Philipp Herrmann

philipp@schickslos.de

Expose

This documentation provides useful information about the content and technical background of the namespace BigInts. It empowers you to work with types that store numbers with up to 4932 decimal places. Methods and functions perform the basic arithmetic operations. Also included are arithmetic and comparison operators, type converters, definitions of maximal and minimal values, square root and logarithm.

Bigints documentation

Documentation of the namespace BigInts

Content

* General information about BigInts
* Files and types included
* Introduction into binary numbers
* The interface of BigInts
* Constructors
* Properties
* Methods
* Operators
* Functions
* Converters
* Preprocessor statements
* Summary

General information about BigInts

Most languages provide basic types including types for numbers, in this case it’s about integers, unsigned integers (which means non-negative numbers), and floating-point numbers. Let’s have a deeper look at unsigned integers, computers store numbers in binary form (more about them later), the affected types use a specific number of bits (computational basic unit which value can be either 1 or 0). The build-in types reserve up to 64 bits, the bits can exist in 2^64 different combinations of states, so you can store values in the area 0 to 2^64 – 1 (cause 0 is one of the values, the max value is 2^64 - 1). The type I just talked about corresponds with the “unsigned long long” in C++, also known as “uint64\_t”, in decimal representation this is about 18 quintillion.

Now you might wonder why to create types for numbers that can grow larger than 2^64, the answer is quite simple: There are not that many classical apps which require them, but they are used very much in cryptography (RSA). The biggest one of the types I created can store a number with up to 4932 decimal places, as I mentioned already. This type reserves 16384 bits, this is equal to 2KB, arithmetic operations on it take much time, which will have a measurable impact on performance.

Comparing advantages and physical limitations, I decided to create eight types and to equip them with converters.

Files and types included

The package includes eight .h (header files), each linked with its .cpp file. The types can store unsigned integers build from 128, 256, 512, 1024, 2048, 4096, 8192 and 16384 bits. The interfaces are quite similar, the types can be converted to higher types e.g. uint128 to uint256. There exist methods to convert back to lower types, which will raise exceptions if the affected object’s value is to large. Every type is in a namespace with the same name and all the namespaces are summarized in a namespace called “BigInts”.

Introduction into binary numbers

We are using the decimal system in our daily life; it provides the chars 0 – 9. If we want to add 1 to our number, we will increase the last char, in case it’s 9, we change it to 0, and increase the char before.

Since the base of computational stored information is a bit (status either 1 or 0), we are used to represent our number in a combination of this states. The principle is like the decimal system, but now we must increase the char before the last char already, when the latter has the value one. The first ten numbers in binary and decimal representation are listed below.

|  |  |
| --- | --- |
| Decimal representation | Binary representation |
| 0 | 0 |
| 1 | 1 |
| 2 | 10 |
| 3 | 11 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |
| 8 | 1000 |
| 9 | 1001 |

The interface of BigInts

All the BigInts provide constructors which are based on: first, a signed integer, if its value is lower than 0, there will be an exception thrown, second, an unsigned integer. Uint64\_t, and int64\_t can be used to pass bigger numbers. Added to that, you can use a string build from ones and zeros, if other chars are included, the constructor will raise an exception. You also can pass a bit set of the same dimension as the object you are just creating. BigInts of higher dimensions take other BigInts of lower dimensions, and of course copy constructors exist.

The next point are the standard arithmetic methods, all of them (addition, subtraction, multiplication and division) are available with each three different signatures. The first one requests an uint64\_t, this is necessary cause it should be useable combined with literals (values written in source code directly). The second one needs another BigInt of the same dimension, and the last one takes an initializer list containing BigInts (same dimension as the current object).

First of these methods is addition. The uint\_64\_t will be converted to a BigInt, and the initializer list calls the method with each of its elements, at least the “real” logic is in the method that takes a BigInt as argument. When it is called it runs a for-Block, which reaches all its bits, the following gate will be applied to the bits.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BigInt 1 | BigInt 2 | Transmitter | Result | Next Transmitter |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 |

This might look confusing but its right simple. If you add two values manually, and in one of the columns the number grows larger than 9, you write a 1 in the next column, in the current column you start over with 0, in our case, the transmitter is these little one, and we set its value to one, when the result in our current column grows larger than 1.

Let’s explain these table in words, if the first, the second, and the transmitter all are 1, the corresponding bit in the result will be set to 1, and the next transmitter too. The next three rows describe the case in which two of maximal three values are one, consequently the result will be set to 0, and the next transmitter to 1. Row 5 to 7 show the reaction if one value is 1, and two values are 0, the result will accept the value of 1, and the transmitter is set to 0. If all bits are 0, result and transmitter will be 0 too.

If we are evaluating the bit sets at the last index, and the next transmitter will become value 1, the method throws an exception, cause the result will grow larger than the maximal value of the BigInt.

The second arithmetic method is subtraction. It also uses a transmitter and is based on the same for-block as addition. The important difference comes up if the transmitters value is 1, it means that the current value will be **de**creased by 1 not **in**creased. Further information can be found in the gate applied to the bits.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BigInt 1 | BigInt 2 | Transmitter | Result | Next Transmitter |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 |

The explanation clarifies the results. First case all bits are one, this means: 1 (affected item) – 1 (argument) – 1 (transmitter) = -1. To represent -1, we set the result to one, but we must decrease the result of the next operation by one, so we set the transmitter to 1 too. Second and third case: 1 – 1 – 0 = 1 – 0 - 1 = 0, result is 0, and no further settings are necessary, the transmitter is 0 too. Case no. 4: 0 – 1 - 1 = -2, result is 0, and transmitter is one. Case no. 5: 1 – 0 – 0 = 1, result = 1 and transmitter = 0. 6th an 7th case corresponds with case one. The last case: 0 – 0 – 0 = 0, result = 0, transmitter = 0. If the for-block evaluates the most significant bit, and the transmitter becomes 1, an exception occurs saying the result will fall under 0, which can’t be expressed by our BigInts

It is possible to perform multiplication by rewriting the task as addition e.g. 3\*4 = 3+3+3+3. BigInts don’t use this trick, they perform multiplication as a separate operation, which is not that familiar with addition and subtraction. The method first declares a forward list, it will save values which sum represents the result of multiplication. Next step is the declaration of four variables, two integers initialized with 0, and two bools initialized with one. The integers values represent the number of valid bits (the max number of bits minus the number of 0s in front of the first 1). The bools indicate if this number was found. If so, we can make sure if the result will grow larger than the maximum, by evaluating the inequation:

Number of valid bits factor one + valid bits factor two – 1 > Max number of bits

If it returns true, our result will grow to large and the method throws an exception. If not, we can start the multiplication itself, it’s made up by two nested for-blocks. The first for-block accesses every bit of the number, if its value is one, the second for-block will be executed. It also runs once for every bit. Note, the first block runs for one of the factors, the second one treats the other factor. So, if the bits value was one, we will take the other numbers value, and we introduce a new variable, which we fill with 0s, the number of 0s we add is indicated by the outer for-block. After the number of 0s we want to add was reached (int code: if (x < i) returns false), we enter our original value, the result is added to a forward list. These values will be added now, finally result of addition is our result. If while performing the addition, the value grows larger than the max value of the BigInt, the add-method will throw the exception needed.

Now, let’s clear up the process of division. First, we check that none of the following cases is satisfied. Case 1: the program wants to divide by 0, that’s a mathematical error, cause division by 0 is not defined. Case 2: if the number we divide by, is bigger than the caller of the function, if so, we return the value 0 (it’s all about integers, we can’t return a floating-point number). If these cases are not satisfied, we can continue in code. We introduce four strings and two bools now, don’t be frustrated, there task is quite simple. “ThisStream” and “InitStream” contain the binary information of our numbers. The 0s in front of the first one all are removed, so their lengths are not necessarily equal. The next variable is called “ThisCurrentStream” it holds the bits currently handled; I will explain its function in depth later. The last string represents our future result, decoded as string. Added to that we also introduced two bools (= Booleans), they are needed when assigning values to “ThisStream” and “InitStream” only (They indicate when the first one in our streams appeared, it’s just about helpers). Now we assign the values to “This-” and “InitStream”, we need to perform a for-block which handles each bit in our number, in this case it must start with the most significant bit, so we access our bit inside the bit set at index [Max number of bits - 1 - counter variable of the for-block]. When the first bit with value = 1 was detected, we set the corresponding bool = true, this affects that significant 0s which could follow will be copied into the strings. We want to save time, so we perform these operations for both strings at once.

Armed with our strings, we can start the main part of division now. But first we must assign a value to the string called “ThisCurrentStream”, this value is “1”. After that, we delete the first char of “ThisStream” (which was one). Maybe you realized the principle already, during the operation “ThisCurrentStream” will grow, we always move the first char of “ThisStream” after the last char of “ThisCurrentStream”.

The following while-block will be repeated if “ThisStream” still contains chars (remember we will delete chars). Now, if “InitStream” is smaller or equal to “ThisCurrentStream”, we add “1” to the result, perform the calculation “ThisCurrentStream” – “InitStream”, we need to convert the strings int BigInts, if we don’t do so, we can’t profit from our subtraction-Method. The result of the substraction will be assigned to “ThisCurrentStream”. In case the condition is not satisfied (“InitStream” is bigger than “ThisCurrentStream”), we add 0 to our result, and we add the first char of “ThisStream” to “ThisCurrentStream” (don’t forget, we delete the first char of “ThisStream” now).

When our while-block stopped running, we can do the last if-check, remove the insignificant 0s, and convert our result to a BigInt via the string-constructor.

If you have understood the basic arithmetic operations on BigInts, the rest will become more and more easy, because we profit from all the functions existing already. The first example of “higher” operations is the power-method. When describing multiplication, I said, we will not change the calculation into an addition. In case of the power-method we will do exactly this, e.g. 3³ = 3\*3\*3. So, there isn’t much to talk about, we take the result, and multiply it with our former value, until the number of repetitions required was reached.

Next operation is the logarithm. The Search algorithm applied by BigInts returns the result in max. log2(log2(max. Value of affected uintX-lass)). In case of uint128 its about 7 comparisons, in case of uint16384 you need 16 comparisons in worst case. First the method makes sure two specific cases are not satisfied by the arguments, if either the calling object or the argument (base) is 0, the operation returns an exception, referring to an arithmetic error. A third if-block checks if the calling object is smaller then the argument, if so, the method returns 0 (remember, integers never can handle floating-point numbers). The first “formal” step of the method declares two integer values, “CurrentPosition” its value is the result of: