

Assignment: Bigint

How much, *exactly*, is 987654321 times 123456789?

Your calculator can't tell you (or at least, mine can't). Neither can the standard C `int` operators; the maximum integer that most C compilers can handle is $2^{31} - 1$, or a little over 2 billion. And floating point types can't do the job either, because they aren't precise enough to give the exact result.

Of course, you could work the answer out yourself by hand if you had enough time and paper!

Task

Consider the following code, which implements a very simple calculator:

```
int n1, n2;
char op;
while (cin >> n1 >> n2 >> op) {
    switch (op) {
        case '+':
            cout << n1 + n2 << endl; break;
        case '-':
            cout << n1 - n2 << endl; break;
        case '*':
            cout << n1 * n2 << endl; break;
        case '/':
            cout << n1 / n2 << endl; break;
    }
}
```

The code will read input consisting of Reverse Polish Notation (RPN) expressions like these

```
2 2 +
100 100 -
10 3 /
6 7 *
```

and for each expression it will display the result

```
4
0
3
42
```

This code will give you the answer to expressions with ordinary-sized integers, but it can't handle numbers as big as the ones above. If you try, you'll just get numeric overflow errors.

Your task is to define a C++ class, **Bigint**, that can compute results for truly l-o-n-g integers. Your class should be contained in 2 source code files: **Bigint.h** (the interface) and **Bigint.cpp** (the implementation). To use your class in the calculator, simply include the class header and change the `int` variables in the code to **Bigint**. You should find that the answer to the problem above is 121932631112635269.

Background

For your class to work properly, you'll need to define appropriate constructors, assignment operators, input and output operators, and of course arithmetic operators. (If you wanted to use it as a general-purpose numeric type, you would also need to add features that are not needed here, such as comparison operators and type conversions to and from other numeric types.)

The arithmetic operators and input and output operators should be defined as friend functions:

```
friend Bigint operator+ (const Bigint& n1, const Bigint& n2);
... // other arithmetic operators

friend std::ostream& operator<< (std::ostream& out, const Bigint& n);
friend std::istream& operator>> (std::istream& in, Bigint& n);
```

Because the numbers involved are too large for a simple `int` (or even a `long`) variable, the class will need to contain a data structure to store information about the number it represents, and methods to manipulate the data. The simplest approach is to use an array, with each array element storing a separate digit. Then you could implement addition, for example, by adding corresponding digits in the same way as you would do it on paper. Of course, you would need to deal with carries as well. You'll find it easier to do the arithmetic if you right-align the number in the digit array and fill unused digits with zeros. For the purposes of this assignment, you can assume that no number will contain more than 256 digits.

The input operator will need to extract characters from the input stream and use them to initialise the elements of the digit array. One way is to read a string from the stream, then convert each character in the string into the corresponding digit value and store it in the appropriate element of the array. Since the digit characters are consecutive in the ASCII table, you can convert the ASCII code for a digit character into the corresponding numerical value by subtracting the ASCII code for digit 0. The output operator should skip over leading zeros and convert the numeric values back to characters by adding ASCII 0.

Multiplication and division are harder than addition and subtraction (but you already know that!). One way to implement multiplication is by repeated addition. For example, you could multiply a number by 23 by adding it up 23 times. And if you already have code to do additions, you might as well use it to help with the multiplication. The same idea applies to division, which you could implement as repeated subtraction.

For smallish numbers (say, up to about a million!) you could use this simple approach. But for larger numbers, it's not quite that easy. For example if you tried to multiply two 20-digit numbers by repeated addition on a computer that could perform one million additions per second, it would take around one million years to finish the job! Instead, you'll need to deal with the problem one digit at a time, much as you would on paper. For example, you can think of multiplication by 247 as multiplication by 7 plus multiplication by 40 plus multiplication by 200. Multiplying by 7 is easy: just add 7 times. Multiplication by 40 or 200 is also easy: add 4 (or 2) times and then add one (or two) zeros at the end. This way, you only need to do a small number of additions. For the 20-digit number, for example, the most additions you would ever need is 200 (20 digits times at most 9 adds per digit plus 20 more additions to total the partial results). You'll get the answer in a fraction of a second.

Core Function Points

In writing your code, you can make the following assumptions:

- No input line will be longer than 255 characters.
- Numbers are represented by strings of decimal digits of up to 255 characters.
- Operators are represented by the usual symbols: +, -, *, /
- Input expressions consist of numbers and operators with one space between symbols.
- All input numbers and results are non-negative.
- Only input that conforms to the given requirements will be used to test your code at a particular level
- Test expressions are “well behaved”: all are correctly formed, with no “syntax errors”.

Write the main function for the calculator and define the `BigInt` class header and enough of the implementation so that the program will compile without errors. Although all of the operators will need to be declared in the class header, their implementation need not be correct. For now, just return the first parameter; it’s clearly not the correct answer, but it will keep the compiler happy!

Define the input extraction and output insertion operators. Input will consist of a sequence of decimal digit characters terminated by whitespace. Output should suppress leading zeros, except that the value zero should appear as a single 0 character.

1. The program evaluates additions of “short” integers, which can be represented as `ints`.

input	output
42 1 + 123 321 +	43 444

2. The program evaluates additions of a long integer and a short integer.

input	output
1000000000000 1 + 123456789123456789 1234 +	1000000000001 123456789123458023

3. Implement the + operator so that the calculator works correctly for addition of long integers.

input	output
1000000000000 1 + 123456789123456789 1234 +	1000000000001 123456789123458023

4. Implement the - operator so that the calculator works correctly for subtraction of long integers. You can assume that no expression will give a negative result.

input	output
1000000000000 1 - 3000000000000 2000000000000 -	999999999999 1000000000000

5. Implement the `*` operator so that the calculator works correctly for multiplication of long integers. For this level, you can assume that the second operand is small enough that the computation can be performed by repeated addition.

input	output
1000000000000 2 *	2000000000000
123456789123456789 123 *	15185185062185185047

6. Implement the `/` operator so that the calculator works correctly for division of long integers. For this level, you can assume that the result is small enough that the computation can be performed by repeated subtraction.

input	output
10000000000 50000000 /	200
987654321 123456789 /	8

Extension Function Points

7. Extend the `*` operator so that the calculator works correctly for multiplication of arbitrary long integers. You can assume that no result will be longer than 256 digits.

input	output
3000000000000 2000000000000 *	6000000000000000000000000000000
987654321 123456789 *	121932631112635269

8. Extend the `/` operator so that the calculator works correctly for division of arbitrary long integers. You can assume that division by zero will never occur.

input	output
6000000000000000000000 20000 /	300000000000000000000
121932631112635269 123456789 /	987654321


9. Modify the calculator code (you shouldn't need to change the `Bigint` code) so that the program evaluates multi-operand RPN expressions of depth 1. You can assume that the expressions do not contain format errors.

input	output
6 5 + 4 - 3 * 2 /	10
123456789 987654321 * 123456789 /	987654321

10. Document your reasoning on the data structures used and method you chose to solve this problem.

Also included here is the understanding that you will:

- Layout.** Use indentation and white space to clearly display organisation and function. Apply layout rules universally and consistently.
- Names.** Choose names that clearly indicate the purpose and function of the entity they name, and that are of suitable length for their roles. Where appropriate, follow naming conventions.

- 
- iii. **Structure.** Match control structures to the tasks they perform. Declare variables only when necessary, and use appropriate scopes and types.
 - iv. **Documentation.** Document all external interfaces according to accepted conventions. Use internal documentation where necessary and where it adds value to the code.

Assessment

- Your solutions to the first nine core function points will be marked automatically.
- The last function point; Documentation, will be handed in as a ‘pdf’ document. It should be at least 300 words but no more than 500. Penalties will apply if these counts are exceeded by 10%.