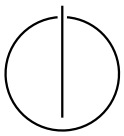


TU Munich, Department of Informatics
Chair XXV: Data Science and Engineering
Prof. Dr. Thomas Neumann

Exercise for the *Practical Course: Systems Programming in C++* Summer Term 2021

Michael Freitag, Moritz Sichert ({freitagm,sichert}@in.tum.de)
https://db.in.tum.de/teaching/ss21/c++praktikum

Sheet Nr. 04 (due on 17.05.2021 at 23:59)

Create a fork of the Git repository at `https://gitlab.db.in.tum.de/cpplab21/sheet04`. The repository contains a subdirectory for each exercise on this sheet, into which you should put your answers. **Do not forget to `git push` your solution before the deadline expires.**

Once the deadline for this sheet expires, a signed tag will be created in your fork automatically. Your solutions will be graded based on the state of your repository at this tag. **Do not attempt to modify or remove this tag, as we cannot grade your solution otherwise.**

Exercise 1 (40 points)

In this exercise you will implement two arithmetic types: complex and rational numbers. In both cases you need to define and implement a class with suitable members in the `arithmetic` namespace using the program scaffold contained in the subdirectory for this exercise.

Your code should contain some minimal documentation, similar to the documentation provided as part of the program scaffolds in the previous assignments.

(a) Implement a class named `Complex` representing complex numbers with `double` real and imaginary parts. Your class should support at least the following operations:

- Constructors
  - Default-construction to 0
  - Implicit conversion from a real number
  - Construction from real and imaginary parts
- Member functions
  - `real`: return the real part
  - `imag`: return the imaginary part
  - `abs`: return the absolute value
  - `norm`: return the squared absolute value
  - `arg`: return the argument in radians (angle between the real axis and the number)
  - `conj`: return the complex conjugate
- Operators
  - Unary minus
  - Unary plus
  - Binary minus
  - Binary plus
  - Multiplication
  - Division
  - Equality and inequality

Your implementation may use math functions from the `<cmath>` header. You do not have to manually implement all required comparison operators, but they should all work as expected.

(b) Implement a class named `Rational` representing rational numbers with `long long` numerators and denominators. Your class should support at least the following operations:

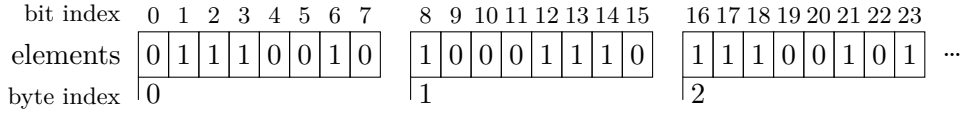
- Constructors
  - Default-construction to 0
  - Implicit conversion from an integer
  - Construction from numerator and denominator
- Member functions
  - `num`: return the numerator
  - `den`: return the denominator
  - `inv`: return the inverse of the rational number
- Operators
  - Unary minus
  - Unary plus
  - Binary minus
  - Binary plus
  - Multiplication
  - Division
  - Equality and inequality
  - Relational operators (`<, <=, >, >=, <=>`)
  - Explicit conversion to `double`

The class should store rational numbers in a canonical form, i.e. the numerator and denominator should be fully canceled (e.g. $6/12 = 1/2$). Negative numbers should have a negative numerator and a positive denominator, otherwise both the numerator and denominator should be positive. Your implementation of canonicalization does not need to be particularly efficient but you should avoid unnecessary canonicalization (e.g. when computing the inverse of a canonical rational number). You do not have to manually implement all required comparison operators, but they should all work as expected.

Exercise 2

(60 points)

Arrays of `bool` can be implemented quite efficiently by exploiting the fact that each `bool` can be represented by just one bit. This allows us to store 8 `bool` values within a single byte (assuming one byte has exactly 8 bits). The resulting data structure is called a *bitset*.



In this exercise, you will implement a bitset data structure which uses n bytes to store up to $8n$ `bool` elements, as shown in the illustration above. Note that the C++ standard library already has `std::vector<bool>`, `std::bitset`, and similar types that you should not use in your implementation (using `std::vector` to store bytes is fine).

Your code should contain some minimal documentation, similar to the documentation provided as part of the program scaffolds in the previous assignments.

(a) Implement a class named `BitSet` in the `bitset` namespace, using the program scaffold contained in the subdirectory for this exercise. Your class should support at least the following operations.

- Constructors
  - Default-constructor: Initialize an empty bitset.
  - Construction with a specified size: Initialize a bitset containing the specified number of `false` elements.
- Member functions
  - `size`: Return the total number of elements in the bitset.
  - `cardinality`: Return the number of `true` elements in the bitset.
  - `push_back`: Insert an element at the end of the bitset.
  - `front`: Access the first element.
  - `back`: Access the last element.
- Operators
  - Subscript: Access a specific element

Take care to use the most restrictive cv-qualifiers possible and define suitable `const` and non-`const` overloads for the `front` and `back` functions as well as for the subscript operator. Add suitable assertions in each function.

The `const` overloads of `front`, `back`, and the subscript operator can simply return a `bool`. However, the non-`const` overloads must return an object that acts as a reference to an individual bit, so that it is possible to write, for example, `my_bitset[10] = false` or `my_bitset[10] = my_bitset[11]`. Since individual bits are not actually stored in a dedicated variable of type `bool` in our implementation, we cannot directly return a reference to an individual bit. Instead, you have to implement the nested class-type `BitReference` which should be used as the return value of the non-`const` overloads. It should support the following operations:

- Constructors
  - Private constructor with suitable parameters that identify an individual bit (e.g. byte-pointer and bit offset)
- Operators
  - Implicit conversion to `bool`
  - Assignment from `bool`
  - Assignment from `const BitReference&`

Since the helper class is implicitly convertible to `bool`, it is valid to write code like `bool a = my_bitset.front()`. Since it also defines an assignment operator from `bool`, it is valid to write code like `my_bitset.front() = true`. Finally, since it defines an assignment operator from `const BitReference&`, it is valid to write code like `my_bitset.front() = my_bitset.back()`.

At this point, our bitset container is already useful but it is missing one important feature: We cannot use it in range-for statements. For this, we need to implement a suitable *iterator* along with some more functions.

Iterators are an important concept in the C++ standard library which will be covered in more detail later during the lecture. Iterators are typically nested classes that implement the logic required to identify a specific element within a container along with operations to move the iterator. In our case, we can use an index and a pointer to the first byte of storage to identify individual bits. Moving the iterator can be accomplished by updating the index accordingly.

(b) Implement a nested class `BitIterator` within the `BitSet` class. This class should publicly expose the following members:

- Nested type aliases
  - `difference_type`: alias for `std::ptrdiff_t`
  - `value_type`: alias for `bool`
  - `reference`: alias for `BitReference`
  - `iterator_category`: alias for `std::bidirectional_iterator_tag`
- Constructors
  - Default construction
  - Construction from an index and a pointer to the first byte of storage
- Operators
  - Equality: Check if two iterators point to the same bit
  - Dereference: Return a `BitReference` referencing the current bit
  - Pre- and Post-Increment (`++it` and `it++`): Move the iterator to the next bit
  - Pre- and Post-Decrement (`--it` and `it--`): Move the iterator to the previous bit

Add `begin` and `end` functions to the `BitSet` class which return a `BitIterator` that points to the first bit, and past the last bit, respectively.

The `BitIterator` class follows some specific requirements that are imposed on iterators by the C++ standard through the `std::bidirectional_iterator` concept. Together with the `begin` and `end` functions, this allows us to use the `BitSet` class in range-for statements (take a look at the test cases for this exercise, for example).

(c) As it is possible to change individual elements in the bitset through the `BitIterator` class (e.g. `*bitset.begin() = false`), they cannot be used with `const` instances. In order to support iteration over `const BitSet`s, we need to implement another suitable iterator class which does not permit modification of the elements.

Implement a nested class `ConstBitIterator` within the `BitSet` class, which publicly exposes the same members as the `BitIterator` class from task (b), with the following exceptions.

- The `reference` nested type alias should be an alias for `bool`
- The dereference operator should return `bool`

Add `const`-qualified `begin` and `end` functions to the `BitSet` class which return objects of type `ConstBitIterator` that point to the first bit, and past the last bit, respectively. This allows us to also use `const BitSet`s in range-for statements.