My Project

Generated by Doxygen 1.10.0

Chapter 1

Specs

```
CPU: AMD Ryzen 5 5600U with Radeon Graphics 2.30 GHz
RAM: 16.0 GB (15.3 GB usable)
SSD: WDC PC SN530 512Gb
GPU: Integrated with CPU
OS: Windows 10 Pro 64-bit
```

1.1 Introduction

This project provides an implementation of a dynamic array container in C++ named `vector`, similar to `std::vector` from the C++ Standard Library. This custom `vector` class template provides various functionalities to handle dynamic arrays with an emphasis on performance, memory management, and ease of use. The class supports common operations such as insertion, deletion, resizing, and element access while handling memory allocation and deallocation internally.

1.1.1 Features

1.1.1.1 Constructors

1. Default Constructor

   `vector()`: Initializes an empty vector.

2. Fill Constructor

`explicit vector(size_type n, const T& t = T{})`: Initializes a vector with n elements, each initialized to t.

1. Copy Constructor

`vector(const vector& v)`: Initializes a vector as a copy of another vector v.

1. Range Constructor

`template <class InputIterator> vector(InputIterator first, InputIterator last)`↩ : Initializes a vector with elements from the range `[first, last)`.

1. Move Constructor

`vector(vector&& v)`: Initializes a vector by moving resources from another vector `v`.

1. Initializer List Constructor

`vector(const std::initializer_list<T> il)`: Initializes a vector with elements from an initializer list il.

1.1.1.2 Destructor

`~vector()`: Destroys the vector and deallocates its memory.

1.1.1.3 Assignment Operators

1. Copy Assignment

`vector& operator=(const vector& other)`: Assigns the contents of `other` to the vector.

1. Move Assignment

`vector& operator=(vector&& other) noexcept`: Moves the contents of other to the vector.

1.1.1.4 Iterators

1. `iterator begin()`
2. `const_iterator begin() const`
3. `iterator end()`
4. `const_iterator end() const`

1.1.1.5 Capacity

1. `size_type size() const`: Returns the number of elements in the vector.
2. `size_type max_size() const`: Returns the maximum possible number of elements.
3. `void resize(size_type sz)`: Resizes the vector to contain sz elements.

4.`void resize(size_type sz, const value_type& value)`: Resizes the vector to contain `sz` elements, each initialized to value.

1. `size_type capacity() const`: Returns the number of elements that can be held in currently allocated storage.
2. `bool empty() const noexcept`: Checks if the vector is empty.
3. `void reserve(size_type n)`: Reserves storage for at least `n` elements.
4. `void shrink_to_fit()`: Reduces capacity to fit the size.

1.1.1.6 Element Access

- `T& operator[](size_type n)`: Accesses the element at position n.
- `const T& operator[](size_type n) const`
- `reference at(size_type n)`: Accesses the element at position n with bounds checking.
- `const_reference at(size_type n) const`
- `reference front()`: Accesses the first element.
- `const_reference front() const`
- `reference back()`: Accesses the last element.
- `const_reference back() const`
- `value_type* data() noexcept`: Returns a pointer to the underlying array.
- `const value_type* data() const noexcept`

1.1.1.7 Modifiers

- `template <class InputIterator> void assign(InputIterator first, Input↩ Iterator last)`: Assigns values from the range [first, last).
- `void assign(size_type n, const value_type& val)`: Assigns n copies of val to the vector.
- `void assign(std::initializer_list<value_type> il)`: Assigns values from the initializer list il.
- `void push_back(const value_type& t)`: Adds an element to the end.
- `void push_back(value_type&& val)`: Adds an element to the end (move).
- `void pop_back()`: Removes the last element.
- `iterator insert(iterator pos, const T& value)`: Inserts an element at the specified position.
- `iterator erase(iterator position)`: Erases the element at the specified position.
- `iterator erase(iterator first, iterator last)`: Erases elements in the range [first, last).
- `void swap(vector& x)`: Swaps the contents of this vector with x.
- `void clear() noexcept`: Clears the contents of the vector.

1.1.1.8 Relational Operators

- `bool operator==(const vector<T>& other) const`: Checks if two vectors are equal.
- `bool operator!=(const vector<T>& other) const`
- `bool operator<(const vector<T>& other) const`
- `bool operator<=(const vector<T>& other) const`
- `bool operator>(const vector<T>& other) const`
- `bool operator>=(const vector<T>& other) const`

1.1.1.9 Private Member Functions

- `void create()`: Initializes an empty vector.
- `void create(size_type n, const T& val)`: Allocates and initializes storage for n elements.
- `void create(const_iterator i, const_iterator j)`: Allocates and initializes storage from the range [i, j).
- `void uncreate()`: Destroys elements and deallocates storage.
- `void grow(size_type new_capacity = 1)`: Grows the vector to accommodate more elements.
- `void unchecked_append(const T& val)`: Appends an element without checking capacity.

1.1.1.10 Testing `std::vector` and `vector` speed and reallocations

Size	std::vector Time	Custom vector Time	std::vector Reallocations	vector Reallocations
10000	0 s	0 s	14	14
100000	0.001994 s	0.001995 s	17	17
1000000	0.020944 s	0.016955 s	20	20
10000000	0.206448 s	0.183509 s	24	24
10000000	2.0415s	1.77326s	27	27

1.1.1.11 Testing `std::vector` and `vector` file generating

Size	std::vector Time	Custom vector Time
1000	0.013903s	0.010972s
10000	0.100278s	0.105718s
100000	0.964256s	1.06715s
1000000	9.63277s	10.2358s
1000000	117.533s	106.347s

1.1.1.12 Testing `std::vector` and `vector` file read/sort/divide time

`std::vector`	1000 students	10000 students	100000 students	1000000 students	10000000 students
Skaitymas uztruko:	0.015444s	0.166344s	1.59188s	15.4619s	165.159s
Rusiavimas uztruko:	0.006081s	0.0552494s	0.538313s	6.18915s	60.0473s
Studentu skirstymas uztruko:	0.002001s	0.00835907s	0.0876151s	1.10855s	11.3426s

vector	1000 students	10000 students	100000 students	1000000 students	10000000 students
Skaitymas uztruko:	0.008976000s	0.085770000s	0.831775000s	8.325740000s	84.65230s
Rusiavimas uztruko:	0.003989000s	0.092781000s	0.4375302s	5.6458932s	58.65063s

vector	1000 students	10000 students	100000 students	1000000 students	10000000 students
Studentu skirstymas uztruko:	0.000998000s	0.010970000s	0.099732000s	1.096098000s	11.39806s

1.1.1.13 Installing Inno Setup Compiler

1. Open your web browser and go to the official Inno Setup website: Inno Setup.
2. Locate the downloaded setup file (e.g., `is-X.X.X.exe`).
3. Double-click the setup file to start the installation process.
4. Follow the on-screen instructions to complete the installation. You can use the default settings.

1.1.1.14 Creating a Simple Installer Script

1. Open the Inno Setup Compiler from the Start Menu or desktop shortcut.
2. Click on "File" and then "New" to create a new script. The "New Script Wizard" will open.
3. Follow the steps in the wizard:

- Application Information: Enter your application's name, version, and publisher information.
- Application Folder: Specify the default installation folder (e.g., {pf}\MyApp for Program Files).
- Application Files: Add the files you want to include in your installer (e.g., executable files, DLLs, etc.).
- Application Icons: Specify any shortcuts to create (e.g., desktop or Start Menu shortcuts).
- Setup Languages: Select the languages you want to support in your installer.

1. Review your settings and click "Finish." The wizard will generate a basic script.

1.1.1.15 Compiling and Running the Script

1. After the script is generated, review it in the Inno Setup Compiler. Make any necessary changes or customizations.
2. Save your script by clicking "File" and then "Save As." Give it a meaningful name (e.g., setup.iss).
3. To compile the script, click "Build" and then "Compile" (or press F9). The compiler will create an installer executable based on your script.
4. Once the compilation is complete, you will see a message indicating that the setup has been compiled successfully.
5. Locate the compiled installer executable (e.g., setup.exe) in the output directory specified in your script.
6. Run the installer to test it and ensure everything works as expected.

1.2 Abstract Class "Zmogus"

An abstract class in C++ is a class that contains at least one pure virtual function. A pure virtual function is a virtual function for which we provide only the declaration in the base class, without providing any implementation. Abstract classes are designed to be used as base classes, and they cannot be instantiated directly. Instead, they are intended to serve as interfaces that define a common set of methods that derived classes must implement.

```
class Zmogus {
public:
    virtual void setVardas(std::string vardas) = 0;
    virtual std::string getVardas() const = 0;
    virtual void setPavarde(std::string pavarde) = 0;
    virtual std::string getPavarde() const = 0;
    virtual ~Zmogus() = default;

};
```

1.2.0.1 The key characteristics of an abstract class are:

- Contains Pure Virtual Functions: An abstract class contains at least one pure virtual function, which is declared with the virtual keyword and assigned the value 0 as its implementation.
- Cannot be Instantiated: Since abstract classes have at least one pure virtual function without an implementation, objects of abstract classes cannot be created directly. Attempting to create an instance of an abstract class will result in a compilation error.
- Used as Base Classes: Abstract classes are meant to be used as base classes. Derived classes inherit from abstract classes and provide concrete implementations for all the pure virtual functions defined in the abstract base class.

1.3 Rule of Five and Overloaded Methods

1.3.1 Rule of Five

In C++, the Rule of Five refers to a set of guidelines concerning resource management for classes that manage dynamic memory allocation or external resources. The Rule of Five consists of five special member functions that need to be defined or explicitly disabled if one of them is used:

1.3.1.1 Destructor

Responsible for releasing resources acquired by the object.

```
Studentas::~Studentas() {
nd_rezultatai.clear();
vardas.clear();
pavarde.clear();
egzaminas = 0;
```

}

1.3.1.2 Copy Constructor

Creates a new object as a copy of an existing object.

```
Studentas::Studentas(const Studentas &copy)
: vardas(copy.vardas), pavarde(copy.pavarde), nd_rezultatai(copy.nd_rezultatai),egzaminas(copy.egzaminas) {}
```

1.3.1.3 Copy Assignment Operator

Assigns the state of one object to another existing object.

```
Studentas& Studentas::operator=(const Studentas& copy)
{
    if(this !=&copy)
    {
        vardas = copy.vardas;
        pavarde = copy.pavarde;
        nd_rezultatai = copy.nd_rezultatai;
        egzaminas = copy.egzaminas;
    }
    return *this;
}
```

1.3.1.4 Move Constructor

Transfers resources from a temporary object to a new object.

```
    Studentas& Studentas::operator=(Studentas&& copy) noexcept {
        if (this!= &copy) {
            // Swap the members of the current object with the members of the other object
            std::swap(vardas, copy.vardas);
            std::swap(pavarde, copy.pavarde);
            std::swap(nd_rezultatai, copy.nd_rezultatai);
            std::swap(egzaminas, copy.egzaminas);
        }
        return *this;
    }
```

1.3.1.5 Move Assignment Operator

Transfers resources from one object to another existing object.

```
    Studentas& Studentas::operator=(Studentas&& copy) noexcept {
        if (this!= &copy) {
            // Swap the members of the current object with the members of the other object
            std::swap(vardas, copy.vardas);
            std::swap(pavarde, copy.pavarde);
            std::swap(nd_rezultatai, copy.nd_rezultatai);
            std::swap(egzaminas, copy.egzaminas);
        }
        return *this;
    }
```

1.3.2 Overloaded Methods

The `Studentas` class overloads the input and output operators (`operator<<` and `operator>>`) to provide serialization and deserialization capabilities. These overloaded methods allow objects of the `Studentas` class to be written to an output stream (e.g., `std::cout` or a file) and read from an input stream (e.g., `std::cin` or a file).

1.3.2.1 Output Operator (`operator<<`)

The output operator `operator<<` is overloaded to serialize a `Studentas` object to an output stream. It prints the `vardas`, `pavarde`, `egzaminas`, and `nd_rezultatai` member variables to the output stream.

```
std::ostream& operator<<(std::ostream& output, const Studentas &student) {
    output << student.vardas << " " << student.pavarde << " " << student.egzaminas << " ";
    for (int pazymys : student.nd_rezultatai) {
        output << std::to_string(pazymys) << " "; // Pries printinant pakeist int'a i string'a
    }
    return output;
}
```

1.3.2.2 Input Operator (`operator>>`)

The input operator `operator>>` is overloaded to deserialize a `Studentas` object from an input stream. It reads `vardas`, `pavarde`, `egzaminas`, and `nd_rezultatai` from the input stream and constructs a `Studentas` object accordingly.

```
std::istream& operator>>(std::istream& input, Studentas &student) {
    input >> student.vardas >> student.pavarde;
    input >> student.egzaminas;
    student.nd_rezultatai.clear();
    int pazymys;
    while (input >> pazymys) {
        student.nd_rezultatai.push_back(pazymys);
    }
    return input;
}
```

1.4 Running a Makefile for C/C++ Projects

This guide will walk you through the process of running a Makefile for compiling and executing C/C++ programs on both macOS and Windows. If this tutorial does not work for you, try these solutions `Makefile`.

1.4.1 Prerequisites

1. **Make Installation:**
    - macOS: Make is usually pre-installed. You can verify by opening a terminal and typing `make -v`.
    - Windows: Install Make using a package manager like `Chocolatey`. Run `choco install make` in PowerShell or Command Prompt.

- **C/C++ Compiler:**
    - Ensure you have a C/C++ compiler installed. On macOS, Clang is typically pre-installed. On Windows, you can use MinGW or Cygwin.
- **Text Editor or IDE:**
    - Use a text editor or IDE to write your C/C++ code and Makefile. Popular choices include Visual Studio Code, Sublime Text, Atom, etc.

1.4.1.1 1. Write Your Code

- Create your C/C++ code in one or more `.cpp` or `.c` files.

1.4.1.2 2. Write Makefile

- Create a file named `Makefile` (without extension) in the same directory as your source code.
- Open `Makefile` in a text editor and define build rules.

1.4.1.3 3. Open Terminal/Command Prompt

- macOS: Open Terminal.
- Windows: Open Command Prompt or PowerShell.

1.4.1.4 4. Navigate to Project Directory

- Use `cd` command to navigate to the directory containing your code and Makefile.

1.4.1.5 5. Run Make

- Type `make` and press Enter. This executes the default target (`all`) in the Makefile.

1.4.1.6 6. Run Your Program

- After successful build, an executable file (e.g., `run` on macOS or `run.exe` on Windows) will be generated in the same directory.
- Run the program by typing `./run` on macOS or `run.exe` on Windows, and press Enter.

Congratulations! You've successfully compiled and executed your C/C++ program using a Makefile. If you encounter any errors during compilation, check your Makefile and source code for issues.

1.4.1.7 6. How To Run Code With Flags

- Type `make optimize`
- Type `./run_o1 ./run_o2 ./run_o3`

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

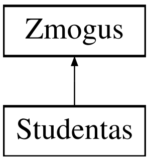
Here is a list of all documented files with brief descriptions:

Chapter 5

Class Documentation

5.1 Studentas Class Reference

Inheritance diagram for Studentas:



Public Member Functions

- **Studentas** (const std::string &vardas, const std::string &pavarde)
- **Studentas** (const Studentas &copy)
- Studentas & **operator=** (const Studentas &copy)
- **Studentas** (Studentas &&copy) noexcept
- Studentas & **operator=** (Studentas &&copy) noexcept
- void setVardas (std::string vardas)
- std::string getVardas () const
- void setPavarde (std::string pavarde)
- std::string getPavarde () const
- void **setNamuDarbai** (const vector< int > &nd)
- vector< int > **getNamuDarbai** () const
- void **addNamuDarbai** (int pazymys)
- void **setEgzaminas** (int egzaminas)
- int **getEgzaminas** () const
- double **calcVidurkis** () const
- double **calcMediana** () const
- double **calcGalutinis** (bool useVidurkis) const
- void **randomND** ()
- void **randomStudentai** ()

Friends

- std::ostream & **operator<<** (std::ostream &output, const Studentas &student)
- std::istream & **operator>>** (std::istream &input, Studentas &student)

5.1.1 Member Function Documentation

5.1.1.1 getPavarde()

```
std::string Studentas::getPavarde ( ) const  [virtual]
```

Implements Zmogus.

5.1.1.2 getVardas()

```
std::string Studentas::getVardas ( ) const  [virtual]
```

Implements Zmogus.

5.1.1.3 setPavarde()

```
void Studentas::setPavarde (
             std::string pavarde )  [virtual]
```

Implements Zmogus.

5.1.1.4 setVardas()

```
void Studentas::setVardas (
             std::string vardas )  [virtual]
```

Implements Zmogus.

The documentation for this class was generated from the following files:

- studentas.h
- studentas.cpp

5.2 vector< T > Class Template Reference

Public Types

- typedef size_t **size_type**
- typedef T **value_type**
- typedef T & **reference**
- typedef const T & **const_reference**
- typedef T ∗ **iterator**
- typedef const T ∗ **const_iterator**

Public Member Functions

- **vector** (size_type n, const T &t=T{})
- **vector** (const vector &v)
- template<class InputIterator >
  vector (InputIterator first, InputIterator last)
- **vector** (vector &&v)
- **vector** (const std::initializer_list< T > il)
- vector & **operator=** (const vector &other)
- vector & **operator=** (vector &&other) noexcept
- iterator **begin** ()
- const_iterator **begin** () const
- iterator **end** ()
- const_iterator **end** () const
- size_type **size** () const
- size_type **max_size** () const
- void **resize** (size_type sz)
- void **resize** (size_type sz, const value_type &value)
- size_type **capacity** () const
- bool **empty** () const noexcept
- void **reserve** (size_type n)
- void **shrink_to_fit** ()
- T & **operator[ ]** (size_type n)
- const T & **operator[ ]** (size_type n) const
- reference **at** (size_type n)

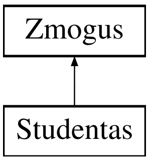
- const_reference **at** (size_type n) const
- reference **front** ()
- const_reference **front** () const
- reference **back** ()
- const_reference **back** () const
- value_type ∗ **data** () noexcept
- const value_type ∗ **data** () const noexcept
- template<class InputIterator >
  void **assign** (InputIterator first, InputIterator last)
- void **assign** (size_type n, const value_type &val)
- void **assign** (std::initializer_list< value_type > il)
- void **push_back** (const value_type &t)
- void **push_back** (value_type &&val)
- void **pop_back** ()
- iterator **insert** (iterator pos, const T &value)
- iterator **erase** (iterator position)
- iterator **erase** (iterator first, iterator last)
- void **swap** (vector &x)
- void **clear** () noexcept
- bool **operator==** (const vector< T > &other) const
- bool **operator!=** (const vector< T > &other) const
- bool **operator<** (const vector< T > &other) const
- bool **operator<=** (const vector< T > &other) const
- bool **operator>** (const vector< T > &other) const
- bool **operator>=** (const vector< T > &other) const
- void **swap** (vector< T > &x, vector< T > &y)

The documentation for this class was generated from the following file:

- vector.h

5.3 Zmogus Class Reference

Inheritance diagram for Zmogus:



Public Member Functions

- virtual void **setVardas** (std::string vardas)=0
- virtual std::string **getVardas** () const =0
- virtual void **setPavarde** (std::string pavarde)=0
- virtual std::string **getPavarde** () const =0

The documentation for this class was generated from the following file:

- studentas.h

Chapter 6

File Documentation

6.1 containers.h

```
#ifndef CONTAINERS_H
#define CONTAINERS_H

#include "vector.h"

enum class ContainerType { None, Vector};
enum class Action { None, Generate, Sort };
ContainerType getContainerChoice();
Action getActionChoice();

void performAction(ContainerType containerChoice, Action actionChoice, const vector<int>& sizes);
void runApp();

#endif
```

6.2 funkcijos.h

```
#ifndef FUNKCIJOS_H
#define FUNKCIJOS_H

#include "studentas.h"
#include "vector.h"
#include <string>

void spausdintiGalutiniusBalus(const vector<Studentas>& studentai, const std::string& isvedimoFailoVardas = "", int rusiavimoTipas = 1);
void manualInput(vector<Studentas>& studentai);
void generateGradesOnly(vector<Studentas>& studentai);
void readFileDataFromFile(vector<Studentas>& studentai, const std::string& failoVardas);
void generateStudentFiles(const vector<int>& sizes);
void rusiuotiStudentus(const vector<int>& sizes);

#endif
```

6.3 funkcijosVector.h

```
#ifndef FUNKCIJOSVECTOR_H
#define FUNKCIJOSVECTOR_H

#include "studentas.h"
#include "vector.h"

void readDataVector(vector<Studentas>& studentai, const std::string& failoVardas);
void generateStudentFilesVector(int size);
void rusiuotStudentusVector(const std::string& failoVardas);
void rusiuotStudentusVector2(const std::string& failoVardas);
void rusiuotStudentusVector3(const std::string &failoVardas);

#endif // FUNKCIJOSVECTOR_H
```

6.4 studentas.h

```
#ifndef STUDENTAS_H
#define STUDENTAS_H
```

```

#include <string>
#include <vector>
#include "vector.h"

class Zmogus {
public:
    virtual void setVardas(std::string vardas) = 0;
    virtual std::string getVardas() const = 0;
    virtual void setPavarde(std::string pavarde) = 0;
    virtual std::string getPavarde() const = 0;
    virtual ~Zmogus() = default;

};

class Studentas : public Zmogus {
private:
    std::string vardas;
    std::string pavarde;
    vector<int> nd_rezultatai;
    int egzaminas;
public:
    // Constructor
    Studentas();
    // Constructor with parameters
    Studentas(const std::string &vardas, const std::string &pavarde);
    //Destructor
    ~Studentas();
    // Copying constructor
    Studentas(const Studentas &copy);

    // Copy assignment
    Studentas& operator=(const Studentas& copy);

    // Move constructor
    Studentas(Studentas&& copy) noexcept;

    // Move assignment operator
    Studentas& operator=(Studentas&& copy) noexcept;

    void setVardas(std::string vardas);
    std::string getVardas() const;

    void setPavarde(std::string pavarde);
    std::string getPavarde() const;

    void setNamuDarbai(const vector<int> &nd);
    vector<int> getNamuDarbai() const;

    void addNamuDarbai(int pazymys);

    void setEgzaminas(int egzaminas);
    int getEgzaminas() const;

    double calcVidurkis() const;
    double calcMediana() const;
    double calcGalutinis(bool useVidurkis) const;
    void randomND();
    void randomStudentai();

    friend std::ostream &operator«(std::ostream &output, const Studentas &student);
    friend std::istream &operator»(std::istream &input, Studentas &student);
};

#endif // STUDENTAS_H
```

6.5 testRules.h

```
#ifndef TESTRULES_H
#define TESTRULES_H

#include "studentas.h"
#include <iostream>
#include <fstream>

void testRuleOfFive();
void testSerializationDeserialization();

#endif // TESTRULES_H
```

6.6 vector.h

```
#ifndef VECTOR_H
#define VECTOR_H


#include <iostream>
#include <memory>
#include <algorithm>
#include <limits>

template <typename T>
class vector{
    public:
        typedef size_t size_type;
        typedef T value_type;
        typedef T& reference;
        typedef const T& const_reference;
        typedef T* iterator;
        typedef const T* const_iterator;

    //KONSTRUKTORIAI
        //default
        vector() {create();}
        //fill
        explicit vector(size_type n, const T& t = T{}) { create (n,t); }
        //copy constructor
        vector(const vector& v) { create(v.begin(), v.end()); }
        //range constructor
        template <class InputIterator>
        vector (InputIterator first, InputIterator last) { create(first,last); }
        //move constructor
        vector (vector&& v) {
            create();
            swap(v);
            v.uncreate();
        }
        //initializer list constructor
        vector(const std::initializer_list<T> il) { create(il.begin(), il.end()); }

    //DESTRUKTORIUS
        ~vector() {uncreate();}

    //OPERATOR =
        //copy assignment
        vector& operator = (const vector& other) {
            if (this != &other) {
                uncreate();
                create(other.begin(), other.end());
            }
            return *this;
        };

        //move assignment
        vector& operator = (vector&& other) noexcept {
            if (this != &other) {
                // Free the current resources
                uncreate();
                // Swap pointers with the source vector
                std::swap(dat, other.dat);
                std::swap(avail, other.avail);
                std::swap(limit, other.limit);
            }
            return *this;
        }

    //ITERATORIAI
        iterator begin() {return dat;}
        const_iterator begin() const {return dat;}
        iterator end() {return avail;}
        const_iterator end() const {return avail;}

    //CAPACITY
        size_type size() const {return avail-dat;}
        size_type max_size() const {return std::numeric_limits<size_type>::max();}
        void resize(size_type sz) {
            if (sz < size()) {
                iterator it = dat + sz;
                while (it != avail) {
                    alloc.destroy(it++);
                }
                avail = dat + sz;
            }
            else if (sz > capacity()) {
                grow(sz);
                std::uninitialized_fill(avail, dat + sz, value_type());
                avail = dat + sz;
            }
            else if (sz > size()) {
                std::uninitialized_fill(avail, dat + sz, value_type());
                avail = dat + sz;
            }
        }
        void resize(size_type sz, const value_type& value) {
            if (sz > capacity()) {
                grow(sz);
            }

            if (sz > size()) {
                insert(end(), sz - size(), value);
            } else if (sz < size()) {
                avail = dat + sz;
            }
        }

        size_type capacity() const {return limit-dat;}
        bool empty() const noexcept { return size() == 0;}
        void reserve (size_type n) {
            if (n > capacity()) {
                grow(n);
            }
        }
        void shrink_to_fit(){
            if (limit > avail)
            limit = avail;
        }

    //ELEMENT ACCESS
        T& operator[] (size_type n) {return dat[n];}
        const T& operator[] (size_type n) const {return dat[n];}
        reference at (size_type n) {
            if (n >= size() || n < 0)
                throw std::out_of_range("Index out of range");
            return dat[n];
        }
        const_reference at (size_type n) const {
            if (n >= size() || n < 0)
                throw std::out_of_range("Index out of range");
            return dat[n];
        }
        reference front() {
            return dat[0];
        };
        const_reference front() const {
            return dat[0];
        }
        reference back() {
            return dat[size() - 1];
        }
        const_reference back() const {
            return dat[size() - 1];
        }
        value_type* data() noexcept {
            return dat;
        }
        const value_type* data() const noexcept {
            return dat;
        }

    //MODIFIERS
        template <class InputIterator>
        void assign (InputIterator first, InputIterator last) {
            uncreate();
            create(first, last);
        }
        void assign (size_type n, const value_type& val) {
            uncreate();
            create(n, val);
        }
        void assign (std::initializer_list<value_type> il) {
            uncreate();
            create(il);
        }
        void push_back (const value_type& t) {
            if (avail==limit)
                grow();
            unchecked_append(t);
        }
        void push_back (value_type&& val) {
            if (avail == limit)
                grow();
            unchecked_append(val);
        }
        void pop_back() {
            if (avail != dat)
                alloc.destroy(--avail);
        }
        iterator insert(iterator pos, const T &value) {
            size_type index = pos - begin();
            size_type numNewElements = 1; // Since we're inserting a single element

            // Check if resizing is necessary
            if (size() + numNewElements > capacity()) {
                reserve((size() + numNewElements) * 2);
            }

            // Move elements to make space for the new one
            std::move_backward(dat + index, avail, avail + numNewElements);
            // Insert the new element
            dat[index] = value;
            // Update the size
            avail += numNewElements;

            return dat + index; // Return iterator pointing to the inserted element
}

        iterator erase(iterator position) {
            if (position < dat || position > avail) {
                throw std::out_of_range("Index out of range");
            }
            std::move(position + 1, avail, position);
            alloc.destroy(avail - 1);
            --avail;

            return position;
        }
        iterator erase(iterator first, iterator last) {
            iterator new_available = std::uninitialized_copy(last, avail, first);

            iterator it = avail;
            while (it != new_available) {
                alloc.destroy(--it);
            }

            avail= new_available;
            return last;
        }

        void swap(vector& x) {
            std::swap(dat, x.dat);
            std::swap(avail, x.avail);
            std::swap(limit, x.limit);
        }
        void clear() noexcept {
            uncreate();
        }

    //RELATION OPERATORS
        bool operator== (const vector<T>& other) const {
            if (size() != other.size()) {
                return false;
            }

            return std::equal(begin(), end(), other.begin());
        }
        bool operator!= (const vector<T>& other) const {
            return !(*this == other);
        }
        bool operator < (const vector<T> & other) const {
            return std::lexicographical_compare(begin(), end(), other.begin(), other.end());
        }
        bool operator <= (const vector<T> & other) const {
            return !(other < *this);
        }
        bool operator > (const vector<T> & other) const {
            return std::lexicographical_compare(other.begin(), other.end(), begin(), end());
        }
        bool operator >= (const vector<T> & other) const {
            return !(other > *this);
        }

        void swap (vector<T>& x, vector<T>& y) {
            std::swap(x,y);
        }

    private:
        iterator dat;
        iterator avail;
        iterator limit;
        std::allocator<T> alloc;
        void create() {dat = avail = limit = nullptr;}
        void create (size_type n, const T& val) {
            dat = alloc.allocate(n);
            limit = avail = dat + n;
            std::uninitialized_fill(dat, limit, val);
        }
        void create(const_iterator i, const_iterator j) {
            dat = alloc.allocate(j - i);
            limit = avail = std::uninitialized_copy(i, j, dat);
        }
        void uncreate(){
            if (dat) {
                iterator it = avail;
                while (it != dat) {
                    alloc.destroy(--it);
                }
            alloc.deallocate(dat, limit - dat);
            }
            dat = limit = avail = nullptr;
        }
        void grow(size_type new_capacity = 1) {
            size_type new_size = std::max(new_capacity, 2 * capacity());
            iterator new_data = alloc.allocate(new_size);
            iterator new_avail = std::uninitialized_copy(dat, avail, new_data);
            uncreate();
            dat = new_data;
            avail = new_avail;
            limit = dat + new_size;
        }
        void unchecked_append(const T& val) {
            alloc.construct(avail++, val);
        }
};

#endif // VECTOR_H
```

6.7 VectorTest.h

```
#ifndef VECTORTEST_H
#define VECTORTEST_H

void test_default_constructor();
void test_fill_constructor();
void test_copy_constructor();
void test_move_constructor();
void test_initializer_list_constructor();
void test_assignment_operator();
void test_move_assignment_operator();
void test_element_access();
void test_modifiers();
void test_comparisons();
void test_fill_time();

#endif // VECTORTEST_H
```

Index