulus on large integers.

Although the algorithms presented here use the decimal base b = 10, they remain essentially the same when applied to integers in any number system with b 2. Analysis of eciency of the algorithms presented is straightforward and is left to the reader.

2 Sign{Magnitude Representation of Integers

The most natural and simplest way to represent signed integers is to use the sign{magnitude representation, which resembles the way people habitually write numbers. For example, +10349 represents an integer with + as its sign and 10349 as its magnitude (or absolute value). The magnitude can be expressed in any number system with base b 2. Without loss of generality, we use the familiar decimal number system in base 10, which has ten digits represented by the symbols 0,1,2,3,4,5,6,7,8, and 9.

Formally, the sign{magnitude representation of an n-digit integer Dn is expressed as

Dn = sdDn = sd(d0d1d2 dn 1) n 1 (1)

where sd represents the sign of Dn, and Dn = (d0d1d2 dn 1), a sequence of n decimal digits, represents the magnitude (or absolute value) of Dn. Depending on its sign, Dn may be be expressed as Dn, +Dn, or simply Dn when the sign is +. Thus, Dn = (d0d1d2 dn 1) denotes an unsigned integer, which by denition is a non-negative ( 0) integer. In a context where the number of digits n is implied or not needed, the notations D and D are used for Dn and Dn, respectively. The notation sd denotes the complement of sd.

c

Despite its simplicity, the sign{magnitude representation creates computational problems. It allows two dierent representations -0 and +0 for the number 0, forcing the systems using the representation to handle the two representations identically. That is, software and hardware systems using this representation require additional software code and/or hardware circuitry to support this \double" representation for the number zero.

Another problem with the sign{magnitude representation is that it allows leading zeros in the representation of numbers. For example, 12345 12333 = 00012. The leading zeros are

6

not signicant; they serve only as placeholders to show the scale of a number. For example, the numbers +0123 and +000000123 have dierent scales but the same magnitude 123.

In sign{magnitude representation, therefore, special care must be taken when performing arithmetic and relational operations on numbers with leading zeros. In such operations, the problems caused by the presence of leading zeros can be avoided by simply normalizing the numbers before operating on them. That is, by removing the leading zeros so that the leftmost digit (d0) is non-zero whenever n > 1. Thus, +0 and +123 are in normalized form, but +00000 and +0123 are not.

3 Addition and Subtraction

3.1 Background

Let A = saA and B = sbB be two integers in sign{magnitude representation. The object is to dene addition and subtraction algorithms for computing the integers A+B and A B.

In terms of the magnitudes A of A and B of B, the actual computations of A+B and A B are each carried out in terms of one of the four forms A+B, A B, B A, (A+B). Since A and B are non-negative integers, A + B is an unsigned integer. However, unless A = B, one of the integers A B and B A is unsigned and the other signed. To facilitate our tasks, we set out to avoid the signed case altogether.

For example, if A = +(5) and B = (17), the two unsigned magnitudes are (5) and (17). Thus, we want to compute A + B and A B using only one of the non-negative integers (5) + (17), (17) + (5), and (17) (5), and avoiding (5) (17). We avoid the operation (5) (17) as follows:

A+B = +(5)+ (17) = (17)++(5) = (+(17) +(5)) = ((17) (5)) = (B A)

A B = +(5) (17) = (5)+(17) = +(A+B)

The ideas above can be generalized as shown in the following algorithms:

8

> sa(A+B)

>

>

>

>

>

>

>

>

>

< s (A B) A+B =

a

>

>

>

>

>

>

>

>

:

> sa(B A)

c

> +(0)

if sa = sb

if sa = sb and A > B

if sa = sb and A < B

if sa = sb and A = B

8

> s (A+B) >

>

>

>

>

>

>

>

>

a

< s (A B) A B =

a

>

>

>

>

>

>

>

>

>

> sa(B A)

c

: +(0)

if sa = sb

if sa = sb and A > B

if sa = sb and A < B

if sa = sb and A = B

7

The desired eect is that we always end up adding two unsigned integers and subtracting a smaller unsigned integer from a larger one, each resulting in an unsigned integer.

Therefore, to actually compute A + B or A - B, we are going to need two pairs of algorithms we name add and plus, and subtract and minus. Algorithms plus and minus are private helpers that perform the actual unsigned operations. Algorithms add and subtract provide user interface.

3.2 Addition and Subtraction Algorithms

Algorithm add(A, B)

Input : A = saA and B = sbB Output: C = scC where C = A+B

1 if sa = sb then 2 sc = sa

3 C = plus(A, B)

4 else

Algorithm subtract(A, B) Input : A = saA and B = sbB

Output: C = scC where C = A B 1 if sa = sb then

2 sc = sa

3 C = plus(A, B)

4 else

5 if A > B then 6 sc = sa

7 C = minus(A, B)

8 else if A < B then 9 sc = sc

a

10 C = minus(B, A)

11 else

12 sc = + 13 C =(0)

5 if A > B then 6 sc = sa

7 C = minus(A, B)

8 else if A < B then 9 sc = sc

a

10 C = minus(B, A)

11 else

12 sc = + 13 C =(0)

14 Normalize C 15 return C

14 Normalize C 15 return C

In C++ you can hide the algorithm pairs add and plus, and subtract and minus alto-gether, by providing them only through overloaded operators. Specically, you implement operator+= using add, and operator-= using subtract.

Consider using similar techniques to separate interface and representation for multiplication and division. For example, dene multiply(A, B) and product(A, B) where multiply provided user interface and product computes the product; then hide them both, and implement operator\*= using multiply.

Note that in C++ you can always dene an arithmetic operatorX using operatorX=.

8

3.3 Helper Algorithm plus

Algorithm plus(X, Y)

Input Output

1 carry 0

: X = (x0x1x2 xm 1), Y = (y0y1y2 yn 1), with n 1 and m 1. : Z = (z0z1z2 zp 1), where p = Max(m;n)+1, and Z = X +Y.

2 i p 1 3 j m 1 4 k n 1

5 while (j 0 && k 0) do 6 t xj +yk +carry

7 zi t%10

8 carry t=10 9 i i 1

10 j j 1 11 k k 1

12 // Propagate last carry to unprocessed portion of X 13 while (j 0) do

14 t xj +carry 15 zi t%10

16 carry t=10 17 i i 1

18 j j 1

19 // Propagate last carry to unprocessed portion of Y 20 while (k 0) do

21 t yk +carry 22 zi t%10

23 carry t=10 24 i i 1

25 k k 1

26 // Complete the operation 27 z0 carry

28 return Z

Algorithm 1: Algorithm plus(X, Y)

Example:

carry ! + 0 10 010 1010 11010 111010 111010

left operand ! + rightoperand ! +

627 627 99567 99567

627 627 627 99567 99567 99567

627 627 99567 99567

sum ! 4 94 194 0194 00194 100194

9

3.4 Helper Algorithm minus

Algorithm minus(X, Y)

Input

Output

1 borrow 0

: X = (x0x1x2 xm 1), Y = (y0y1y2 yn 1), n 1, m 1, and X Y.

: Z = (z0z1z2 zp 1), where p = Max(m;n), and Z = X Y

2 i p 1 3 j m 1 4 k n 1

5 while j 0 and k 0 do 6 t xj (yk +borrow) 7 borrow 0

8 if t < 0 then 9 borrow 1

10 t 10+ t

11 zi t

12 i i 1 13 j j 1 14 k k 1

15 // Propagate last borrow to unprocessed portion of X 16 while j 0 do

17 t xj borrow 18 borrow 0

19 if t < 0 then 20 borrow 1 21 t 10+ t

22 zi t

23 i i 1 24 j j 1

25 if borrow = 1 or k 0 then

26 throw \X cannot be less than Y in (X Y)" // Impossible! jXj < jYj in (X Y)

27 else

28 return Z

Example:

X: left operand ! + 32045 32045 32045 32045 32045 32045 Y: right operand ! - 327 327 327 327 327 327 borrow ! - 0 10 010 1010 01010 001010

Z: result ! 8 18 718 1718 31718

10

4 Multiplication

The object is to compute a = b c where the operands b and c are arbitrary long decimal integers, one called multiplicand and the other multiplier.

Here are only three alternative methods for computing a = 345 6789. For eciency purposes, we choose as multiplier the smaller of b and c and denote it by m, and the other as multiplicand and denote it by n. thus, m = 345 and n = 6789.

M1. Simply compute the result of adding n to itself m times; that is, compute a = Pk=1 n. For example, a = 6789 345 = k=1 6789 = 2342205. This is not an ecient method especially when a and m are large and close each other.

m

P

345

M2. This is the traditional paper and pencil algorithm taught at elementary schools, which is by far more ecient than method M1 above. It is also easy to implement.

6789 multiplicand n 345 multiplier m

33945 multiply 5 by n and then left-shift 0 positions

partial sums 0

add 33945 to partial sum ! 33945 271560 multiply 4 by n and then left-shift 1 position

add 271560 to partial sum ! 305505 2036700 multiply 3 by n and then left-shift 2 positions

add 2036700 to partial sum ! 2342205

2342205 ! product

M3.

This methoda uses integer addition and comparison operations only. The method starts by listing in a col-umn headed 2k all successive powers of 2 not greater than m; that is, starting with 20 and stopping at 2k, where k is the largest integer such that 2k m. In a parallel column headed 2kn, it lists n, 2n, 4n,, 2kn. Next, the method marks those entries in the column headed 2k that add up to m. Finally, to compute a, it sums the entries in the column headed 2kn that face the marked entries in the column headed 2k.

aThis method, and the method D3 on the following page, are based on binary arithmetic. They were invented by the ancient Egyptians and in use until around 540. The mathematical jus-tication is straightforward and is left to the curious students.

2k 2kn

1 ! 6789 2 13578 4 27156 8 ! 54312 16 ! 108624 32 217248 64 ! 434496

128 868992 256 ! 1737984

345 2342205 " # m a

11

5 Division

Given two integers n 0 and d > 0, the object is to compute the quotient q and remainder r such that n = dq + r with 0 r < d. Thus, n denotes the dividend and d the divisor. Here are some alternative algorithms.

D1. Repeatedly reduce n by d as long as the successive remainders are greater than or equal zero. The number of reductions made gives q, and the last remainder gives r.

D2. This algorithm is based on the traditional paper and pencil method we learned at elementary school for dividing n by d.1 Specically, it is based on a trivial observation that the traditional paper and pencil method computes the quotient q one digit at a time from left to right, q0, q1, . Each digit qk in q in turn is the quotient of a division in which the dividend n0 has at most one digit more than the digits in the divisor d.

Here is an example, where n = 1594347364730 and d = 7777. Since, there are 4 digits in d, we initially pick the rst 4 digits n0 = 1234 from n to start the division process:

Given n = 1594347364730 and d = 7777 compute q = n=d and r = n%d

k n n0 n0 < d qk 0 1594347364730 1594 0 1 1594347364730 15943 2 2 1594347364730 3894 0 3 1594347364730 38947 5 4 1594347364730 623 0 5 1594347364730 6236 0 6 1594347364730 62364 8 7 1594347364730 1487 0 8 1594347364730 14873 1

9 1594347364730 70960 9

dqk r0 = n0 dqk 0 1594 15554 389

0 3894 38885 62 0 623 0 6236 62216 148 0 1487 7777 7096

69993 967

q in progress

0 02 020

0205 02050 020500 0205008 02050080 020500801

0205008019

Thus, q = 1594347364730=7777 = 205008019 and r = 1594347364730%7777 = 967, both normalized.

Note that the leading digits in n0 shown in black come form r0 in the previous row.

A check-marked ( ) entry in a row k indicates a trivial case where the current dividend is less than the divisor (i.e., n0 < d), implying the trivial solution qk = 0 and r0 = n0.

1See <http://en.wikipedia.org/wiki/Long_division>

12

Now this tricky question: how do we compute the single-digit quotients qk when n0 d? For example, how do we decide that q9 = 9 in the last row or q8 = 1 in the row above it? The answer is we follow the paper and pencil tradition and take \an initial guess"

0 0

qk and then rene qk towards the actual answer qk, which is always a single-digit from 1 through 9 = 10 1.

In base 10, we might decide to run the guesses q0 from 9 through 1 and stop as soon as we nd a q0 for which the remainder r0 is 0 or equivalently n0 dq0 . For example, when n0 = 70960, we nd q9 = q9 = 9 immediately, but when n0 = 15943, we nd q1 = q1 = 2 after 8 tries.

k

k k

0

0

To reduce the number of tries in base 10, we may start with the initial guess in the middle at q0 = 5, getting the initial remainder r0 = n0 d5. Then, if r0 = 0 then we have the answer qk = 5 with r0 = 0. If r0 < 0 we keep decrementing our guess q0 and compute r0 until r0 0; otherwise (i.e., r0 > 0), we keep incrementing qk until we nd a qk for which r d.

k

k

0

0

0

However, for higher bases, we would get help from the leading digits in n0 and d in order to take an \educated" initial guess for qk.2 It is not uncommon for an arbitrary precision calculators or a computer programs to use every bit in an unsigned int of size, say, 4 bytes to represent all 4294967296 \digits" in base 232 = 4294967296.

D3. This method is essentially the reverse of method M3 above. It starts by listing in a column, headed mj = 2jd, the power 2 multiples of d, which consist of the values d, 2d, 4d, 8d, , mk = 2kd such that k is the largest integer for which mk n. It also lists in a parallel column headed 2j the binary powers 20, 21, , 2k.

To compute r, the method subtracts from n all multiples of d in k steps in the bottom-up direction, producing a list of non-negative remainders rk, rk 1, ,r1, r0, starting with rk = n mk. For j = k 1;k 2; ;1;0, if rj+1 mj then mj is marked and rj = rj+1 mj ; otherwise, rj = rj+1. Finally, r = r0.

To compute q, it sums the binary powers 2j that correspond to the marked mj entries.

The gure shown below illustrates the entire process in the context of an example where n = 10;000;000 and d = 1234. After the column headed mj = 2jd has been completed, we nd k = 12. To further clarify the process, we construct a column headed rj listing the successive remainder values, starting at the bottom with r12 = n m12 and going upward and ending at the top with r0.

2See Donald Knuth, Vol. 2, Sec. 4.3. page 250.

13

j 2j mj = 2jd rj

0 1 1234 r1 m0 1 2 2468 r2 m1 2 4 4936 r3 m2 3 8 9872 r4

4 16 19744 r5

5 32 39488 r6 m5 6 64 78976 r7

7 128 157952 r8 m7 8 256 315904 r9 m8 9 512 631808 r10 m9

10 1024 1263616 r11 m10 11 2048 2527232 r12 m11 12 4096 5054464 n m12

898 ! r 2132

4600 9536 9536 9536 49024 49024 206976 522880 1154688 2418304

4945536 "

8103 ! q

14

Sample program run 2: MegaClac computing 50!

Accumulator: +50 Enter input: f +1! = +1

+2! = +2 +3! = +6 +4! = +24 +5! = +120 +6! = +720 +7! = +5040

+8! = +40320 +9! = +362880 +10! = +3628800

+11! = +39916800 +12! = +479001600 +13! = +6227020800 +14! = +87178291200

+15! = +1307674368000 +16! = +20922789888000 +17! = +355687428096000 +18! = +6402373705728000

+19! = +121645100408832000 +20! = +2432902008176640000 +21! = +51090942171709440000 +22! = +1124000727777607680000

+23! = +25852016738884976640000 +24! = +620448401733239439360000 +25! = +15511210043330985984000000 +26! = +403291461126605635584000000

+27! = +10888869450418352160768000000 +28! = +304888344611713860501504000000 +29! = +8841761993739701954543616000000 +30! = +265252859812191058636308480000000

+31! = +8222838654177922817725562880000000 +32! = +263130836933693530167218012160000000 +33! = +8683317618811886495518194401280000000

+34! = +295232799039604140847618609643520000000 +35! = +10333147966386144929666651337523200000000 +36! = +371993326789901217467999448150835200000000

+37! = +13763753091226345046315979581580902400000000 +38! = +523022617466601111760007224100074291200000000 +39! = +20397882081197443358640281739902897356800000000 +40! = +815915283247897734345611269596115894272000000000

+41! = +33452526613163807108170062053440751665152000000000 +42! = +1405006117752879898543142606244511569936384000000000 +43! = +60415263063373835637355132068513997507264512000000000

+44! = +2658271574788448768043625811014615890319638528000000000 +45! = +119622220865480194561963161495657715064383733760000000000 +46! = +5502622159812088949850305428800254892961651752960000000000

+47! = +258623241511168180642964355153611979969197632389120000000000 +48! = +12413915592536072670862289047373375038521486354677760000000000 +49! = +608281864034267560872252163321295376887552831379210240000000000

+50! = +30414093201713378043612608166064768844377641568960512000000000000

Accumulator: +50 Enter input: