COMP 2012 Object-Oriented Programming and Data Structures

Assignment 1 Decimal Number

Page maintained by

Wallace Mak
Email:
wallacem@cse.ust.hk
Last Modified:
10/11/2021 07:49:12

Homepage

Course Homepage

Introduction

In this assignment, you will implement the **Decimal** class which can store a decimal number of infinite precision. Simple operations such as addition and multiplication will be supported. Through the implementation of it, you will have practices on various C++/OOP fundamentals, e.g., classes, objects, constructors, copy constructors, pointers, dynamic arrays, etc.

Decimal Number

Well, it is just the type of numbers we use every day. 1, 2, 3.1415, 9000000 are all decimal numbers. Not much to talk about really, but if you really want to read about its formal definition, feel free.

For this assignment, we will use the format commonly used in Hong Kong and many English-speaking countries, with no commas in it, so again like: 3.1415161718

We also do not add meaningless zeros at the beginning or at the end, so it has to be 3.1415161718 but not 03.1415161718 or 3.14151617180 as those zeros do not really change the numerical value of the original number. We will only consider 3.1415161718 as the only valid representation. Similarly, 0.0 and 7.0 are invalid, as they should just be single-digits: 0 and 7. Pay attention to that when you store the number in the class.

For terminology, we will refer the dot as the "decimal point".

That's it! While the description is very easy to understand, certain tasks such as subtraction and multiplication can be challenging programmatically. Therefore you are highly recommended to start early.

Read the FAQ page for some common clarifications. You should check that a day before the deadline to make sure you don't miss any clarification, even if you have already submitted your work then.

If you need further clarifications of the requirements, please feel free to post on the Piazza (via Canvas) with the pa1 tag. However, to avoid cluttering the forum with repeated/trivial questions, please do **read all the given code, webpage description, sample output, and latest FAQ (refresh this page regularly) carefully before posting your questions**. Also, please be reminded that we won't debug for any student's assignment for fairness.

Submission details are in the Submission and Deadline section.

We value academic integrity very highly. Please read the Honor Code section on our course webpage to make sure you understand what is considered as plagiarism and what the penalties are. The following are some of the highlights:

- Do NOT try your "luck" - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for **BOTH** the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there may be additional disciplinary actions from the department and university, upto and including expulsion.

Download

- Skeleton code: skeleton.zip

Please note that **you should only modify and submit the decimal.cpp** file. While you may modify main.cpp to add your own test cases, you should make sure your submitted decimal.cpp can compile with the original main.cpp and decimal.h on ZINC.

If you use VS Code, you may follow the creating a project and using the terminal for custom compilation command section on our VS Code usage tutorial. That is, create a folder to hold all the extracted files in your file explorer, then open this folder in VS Code. You can then use the terminal command `g++ -std=c++11 -o programName *.cpp` to compile all sources in the folder to the program. You are also welcome to create a Makefile for it yourself. After the compilation, you can then use the command `./programName` OR `.\programName` (depends on the OS/shell you use) to run the program.

The Decimal Class

A Decimal object represents a decimal number. It has infinite precision meaning it can store an infinite number of digits as long as your computer has enough memory for it, since the storage of the digits is backed by a dynamic array. It supports various simple operations such as addition and multiplication which are usable via member functions that will be implemented by you. You should also read the main.cpp and sample output for test cases of the member functions to help yourself understand what exactly each of them does.

You may refer to the decimal.h while you read this section.

Data member

```
char* content
```

This private data member points to a dynamic char array which stores the entire decimal number, including the decimal point (the dot) in the middle if the number has a fractional part, i.e., some digits after the decimal point, and a negative sign (the minus sign "-") at the beginning if the number is negative.

Note that it is NOT a C-string array which always has a null character '\0' at the end. It is always just big enough to store all the digits, the dot (if any), and the negative sign (if any). For example, to store the number -3.14 the dynamic array will be of size 5, and to store the number 0 the dynamic array will be of size 1.

```
int size
```

This private data member stores the exact size of the `content` array.

Member functions

```
Decimal()
```

This is the default constructor. You should allocate a dynamic char array of size 1 and have `content` point to it. The char array should store just the character '0' to represent the decimal number 0. Hint: don't forget the member variable `size`.

```
Decimal(int input)
```

This is the conversion constructor that converts the given integer to the decimal number to be stored in the Decimal object. For example, if the `input` is -123, then your Decimal object should be initialized such that it has a `content` array of size 4 that stores "-123" in it. (without the \0 null character at the end, as mentioned earlier) *(fixed a typo in the example at 4:43pm, Oct 1st)*

```
Decimal(const char* input)
```

This is the conversion constructor that converts the given character array (C-string) to the decimal number to be stored in the Decimal object. For example, if the `input` is "-123.4" (has \0 null character at the end), then your Decimal object should be initialized such that it has a `content` array of size 6 that stores "-123.4" in it. (without the \0 null character at the end, as mentioned earlier) You can assume the `input` always contains a valid decimal number.

```
Decimal(const Decimal& another)
```

This is the deep copy constructor.

```
~Decimal()
```

This is the destructor. Remember to deallocate the `content` array.

```
void print() const
```

It prints the number with double-quotation marks added before and after it. It is given to you in decimal.h. Please read its code to help yourself understand how the number is stored in `content`.

```
bool isNegative() const
```

Return true if the number is negative. Return false otherwise.

Examples:

- Number is "99". The function returns false.
- Number is "0". The function returns false.
- Number is "-123.4". The function returns true.

```
Decimal flipSign() const
```

Return a Decimal number which has the same value but the opposite sign of this number.

Examples:

- This number is "123.4", the function should return a decimal number that stores "-123.4".
- This number is "-123.4", the function should return a decimal number that stores "123.4".
- Special case: this number is "0", the function should just return "0". Note that "-0" is considered invalid in this assignment. We only accept "0".

(fixed the description at 5:09pm, Oct 1st)

```
bool isLargerThan(const Decimal& another) const
```

Return true if this number is larger than `another` number. Return false otherwise.

Examples:

- This number is "123.4" and `another` number is "456.78", the function should return false.
- This number is "1245.4" and `another` number is "456.78", the function should return true.
- This number is "1245.4" and `another` number is "1245.4", the function should return false.

```
Decimal add(const Decimal& another) const
```

Add this number with `another` and return the result.

Examples:

- This number is "123.4" and `another` number is "456.78", the function should return a Decimal object that stores "580.18".
- This number is "0.5" and `another` number is "1.5", the function should return a Decimal object that stores "2".
- This number is "123.4" and `another` number is "-456.78", the function should return a Decimal object that stores "-333.38".
- This number is "456.78" and `another` number is "-456.78", the function should return a Decimal object that stores "0".
- This number is "123412312312312.123" and `another` number is "87546687450968405968", the function should return a Decimal object that stores "8754681086328071828 0.123".

```
void addToSelf(const Decimal& another)
```

Add this number with `another` and store the result in this number.

Examples:

- This number is "123.4" and `another` number is "456.78", this number becomes "580.18" after the function call.
- This number is "1234.5" and `another` number is "-1.5", this number becomes "1233" after the function call.

```
Decimal multiplyByPowerOfTen(int power) const
```

Multiply this number with ten to the power of the given `power` and return the result. You can assume the given `power` is always non-negative.

Examples:

- This number is "123.4" and `power` is 0, the function should return a Decimal object that stores "123.4".
- This number is "123.4" and `power` is 1, the function should return a Decimal object that stores "1234".
- This number is "123.4" and `power` is 2, the function should return a Decimal object that stores "12340".
- This number is "123.4" and `power` is 3, the function should return a Decimal object that stores "123400".

```
Decimal multiplyBySingleDigit(int multiplier) const
```

Multiply this number with a single-digit number given by `multiplier` and return the result. You can assume the given `multiplier` is always in [0, 9].

Examples:

- This number is "123.4" and `power` is 0, the function should return a Decimal object that stores "0".
- This number is "123.4" and `power` is 1, the function should return a Decimal object that stores "123.4".
- This number is "123.4" and `power` is 9, the function should return a Decimal object that stores "1110.6".

```
Decimal multiply(const Decimal& another) const
```

Multiply this number with `another` and return the result.

Examples:

- This number is "123.4" and `another` number is "456.78", the function should return a Decimal object that stores "56366.652".
- This number is "-123.4" and `another` number is "456.78", the function should return a Decimal object that stores "-56366.652".

```
Decimal subtract(const Decimal& another) const
```

Subtract `another` number from this number and return the result.

Examples:

- This number is "123.4" and `another` number is "456.78", the function should return a Decimal object that stores "-333.38".
- This number is "456.78" and `another` number is "456.78", the function should return a Decimal object that stores "0".

```
int countDigitsBeforeDP() const
```

Return the number of digits before the decimal point.

Examples:

- This number is "123.4", the function should return 3.
- This number is "0.4", the function should return 1.
- This number is "99", the function should return 2.

```
int countDigitsAfterDP() const
```

Return the number of digits after the decimal point.

Examples:

- This number is "123.4", the function should return 1.
- This number is "0.4123", the function should return 4.
- This number is "0", the function should return 0.

Sample Output and Grading Scheme

Your finished program should produce the same output as our sample output (*updated at 2:41am on Oct 2nd*) for all given test cases. User input, if any, is omitted in the files. Please note that sample output, naturally, does not show all possible cases. It is part of the assessment for you to design your own test cases to test your program. Be reminded to remove any debugging message that you might have added before submitting your code.

There are 38 given test cases of which the code can be found in the given main function. These 38 test cases are first run without any memory leak checking (they are numbered #1 - #38 on ZINC). Then, the same 38 test cases will be run again, in the same order, with memory leak checking (those will be numbered #39 - #76 on ZINC). For example, test case #40 on ZINC is actually the given test case 2 (in the given main function) run with memory leak checking.

Each of the test cases run without memory leak checking (i.e., #1 - #38 on ZINC) is worth 1 mark. The second run of each test case with memory leak checking (i.e., #39 - #76 on ZINC) is worth 0.25 mark. The maximum score you can get on ZINC, before the deadline, will therefore be 38*(1+0.25) = 47.5.

About memory leak and other potential errors
Memory leak checking is done via the `-fsanitize=address,leak,undefined` option (related documentation here) of the latest g++ compiler on Linux (it won't work on Windows for the versions we have tested). Check the "Errors" tab (next to "Your Output" tab in the test case details popup) for errors such as memory leak. Other errors/bugs such as out-of-bounds, use-after-free bugs, and some undefined-behavior-related bugs may also be detected. You will get 0 mark for the test case if there is any error there. Note that if your program has no errors detected by the sanitizers, then the "Errors" tab may not appear. If you wish to check for memory leak yourself using the same options, you may follow our Checking for memory leak yourself guide.

After the deadline
We will have 46 additional test cases which won't be revealed to you before the deadline. Together with the 38 given test cases, there will then be 84 test cases used to give you the final assignment grade. All 84 test cases will be run two times as well: once without memory leak checking and once with memory leak checking. The assignment total will therefore be 84*(1+0.25) = 105. Details will be provided in the marking scheme which will be released after the deadline.

Here is a summary of the test cases for your information.

Main thing to test	Number of test cases in main before deadline (given test cases)	Number of test cases in main after deadline (given+hidden test cases)
default constructor	1	1
C-string conversion constructor	2	4
int conversion constructor	2	4
countDigitsBeforeDP and countDigitsAfterDP	2	4
isNegative	2	4
flipSign	2	4
isLargerThan	5	10
add	5	13
addToSelf	1	3
deep copy	1	3
timesTenPower	2	5
multiplyBySingleDigit	3	6
multiply	5	10
subtract	5	13

Submission and Deadline

Deadline: October 20, 2021 (Wednesday), 23:59:00 HKT (Extended)

Please submit one cpp file only: **decimal.cpp**. Submit the zip file to ZINC. ZINC usage instructions can be found here.

Notes:

- You may submit your file multiple times, but only the latest version will be graded.
- Submit early to avoid any last-minute problems. Only ZINC submissions will be accepted.
- The ZINC server will be very busy on the last day especially in the last few hours, so you should expect you would get the grading result report not-very-quickly. However, as long as your submission is successful, we would grade your latest submission with all test cases after the deadline.
- If you have encountered any server-side problem or webpage glitches with ZINC, you may post on the ZINC support forum to get attention and help from the ZINC team quickly and directly. If you post on Piazza, you may not get the fastest response as we need to forward your report to them, and then forward their reply to you, etc..

Compilation Requirement

It is **required** that your submissions can be compiled and run successfully in our online auto-grader ZINC. If we cannot even compile your work, it won't be graded. Therefore, for parts that you cannot finish, just put in dummy implementation so that your whole program can be compiled for ZINC to grade the other parts that you have done. Empty implementations can be like:

```
int SomeClass::SomeFunctionICannotFinishRightNow()
{
    return 0;
}

void SomeClass::SomeFunctionICannotFinishRightNowButIWantOtherPartsG
{
}
```

Late submission policy

There will be a penalty of -1 point (out of a maximum 100 points) for every minute you are late. For instance, since the deadline of the assignment is 23:59:00 on Oct 20th, if you submit your solution at 1:00:00 on Oct 21st, there will be a penalty of -61 points for your assignment. However, the lowest grade you may get from an assignment is zero: any negative score after the deduction due to a late penalty (and any other penalties) will be reset to zero.

FAQ

Frequently Asked Questions

Q: My code doesn't work / there is an error, here is the code, can you help me fix it?

A: As the assignment is a major course assessment, to be fair, you are supposed to work on it on your own and we should not finish the tasks for you. We might provide some very general hints to you, but we shall not fix the problem or debug for you.

Q: Can I add extra helper functions?

A: You may do so in the files that you are allowed to modify and submit. That implies you cannot add new member functions to any given class.

Q: Can I include additional libraries?

A: No. Everything you need is already included - there is no need for you to add any include statement (under our official environment).

Q: Can I use global variable or static variable such as "static int x"?

A: No.

Q: Can I use "auto"?

A: No.

Q: It is hard to create test cases for large numbers and verify the calculation result. Is there any easy way?

A: You can make use of online high-precision calculators like this. For the one we linked, choose "130" as the "Digit" and that should be able to calculate pretty big results.

Q: My program gives the correct output on my computer, but it gives a different one on ZINC. What may be the cause?

A: Usually inconsistent strange result (on different machines/platforms, or even different runs on the same machine) is due to relying on uninitialized hence garbage values, missing return statements, accessing out-of-bound array elements, improper use of dynamic memory, or relying on library functions that might be implemented differently on different platforms (such as pow() in cmath).
You may find a list of common causes and tips on debugging in the notes here.
In this particular PA, it is probably related to misuse of dynamic memory. Good luck with bug hunting!

Q: I notice we are using an int to store the number of digits, then at most how many digits will be in the test cases?

A: Since the data member "size" is an int, so the number of digits won't exceed the limit of that, but that can be pretty big. Imagine a decimal number with close to 2 to the power of 31 digits. However with this said in your implementation you should only create the content array to be just big enough to hold the digits etc. as mentioned. Don't just create a content array of a certain fixed size.

As a side-note, to really support actual "inifinite number of digits", i.e., to workaround the memory storage limitation of an "int" size, we may need to use a linked list instead or another way to denote the size of the content array (e.g. to actually use a C-string to store the content). Just FYI, you don't need to do that in this assignment.

Q: For flipSign, as it is a const function, we should return a Decimal number instead of changing the current number in the current object, right?

A: Yes. The description has been updated at 5:09pm on Oct 1st. Sorry for the confusion caused earlier.

Q: Should the test case #38 result be "86.771"?

A: Yes. There was a typo in the expected output. It has been fixed at 2:41am on Oct 2nd. A ZINC regrade has been triggered for all 8 submissions we have received so far. Sorry for the inconvenience caused.

Q: For flipSign, if the input is -123.4, what should countDigitsBeforeDP return?

A: 3 because there are 3 digits.

Q: Can I use function X in this assignment?

A: In general if you can use it without including any additional library on ZINC, then you can use it. We suggest quickly testing it on ZINC (to see if a basic usage of it compiles there) before committing to using it as library inclusion requirement may differ on different environments.