Правительство Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Отчёт к лабораторной работе №12

по дисциплине

«Языки программирования»

Работу выполнила

Студент группы СКБ222 _____ М. Х. Халимов
подпись, дата

Работу проверил _____ С. А. Булгаков
подпись, дата

Москва 2023

Содержание

Постановка задачи

Разработать шаблоны классов *Vector*, описывающий контейнер типа массив динамического размера, а также *Map*, описывающий контейнер типа отображение, параметр шаблона - тип данных для хранения значений. Интерфейс и реализацию разместить в файлах *vector.hpp* и *map.hpp*. Классы должны содержать итератор произвольного доступа и двунаправленный, соответственно (при реализации, класс *std::iterator* не использовать).

Классы должны обладать интерфейсом, аналогичным классам *std::vector* и *std::map*. Для класса *Map* использовать *CRC-32* в качестве хеш-функции.

В основной функции, размещенной в файле *main.cpp*, продемонстрировать применение разработанных классов и их методов совместно с алгоритмами библиотеки *STL*.

1. Описание шаблона классов *vector* и *map*

Объявляются классы *vector* и *map*. Класс *vector* является совокупностью полей *value_type, size_type, reference, const_reference, pointer, const_pointer* и методов *operator=(), capacity(), size(), empty(), operator[], at(), front(), back(), data(), clear(), push_back(), pop_back(), reserve(), resize(), swap()*. Класс *map* является совокупностью полей *keys_, values_, size_* и методов *operator[], size(), empty(), clear(), erase(), contains(), at(), lower_bound(), insert()*. В результате получаются вектор и отображение.

2. Описание методов класса

Описание всех методов классов *map* и *vector* находиться на cppreference.com. методы итератора описаны в *Лабораторной работе №12*.

3. Функция main

Функция *main* включает в себя создание объектов класса *main* и вызов методов этого же класса. В результате работы функции в консоль выводится пример выполнения методов, описанных выше.

4. Результаты тестирования программы

Листинг

```
Our numbers 1 2 3
First number plus 2 is 3
```

Приложение А

Исходный код класса vector

```
#include <cstddef>
#include <stdexcept>

template <typename T>
class Vector {
public:
    using value_type = T;
    using size_type = std::size_t;
    using reference = T&;
    using const_reference = const T&;
    using pointer = T*;
    using const_pointer = const T*;

    // конструкторы
    Vector() : m_size(0), m_capacity(0), m_data(nullptr) {}
    explicit Vector(size_type size) : m_size(size), m_capacity(size),
m_data(new T[size]) {}
    Vector(size_type size, const T& value) : m_size(size),
m_capacity(size), m_data(new T[size]) {
        for (size_type i = 0; i < m_size; ++i) {
            m_data[i] = value;
        }
    }


     Vector(std::initializer_list<T> init) : Vector() {
          for (const auto& x : init) {
               push_back(x);
          }
     }

    Vector(const Vector& other) : m_size(other.m_size),
m_capacity(other.m_capacity), m_data(new T[other.m_capacity]) {
        for (size_type i = 0; i < m_size; ++i) {
            m_data[i] = other.m_data[i];
        }
    }
    Vector(Vector&& other) noexcept : m_size(other.m_size),
m_capacity(other.m_capacity), m_data(other.m_data) {
        other.m_size = 0;
        other.m_capacity = 0;
        other.m_data = nullptr;
    }
    template <typename InputIterator>
```

```
    Vector(InputIterator first, InputIterator last) : m_size(last -
first), m_capacity(last - first), m_data(new T[last - first]) {
        for (size_type i = 0; first != last; ++first, ++i) {
            m_data[i] = *first;
        }
    }
    // деструктор
    ~Vector() {
        delete[] m_data;
    }
    // оператор присваивания
    Vector& operator=(const Vector& other) {
        if (this != &other) {
            Vector tmp(other);
            swap(tmp);
        }
        return *this;
    }
    Vector& operator=(Vector&& other) noexcept {
        if (this != &other) {
            swap(other);
            other.clear();
        }
        return *this;
    }
    // методы размерности
    size_type size() const {
        return m_size;
    }
    size_type capacity() const {
        return m_capacity;
    }
    bool empty() const {
        return m_size == 0;
    }
    // методы доступа
    reference operator[](size_type index) {
        return m_data[index];
    }
    const_reference operator[](size_type index) const {
        return m_data[index];
    }
    reference at(size_type index) {
        if (index < m_size) {
            return m_data[index];
        }
        throw std::out_of_range("Vector::at");
    }
    const_reference at(size_type index) const {
        if (index < m_size) {
            return m_data[index];
        }
```

```
    throw std::out_of_range("Vector::at");
}
reference front() {
    return m_data[0];
}
const_reference front() const {
    return m_data[0];
}
reference back() {
    return m_data[m_size - 1];
}
const_reference back() const {
    return m_data[m_size - 1];
}
pointer data() {
    return m_data;
}
const_pointer data() const {
    return m_data;
}
// методы модификации
void clear() {
      resize(0);
 }
 void push_back(const T& value) {
      if (m_size == m_capacity) {
           reserve(m_capacity == 0 ? 1 : m_capacity * 2);
      }
      m_data[m_size++] = value;
 }
 void pop_back() {
      if (m_size > 0) {
           --m_size;
      }
 }
 void reserve(size_type new_capacity) {
      if (new_capacity > m_capacity) {
           T* new_data = new T[new_capacity];
           for (size_type i = 0; i < m_size; ++i) {
                new_data[i] = m_data[i];
           }
           delete[] m_data;
           m_data = new_data;
           m_capacity = new_capacity;
      }
 }
 void resize(size_type new_size) {
      if (new_size > m_capacity) {
           reserve(new_size);
      }
      if (new_size > m_size) {
           for (size_type i = m_size; i < new_size; ++i) {
```

```
            m_data[i] = T();
            }
        }
        m_size = new_size;
    }
    void resize(size_type new_size, const T& value) {
        if (new_size > m_capacity) {
            reserve(new_size);
        }
        if (new_size > m_size) {
            for (size_type i = m_size; i < new_size; ++i) {
            m_data[i] = value;
            }
        }
        m_size = new_size;
    }
    void swap(Vector& other) noexcept {
        std::swap(m_size, other.m_size);
        std::swap(m_capacity, other.m_capacity);
        std::swap(m_data, other.m_data);
    }


    // итераторы
    class iterator {
        public:
         using value_type = T;
         using reference = T&;
         using pointer = T*;
         using difference_type = std::ptrdiff_t;
         using iterator_category = std::random_access_iterator_tag;
             iterator() : m_ptr(nullptr) {}
        explicit iterator(pointer ptr) : m_ptr(ptr) {}

        reference operator*() const {
            return *m_ptr;
        }
        pointer operator->() const {
            return m_ptr;
        }
        iterator& operator++() {
            ++m_ptr;
            return *this;
        }
        iterator operator++(int) {
            iterator tmp(*this);
            ++m_ptr;
            return tmp;
        }
        iterator& operator--() {
            --m_ptr;
            return *this;
```

```
        }
        iterator operator--(int) {
            iterator tmp(*this);
            --m_ptr;
            return tmp;
        }
        iterator& operator+=(difference_type n) {
            m_ptr += n;
            return *this;
        }
        iterator operator+(difference_type n) const {
            return iterator(m_ptr + n);
        }
        iterator& operator-=(difference_type n) {
            m_ptr -= n;
            return *this;
        }
        iterator operator-(difference_type n) const {
            return iterator(m_ptr - n);
        }
        difference_type operator-(const iterator& other) const {
            return m_ptr - other.m_ptr;
        }
        reference operator[](difference_type n) const {
            return *(m_ptr + n);
        }
        bool operator==(const iterator& other) const {
            return m_ptr == other.m_ptr;
        }
        bool operator!=(const iterator& other) const {
            return m_ptr != other.m_ptr;
        }
        bool operator<(const iterator& other) const {
            return m_ptr < other.m_ptr;
        }
        bool operator>(const iterator& other) const {
              return m_ptr > other.m_ptr;
         }
         bool operator<=(const iterator& other) const {
              return m_ptr <= other.m_ptr;
         }
         bool operator>=(const iterator& other) const {
              return m_ptr >= other.m_ptr;
         }
        private:
        pointer m_ptr;
};

     void erase(iterator pos) {
               if (pos == end()) {
                    return;
               }
```

```
            for (iterator i = pos; i != end() - 1; ++i) {
                *i = *(i + 1);
            }

            pop_back();
    }

    void erase(iterator first, iterator last) {
        if (first == last) {
            return;
        }

        for (iterator i = first; i != last; ++i) {
            erase(i);
            --last;
        }
    }

class const_iterator {
public:
    using value_type = T;
    using reference = const T&;
    using pointer = const T*;
    using difference_type = std::ptrdiff_t;
    using iterator_category = std::random_access_iterator_tag;

    const_iterator() : m_ptr(nullptr) {}
    explicit const_iterator(pointer ptr) : m_ptr(ptr) {}
    const_iterator(const iterator& other) : m_ptr(other.m_ptr) {}

    reference operator*() const {
        return *m_ptr;
    }
    pointer operator->() const {
        return m_ptr;
    }
    const_iterator& operator++() {
        ++m_ptr;
        return *this;
    }
    const_iterator operator++(int) {
        const_iterator tmp(*this);
        ++m_ptr;
        return tmp;
    }
    const_iterator& operator--() {
        --m_ptr;
        return *this;
    }
    const_iterator operator--(int) {
        const_iterator tmp(*this);
```

```
    --m_ptr;
    return tmp;
}
const_iterator& operator+=(difference_type n) {
    m_ptr += n;
    return *this;
}
const_iterator operator+(difference_type n) const {
    return const_iterator(m_ptr + n);
}
const_iterator& operator-=(difference_type n) {
    m_ptr -= n;
    return *this;
}
const_iterator operator-(difference_type n) const {
    return const_iterator(m_ptr - n);
}
difference_type operator-(const const_iterator& other) const {
    return m_ptr - other.m_ptr;
}
reference operator[](difference_type n) const {
    return *(m_ptr + n);
}
bool operator==(const const_iterator& other) const {
    return m_ptr == other.m_ptr;
}
bool operator!=(const const_iterator& other) const {
    return m_ptr != other.m_ptr;
}
bool operator<(const const_iterator& other) const {
    return m_ptr < other.m_ptr;
}
bool operator>(const const_iterator& other) const {
    return m_ptr > other.m_ptr;
}
bool operator<=(const const_iterator& other) const {
    return m_ptr <= other.m_ptr;
}
bool operator>=(const const_iterator& other) const {
    return m_ptr >= other.m_ptr;
}

 private:
      pointer m_ptr;
 };

 // методы для работы с итераторами
 iterator begin() {
      return iterator(m_data);
 }
 iterator end() {
      return iterator(m_data + m_size);
```

```
    }
    const_iterator begin() const {
        return const_iterator(m_data);
    }
    const_iterator end() const {
        return const_iterator(m_data + m_size);
    }
    const_iterator cbegin() const {
        return const_iterator(m_data);
    }
    const_iterator cend() const {
        return const_iterator(m_data + m_size);
    }

    private:
        size_type m_size;
        size_type m_capacity;
        pointer m_data;
};

// функции для работы с вектором

template <typename T>
void swap(Vector<T>& lhs, Vector<T>& rhs) noexcept {
    lhs.swap(rhs);
}

template <typename T>
bool operator==(const Vector<T>& lhs, const Vector<T>& rhs) {
    return lhs.size() == rhs.size() &&
        std::equal(lhs.begin(), lhs.end(), rhs.begin());
}

template <typename T>
bool operator!=(const Vector<T>& lhs, const Vector<T>& rhs) {
    return !(lhs == rhs);
}
```

Приложение Б

Исходный код класса map

```
#include "vector.hpp"
#include "crc32.hpp"
#include <iostream>
#include <algorithm>
#include <iterator>
#include <utility>


template <class Key, class T>
class Map {
 private:
    typedef unsigned long uint32_t;

    Vector< std::pair<uint32_t, T> > data;
    CRC32<Key> crc32;

    static bool val_comp(const std::pair<uint32_t,T> &a, const
std::pair<uint32_t,T> &b)
    {
        return (a.scnd < b.scnd);
    }
 public:
      class Iterator;

      // -- Member types --

      typedef uint32_t key_hash_type;

      typedef Key key_type;
      typedef T mapped_type;
      typedef std::pair<key_hash_type, T> value_type;
      typedef std::size_t size_type;
      typedef std::ptrdiff_t difference_type;
      typedef value_type& reference;
      typedef const value_type& const_reference;
      typedef value_type* pointer;
      typedef const value_type* const_pointer;
      typedef Iterator iterator;
      typedef const Iterator const_iterator;

      // -- Member functions --

      Map() {}
```

```
    Map(const Map &other) : data(other.data) {}

    ~Map() {}

  Map( iterator fst, iterator lst)
  {
      if(lst < fst)
        {
          iterator tmp = lst;
          lst = fst;
          fst = tmp;
        }
      data.resize(lst-fst);
      int i = 0;
      for(;fst != lst; ++fst)
      {
          data[i] = *fst;
          std::cout << data[i].scnd << " ";
      }

      std::cout << std::endl;
  }

  Map &operator=(const Map &other) {
    if (this == &other)
        return *this;
    data = other.data;
    return *this;
  }

  // -- Iterators

  iterator begin() {  return iterator(data.data()); }
  iterator end() { return iterator(data.data() + data.size()); }
  const_iterator       cbegin()       const       {              return
const_iterator(const_cast<pointer>(data.data())); }
  const_iterator       cend()       const       {              return
const_iterator(const_cast<pointer>(data.data() + data.size())); }

  // -- Element accesshashed_key

  T &at(const Key &key) {
    key_hash_type hashed_key = crc32(key);
    iterator it;

    for (it = begin(); it != end(); ++it) {
      if (it->fst == hashed_key)
        return it->scnd;
    }
    throw std::out_of_range("No element with this key\n");
  }
```

```
const T &at(const Key &key) const {
  key_hash_type hashed_key = crc32(key);
  iterator it;

  for (it = begin(); it != end(); ++it) {
    if (it->fst == hashed_key)
      return it->scnd;
  }
  throw std::out_of_range("No element with this key\n");
}

T &operator[](const Key &key) {
  key_hash_type hashed_key = crc32(key);
  iterator it;

  for (it = this->begin(); it != this->end(); ++it)
    if (hashed_key == it->fst)
      break;

  if (it == this->end()) {
    this->insert(std::pair<Key, T>{key, T()});

    for (it = this->begin(); it != this->end(); ++it)
      if (hashed_key == it->fst)
        break;
  }

  if (it == NULL)
    return this->data.back().scnd;

  return it->scnd;
}

// -- Capacity

bool empty() const { return data.empty(); }
size_type size() const { return data.size(); }
size_type max_size() const { return data.max_size(); }

// -- Modifiers

void clear() { data.clear(); }

std::pair<iterator, bool> insert(const std::pair<Key, T> &value) {
    key_hash_type hashed_key = crc32(value.fst);
    iterator it;

    for (it = this->begin(); it != this->end(); ++it) {
      if (it->fst == hashed_key)
        return std::pair<iterator, bool>(it, false);
    }
```

```
      data.push_back(std::pair<key_hash_type,            T>{hashed_key,
value.scnd});

      return std::pair<iterator, bool>(this->find(value.fst), true);
  }

  iterator insert(iterator pos, const std::pair<Key, T> &value) {
      key_hash_type hashed_key = crc32(value.fst);
      *pos = (std::pair<key_hash_type, T>{hashed_key, value.scnd});

      return this->find(value.fst);
  }

  iterator erase(iterator pos) {
      for (typename Vector< std::pair<key_hash_type, T> >::iterator it
= data.begin(); it != data.end(); ++it) {
        if (it->fst == pos->fst)
        {
          data.erase(it);
          return pos;
        }
      }
      return end();
  }

  iterator erase(iterator strt, iterator fnsh) {
      for (typename Vector< std::pair<key_hash_type, T> >::iterator it
= data.begin(); it != data.end() && strt < fnsh;) {
        if (it->fst == strt->fst)
        {
          data.erase(it);
          strt++;
        }
        else
          ++it;
      }
    if(strt != fnsh) return end();
    return fnsh;
  }

  size_type erase(const Key &key) {
    iterator it = find(key);
    if (it == end())
      return 0;
    this->erase(it);
    return 1;
  }

  void swap(Map &other) {
    Map<Key, T> tmp = *this;
    *this = other;
    other = tmp;
```

```
  }

  // -- Lookup

  size_type count(const Key &key) const {
    return !(this->find(key) == this->end());
  }

  iterator find(const Key &key) {
    key_hash_type hashed_key = crc32(key);
    iterator it;

    for (it = begin(); it != end(); ++it) {
      if (hashed_key == it->fst)
        break;
    }
    return it;
  }

  const_iterator find(const Key &key) const {
    key_hash_type hashed_key = crc32(key);
    iterator it;

    for (it = begin(); it != end(); ++it) {
      if (hashed_key == it->fst)
        break;
    }
    return it;
  }

  std::pair<const_iterator, const_iterator> equal_range(const Key &key)
const {
    std::sort(data.begin(), data.end(), val_comp);
    iterator lower_it, upper_it;
    iterator it = begin();
    T elem = this->find(key)->scnd;

    while (it->scnd < elem && it != end())
      ++it;
    lower_it = it;
    while (it->scnd <= elem && it != end())
      ++it;
    upper_it = it;
    return     std::pair<const_iterator,     const_iterator>(lower_it,
upper_it);
  }

  iterator lower_bound(const Key &key) {
    std::sort(data.begin(), data.end(), val_comp);
    iterator it = begin();
    T elem = this->find(key)->scnd;
```

```
    while (it->scnd < elem && it != end())
      ++it;
    return it;
  }

  const_iterator lower_bound(const Key &key) const {
    std::sort(data.begin(), data.end(), val_comp);
    iterator it = begin();
    T elem = this->find(key)->scnd;

    while (it->scnd < elem && it != end())
      ++it;
    return it;
  }

  iterator upper_bound(const Key &key) {
    std::sort(data.begin(), data.end(), val_comp);
    iterator it = begin();
    T elem = this->find(key)->scnd;

    while (it->scnd <= elem && it != end())
      ++it;
    return it;
  }

  const_iterator upper_bound(const Key &key) const {
    std::sort(data.begin(), data.end(), val_comp);
    iterator it = begin();
    T elem = this->find(key)->scnd;

    while (it->scnd <= elem && it != end())
      ++it;
    return it;
  }

  friend void swap(Map &lhs, Map &rhs) {
    Map tmp = lhs;
    lhs = rhs;
    rhs = tmp;
  }

  // -- Non-member functions

  friend bool operator==(const Map<Key, T> &lhs, const Map<Key, T> &rhs)
{
    iterator l = lhs.begin();
    iterator r = rhs.begin();

    if (lhs.size() != rhs.size())
      return false;

    for (; l != lhs.end(); ++l, ++r) {
```

```
      if (l->scnd != r->scnd)
        return false;
    }
    return true;
  }


  friend bool operator>(const Map<Key, T> &lhs, const Map<Key, T> &rhs)
{
    iterator l = lhs.begin();
    iterator r = rhs.begin();

    if(lhs.size() != rhs.size())
        return lhs.size() > rhs.size();

    for (; l != lhs.end(); ++l, ++r) {
      if (l->scnd != r->scnd)
        return l->scnd > r->scnd;
    }
    return false;
  }

  friend bool operator!=(const Map<Key, T> &lhs, const Map<Key, T> &rhs)
{ return !(lhs == rhs); }
  friend bool operator<(const Map<Key, T> &lhs, const Map<Key, T> &rhs)
{ return !(lhs > rhs || lhs == rhs); }
  friend bool operator<=(const Map<Key, T> &lhs, const Map<Key, T> &rhs)
{ return !(lhs > rhs); }
  friend bool operator>=(const Map<Key, T> &lhs, const Map<Key, T> &rhs)
{ return !(lhs < rhs); }
};

template <class Key, class T>
class Map<Key, T>::Iterator {
private:
  std::pair<key_hash_type, T>* ptr;

public:
  // -- Member types --

    typedef ptrdiff_t difference_type;
    typedef typename Map<Key, T>::value_type value_type;
    typedef typename Map<Key, T>::pointer        pointer;
    typedef typename Map<Key, T>::reference      reference;
    typedef std::random_access_iterator_tag iterator_category;

    // -- Member functions --

    Iterator() : ptr(NULL) {}
    Iterator(const_iterator &other) : ptr(other.ptr) {}
    Iterator(pointer cont) : ptr(cont) {}
    ~Iterator() {}
```

```
    Iterator &operator=(const_iterator &other) { ptr = other.ptr; return
*this; }
    Iterator &operator=(pointer rhs) {ptr = rhs; return *this;}


    reference operator*() const {return *ptr;}
    pointer operator->() const {return ptr;}
    reference operator[](difference_type rhs) const {return ptr[rhs];}

    Iterator &operator++() { ++ptr; return *this; }
    Iterator operator++(int) { Iterator old = ptr; ++ptr; return old; }
    Iterator &operator--() { --ptr; return *this; }
    Iterator operator--(int) { Iterator tmp = ptr; --ptr; return tmp; }

    Iterator&  operator+=(difference_type  rhs)  {ptr  +=  rhs;  return
*this;}
    Iterator&  operator-=(difference_type  rhs)  {ptr  -=  rhs;  return
*this;}

    difference_type operator-(const_iterator& rhs) const {return ptr-
rhs.ptr;}
    Iterator    operator+(difference_type    rhs)    const    {return
Iterator(ptr+rhs);}
    Iterator operator-(difference_type rhs) const {return Iterator(ptr-
rhs);}
    friend Iterator operator+(difference_type lhs, const Iterator& rhs)
{return Iterator(lhs+rhs.ptr);}
    friend Iterator operator-(difference_type lhs, const Iterator& rhs)
{return Iterator(lhs-rhs.ptr);}

    bool operator==(const Iterator& rhs) const {return ptr == rhs.ptr;}
    bool operator!=(const Iterator& rhs) const {return ptr != rhs.ptr;}
    bool operator>(const Iterator& rhs) const {return ptr > rhs.ptr;}
    bool operator<(const Iterator& rhs) const {return ptr < rhs.ptr;}
    bool operator>=(const Iterator& rhs) const {return ptr >= rhs.ptr;}
    bool operator<=(const Iterator& rhs) const {return ptr <= rhs.ptr;}
};
```