A\*. Vector

|  |  |
| --- | --- |
| Ограничение времени | 20 секунд |
| Ограничение памяти | 128.0 Мб |
| Ввод | стандартный ввод или input.txt |
| Вывод | стандартный вывод или output.txt |

<https://gitlab.com/ibr11/cppmipt_spring2022/-/tree/master/vector>

Пришлите архив с файлом vector.h и, возможно, другими файлами реализации.

Vector

*Ранее в курсе был рассмотрен класс динамической строки std::string (см. задание String), которая увеличивает размер массива символов по мере необходимости и самостоятельно управляет выделяемой памятью. В этом задании мы рассмотрим класс std::vector, являющийся обобщением динамической строки на произвольный тип, то есть класс динамического массива.*

std::vector

Интерфейс и реализация std::vector<T> во многом похожи на std::string. Ключевым отличием является семантика работы метода reserve (и остальных методов, которые приводят к увеличению capacity). При резервировании памяти под большее число объектов, выделяется "сырая" (неинициализированная) память достаточная для хранения нужного числа дополнительных объектов. Пустые ячейки заполняются лениво по мере необходимости. То есть, если size < capacity, то это значит, что первые size \* sizeof(T) байт реально заняты объектами, а последние (capacity - size) \* sizeof(T) байт пусты - объектов там не создано (как такого добиться - см. задание Optional). Это нужно, например, для того, чтобы можно было создавать вектор из объектов, у которых нет конструктора по умолчанию (а как бы тогда нужно было инициализировать неиспользуемые ячейки?):

struct A {

int x;

A() = delete;

explicit A(int x\_param) : x(x\_param) {

}

};

std::vector<A> v;

v.reserve(1000); // объекты A не создаются! Выделяется "сырая" память размера 1000 \* sizeof(A)

for (int i = 0; i < 1000; ++i) {

v.push\_back(A(i));

// а лучше v.emplace\_back(i);

}

Подробности на лекциях, семинарах, в чатах, на заборе, а также [в справочнике](https://en.cppreference.com/w/cpp/container/vector).

Детали реализации

От вас требуется реализовать шаблонный класс Vector с единственным шаблонным параметром - типом хранящихся элементов. При реализации можно (и даже нужно) пользоваться обобщенными алгоритмами из STL (std::copy, std::fill и т.п.), но нельзя использовать стандартные контейнеры. Будет проверяться корректность мультипликативной схемы расширения массива с коэффициентом 2. В базовой версии ручное управление временем жизни объектов не требуется (см. доп. задание). Класс должен поддерживать следующий функционал:

* Конструктор по умолчанию - создает пустой массив;
* Явный конструктор от числа - создает массив заданного размера заполненный объектами, сконструированными по умолчанию;
* Конструктор, принимающий size и value (именно в этом порядке) - создает массив длины size, заполненный элементами со значением value;
* Шаблонный конструктор, принимающий пару итераторов - создает копию переданного диапазона;

**Важно:** объявление этого конструктора должно иметь вид

template <class Iterator, class = std::enable\_if\_t<std::is\_base\_of\_v<std::forward\_iterator\_tag, typename std::iterator\_traits<Iterator>::iterator\_category>>>

Vector(Iterator first, Iterator last)

Это делает конструктор доступным только в случае, когда на вход приходят два Forward итератора.

* Конструктор от std::initializer\_list;
* Правило "пяти";
* Методы Size, Capacity, Empty;
* Константный и неконстантный оператор доступа по индексу []. Неконстантный должен позволять изменять полученный элемент;
* Константный и неконстантный метод доступа по индексу At. При выходе за границы массива должен бросать исключение std::out\_of\_range;
* Методы Front() и Back()
  + доступ к первому и последнему элементам (тоже по две версии).
* Метод Data()
  + возвращает указатель на начало массива.
* Метод Swap(other)
  + обменивает содержимое с другим массивом other;
* Метод Resize(new\_size)
  + изменяет размер на new\_size. Если вместимость не позволяет хранить столько элементов, то выделяется новый буфер с вместимостью new\_size. Недостающие элементы конструируются по умолчанию.
* Метод Resize(new\_size, value)
  + то же, что и Resize(new\_size), но в случае new\_size > size заполняет недостающие элементы значением value.
* Метод Reserve(new\_cap)
  + изменяет вместимость на max(new\_cap, текущая вместимость). Размер при этом не изменяется.
* Метод ShrinkToFit()
  + уменьшает capacity до size.
* Метод Clear()
  + устанавливает размер в 0, очищения выделенной памяти при этом НЕ происходит.
* Методы PushBack(const T&) и PushBack(T&&)
  + добавляет новый элемент в конец массива.
* Метод PopBack()
  + удаляет последний элемент.
* Операции сравнения (<, >, <=, >=, ==, !=), задающие лексикографический порядок.

Также реализуйте поддержку итераторов и методы для работы с ними: begin(), end(), cbegin(), cend(), rbegin(), rend(), crbegin(), crend(). begin()-end(), rbegin()-rend() должны иметь две версии, возвращающие константные и неконстантные итераторы.

Внутри класса Vector определите типы-члены ValueType, Pointer, ConstPointer, Reference, ConstReference, SizeType, Iterator, ConstIterator.

Гарантии безопасности

Спроектируйте класс так, чтобы при возникновении исключений (нехватка памяти, ошибка при копировании объектов и т.п.) методы удовлетворяли строгой гарантии безопасности. То есть все методы должны работать атомарно: либо работать без сбоев, либо не изменять исходного состояния вектора. Исключение: копирующее присваивание может давать лишь базовую гарантию безопасности.

**При решении можно предполагать, что конструкторы перемещения, перемещающие присваивания и деструкторы объектов никогда не бросают исключений!**

B\*. UnorderedSet

|  |  |
| --- | --- |
| Ограничение времени | 2 секунды |
| Ограничение памяти | 256.0 Мб |
| Ввод | стандартный ввод или input.txt |
| Вывод | стандартный вывод или output.txt |

<https://gitlab.com/ibr11/cppmipt_spring2022/-/tree/master/unordered_set>

Пришлите архив с файлом unordered\_set.h и, возможно, другими файлами реализации.

UnorderedSet

Реализуйте шаблонный класс UnorderedSet, аналог [std::unordered\_set](https://en.cppreference.com/w/cpp/container/unordered_set), который основан на хешировании методом цепочек с динамическим увеличением числа корзин.

Метод цепочек предполагает хранение массива, в каждой ячейке которого лежит корзина - список вставленных элементов с одинаковыми хеш-значениями. Среднее время операций над такой структурой пропорционально степени загруженности таблицы (load\_factor = n\_elements / n\_buckets). При превышении load\_factor некоторого значения (в задании = 1) происходит перехеширование - создание нового массива корзин в, например, 2 раза большего размера и перенос старых элементов в новые корзины.

В простейшем варианте достаточно хранить вектор (std::vector) списков (std::list / std::forward\_list) и выполнять операции над ним. Более эффективные техники описаны ниже в дополнительных заданиях.

Базовая часть

Шаблонный класс UnorderedSet должен быть параметризован типом ключа KeyT. Стратегия расширения такая же как в задаче String - при добавлении элемента в пустую таблицу число корзин становится равным 1, при добавлении элемента в полную таблицу число корзин увеличивается в 2 раза (таблица считается полной, если число корзин совпадает с числом элементов, то есть load\_factor == 1). В качестве хеш-функции воспользуйтесь std::hash. std::hash отображает объекты в диапазон size\_t, чтобы получить индекс корзины, возьмите остаток от деления полученного числа на количество корзин.

Набор методов включает:

* Конструктор по умолчанию. Создает пустую хеш-таблицу.
* Конструктор от числа корзин count. Создает хеш-таблицу с count пустыми корзинами.
* Конструктор от промежутка заданного двумя Forward итераторами. Создает хеш-таблицу с числом корзин равным числу элементов в последовательности, а затем вставляет элементы в таблицу.
* Конструкторы копирования, перемещения, а также присваивания должны работать корректно.
* Методы Size, Empty, Clear с привычной семантикой.
* Методы вставки Insert(const KeyT&), Insert(KeyT&&).
* Метод удаления Erase(const KeyT&).
* Метод поиска bool Find(const KeyT&).
* Метод Rehash(new\_bucket\_count). Изменяет число корзин в хеш-таблице с перехешированием. Если new\_bucket\_count совпадает с текущим количеством корзин или меньше числа элементов (load\_factor становится больше 1), то ничего делать не нужно.
* Метод Reserve(new\_bucket\_count). То же, что и Rehash, но не уменьшает число корзин, то есть срабатывает, если new\_bucket\_count превышает текущее количество корзин.
* Методы BucketCount (возвращает число корзин в таблице), BucketSize(id) (возвращает размер корзины с номером id), Bucket(key) (возвращает номер корзины, в которую попадает объект key), LoadFactor() (возвращает степень заполненности таблицы).

Обратите внимание, что каждый раз создавать новые узлы списков при перехешировании может быть неэффективно. Для переиспользования старых узлов (перебрасывания указателей старых списков) рассмотрите возможность использования метода [std::list<T>::splice](https://en.cppreference.com/w/cpp/container/list/splice).