My Project

Generated by Doxygen 1.11.0

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

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

Chapter 4

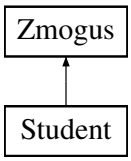
Class Documentation

4.1 Student Class Reference

Class representing a student, derived from Zmogus.

`#include <student.h>`

Inheritance diagram for Student:



Public Member Functions

- **Student** ()

  Constructor initializing the student with default values.
- ~**Student** ()

  Destructor.
- void setName (const string &vardas) override

  Sets the name of the person.
- string getName () const override

  Gets the name of the person.
- void setSurname (const string &pavarde) override

  Sets the surname of the person.
- string getSurname () const override

  Gets the surname of the person.
- void setExamResult (int egzaminas)

  Sets the exam result of the student.
- int getExamResult () const

  Gets the exam result of the student.
- void setFinalAvg (double Gal_vid)

  Sets the final average grade of the student.
- double getFinalAvg () const

Gets the final average grade of the student.

- void setFinalMedian (double Gal_med)

  Sets the final median grade of the student.
- double getFinalMedian () const

  Gets the final median grade of the student.
- void setSingleGrade (int naujasnd)

  Adds a single grade to the student's grades.
- int getSingleGrade (int i) const

  Gets a single grade of the student.
- void setGrades (const Vector< int > &ND)

  Sets the grades of the student.
- Vector< int > getGrades () const

  Gets the grades of the student.
- void **clearData** ()

  Clears all data of the student.

Public Member Functions inherited from Zmogus

- virtual ~**Zmogus** ()

  Virtual destructor.

Friends

- std::istream & **operator**>> (istream &in, Student &student)
- std::ostream & **operator**<< (ostream &out, const Student &student)

Additional Inherited Members

Protected Attributes inherited from Zmogus

- string name
- string surname

4.1.1 Detailed Description

Class representing a student, derived from Zmogus.

4.1.2 Member Function Documentation

4.1.2.1 getExamResult()

```
int Student::getExamResult () const
```

Gets the exam result of the student.

Returns

The exam result.

4.1.2.2 getFinalAvg()

```
double Student::getFinalAvg () const
```

Gets the final average grade of the student.

Returns

The final average grade.

4.1.2.3 getFinalMedian()

```
double Student::getFinalMedian () const
```

Gets the final median grade of the student.

Returns

The final median grade.

4.1.2.4 getGrades()

```
Vector< int > Student::getGrades () const
```

Gets the grades of the student.

Returns

The grades of the student.

4.1.2.5 getName()

```
string Student::getName () const  [override], [virtual]
```

Gets the name of the person.

Returns

The name of the person.

Implements Zmogus.

4.1.2.6 getSingleGrade()

```
int Student::getSingleGrade (
            int i) const
```

Gets a single grade of the student.

Parameters

i	The index of the grade to get.

Returns

The grade at index i.

4.1.2.7 getSurname()

```
string Student::getSurname () const  [override], [virtual]
```

Gets the surname of the person.

Returns

The surname of the person.

Implements Zmogus.

4.1.2.8 setExamResult()

```
void Student::setExamResult (
            int egzaminas)
```

Sets the exam result of the student.

Parameters

egzaminas	The exam result to set.

4.1.2.9 setFinalAvg()

```
void Student::setFinalAvg (
            double Gal_vid)
```

Sets the final average grade of the student.

Parameters

Gal_vid	The final average grade to set.

4.1.2.10 setFinalMedian()

```
void Student::setFinalMedian (
            double Gal_med)
```

Sets the final median grade of the student.

Parameters

Gal_med	The final median grade to set.

4.1.2.11 setGrades()

```
void Student::setGrades (
            const Vector< int > & ND)
```

Sets the grades of the student.

Parameters

ND	The grades to set.

4.1.2.12 setName()

```
void Student::setName (
            const string & vardas)  [override], [virtual]
```

Sets the name of the person.

Parameters

vardas	The name to set.

Implements Zmogus.

4.1.2.13 setSingleGrade()

```
void Student::setSingleGrade (
            int naujasnd)
```

Adds a single grade to the student's grades.

Parameters

naujasnd	The grade to add.

4.1.2.14 setSurname()

```
void Student::setSurname (
            const string & pavarde)  [override], [virtual]
```

Sets the surname of the person.

Parameters

pavarde	The surname to set.

Implements Zmogus.

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

- student.h

4.2 Vector< T > Class Template Reference

A templated dynamic array class.

`#include <vektorius.h>`

Public Types

- typedef size_t **size_type**
  Size type for the vector.
- typedef T **value_type**
  Value type of the elements.
- typedef T & **reference**
  Reference type to an element.
- typedef const T & **const_reference**
  Constant reference type to an element.
- typedef T * **iterator**
  Iterator type.
- typedef const T * **const_iterator**
  Constant iterator type.

Public Member Functions

- ~Vector ()
  Destructor.
- const_reference at (size_type n) const
  Accesses the element at position `n` with bounds checking.
- reference operator[ ] (size_type n)
  Accesses the element at position `n`.
- const_reference operator[ ] (size_type n) const
  Accesses the element at position `n`.
- reference at (size_type n)
  Accesses the element at position `n` with bounds checking.
- reference front ()
  Accesses the first element.
- const_reference front () const
  Accesses the first element.
- reference back ()

Accesses the last element.

- const_reference back () const

  Accesses the last element.
- value_type * data () noexcept

  Returns a pointer to the underlying data.
- const value_type * data () const noexcept

  Returns a constant pointer to the underlying data.
- iterator begin ()

  Returns an iterator to the beginning.
- const_iterator begin () const

  Returns a constant iterator to the beginning.
- iterator end ()

  Returns an iterator to the end.
- const_iterator end () const

  Returns a constant iterator to the end.
- size_type size () const

  Returns the number of elements.
- size_type max_size () const

  Returns the maximum possible number of elements.
- void resize (size_type sz)

  Resizes the vector to contain `sz` elements.
- void resize (size_type sz, const value_type &value)

  Resizes the vector to contain `sz` elements, each initialized to `value`.
- size_type capacity () const

  Returns the number of elements that can be held in currently allocated storage.
- bool empty () const noexcept

  Checks if the vector is empty.
- void reserve (size_type n)

  Requests that the vector capacity be at least enough to contain `n` elements.
- void **shrink_to_fit** ()

  Reduces capacity to fit the size.
- void **clear** () noexcept

  Clears the contents.
- iterator insert (const_iterator position, const value_type &val)

  Inserts `val` before `position`.
- iterator insert (iterator position, size_type n, const value_type &val)

  Inserts `n` elements of `val` before `position`.
- iterator erase (iterator position)

  Erases the element at `position`.
- iterator erase (iterator first, iterator last)

  Erases elements in the range [first, last).
- void push_back (const value_type &t)

  Appends `t` to the end.
- void push_back (value_type &&val)

  Appends `val` to the end.
- void **pop_back** ()

  Removes the last element.
- void swap (Vector &x)

  Swaps the contents with another Vector.
- bool operator== (const Vector< T > &other) const

  Checks if two Vectors are equal.

- bool operator!= (const Vector< T > &other) const
  Checks if two Vectors are not equal.
- bool operator< (const Vector< T > &other) const
  Checks if this Vector is less than another Vector.
- bool operator<= (const Vector< T > &other) const
  Checks if this Vector is less than or equal to another Vector.
- bool operator> (const Vector< T > &other) const
  Checks if this Vector is greater than another Vector.
- bool operator>= (const Vector< T > &other) const
  Checks if this Vector is greater than or equal to another Vector.

Constructors

- Vector ()
  Default constructor.
- Vector (size_type n, const T &t=T{})
  Fill constructor.
- Vector (const Vector &v)
  Copy constructor.
- template<class InputIterator >
  Vector (InputIterator first, InputIterator last)
  Range constructor.
- Vector (Vector &&v)
  Move constructor.
- Vector (const std::initializer_list< T > il)
  Initializer list constructor.

Assignment Operators

- Vector & operator= (const Vector &other)
  Copy assignment operator.
- Vector & operator= (Vector &&other)
  Move assignment operator.

Assign Functions

- template<class InputIterator >
  void assign (InputIterator first, InputIterator last)
  Assigns a range of elements to the Vector.
- void assign (size_type n, const value_type &val)
  Assigns `n` elements of value `val` to the Vector.
- void assign (std::initializer_list< value_type > il)
  Assigns elements from an initializer list to the Vector.

4.2.1 Detailed Description

template<typename T>
class Vector< T >

A templated dynamic array class.

This class provides a dynamic array similar to std::vector.

Template Parameters

T	Type of the elements.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 Vector() [1/6]

```
template<typename T >
Vector< T >::Vector ()  [inline]
```

Default constructor.

Constructs an empty Vector.

4.2.2.2 Vector() [2/6]

```
template<typename T >
Vector< T >::Vector (
            size_type n,
            const T & t = T{})  [inline], [explicit]
```

Fill constructor.

Constructs a Vector with `n` elements, each initialized to `t`.

Parameters

n	Number of elements.
t	Value to initialize elements with.

4.2.2.3 Vector() [3/6]

```
template<typename T >
Vector< T >::Vector (
            const Vector< T > & v)  [inline]
```

Copy constructor.

Constructs a Vector by copying another Vector.

Parameters

v	Vector to copy.

4.2.2.4 Vector() [4/6]

```
template<typename T >
template<class InputIterator >
Vector< T >::Vector (
            InputIterator first,
            InputIterator last)  [inline]
```

Range constructor.

Constructs a Vector with elements from the range [first, last).

Template Parameters

InputIterator	Iterator type.

Parameters

first	Start of the range.
last	End of the range.

4.2.2.5 Vector() [5/6]

```
template<typename T >
Vector< T >::Vector (
            Vector< T > && v)  [inline]
```

Move constructor.

Constructs a Vector by moving another Vector.

Parameters

v	Vector to move.

4.2.2.6 Vector() [6/6]

```
template<typename T >
Vector< T >::Vector (
            const std::initializer_list< T > il)  [inline]
```

Initializer list constructor.

Constructs a Vector with elements from an initializer list.

Parameters

il	Initializer list.

4.2.2.7 ~Vector()

```
template<typename T >
Vector< T >::~Vector ()  [inline]
```

Destructor.

Destructs the Vector and releases resources.

4.2.3 Member Function Documentation

4.2.3.1 assign() [1/3]

```
template<typename T >
template<class InputIterator >
void Vector< T >::assign (
            InputIterator first,
            InputIterator last)  [inline]
```

Assigns a range of elements to the Vector.

Template Parameters

InputIterator	Iterator type.

Parameters

first	Start of the range.
last	End of the range.

4.2.3.2 assign() [2/3]

```
template<typename T >
void Vector< T >::assign (
            size_type n,
            const value_type & val)  [inline]
```

Assigns `n` elements of value `val` to the Vector.

Parameters

n	Number of elements.
val	Value to assign.

4.2.3.3 assign() [3/3]

```
template<typename T >
void Vector< T >::assign (
            std::initializer_list< value_type > il)  [inline]
```

Assigns elements from an initializer list to the Vector.

Parameters

il	Initializer list.

4.2.3.4 at() [1/2]

```
template<typename T >
reference Vector< T >::at (
            size_type n)  [inline]
```

Accesses the element at position `n` with bounds checking.

Parameters

n	Position of the element.

Returns

Reference to the element.

Exceptions

std::out_of_range	If `n` is out of range.

4.2.3.5 at() [2/2]

```
template<typename T >
const_reference Vector< T >::at (
            size_type n) const  [inline]
```

Accesses the element at position `n` with bounds checking.

Parameters

n	Position of the element.

Returns

Constant reference to the element.

Exceptions

std::out_of_range	If `n` is out of range.

4.2.3.6 back() [1/2]

```
template<typename T >
reference Vector< T >::back ()  [inline]
```

Accesses the last element.

Returns

Reference to the last element.

4.2.3.7 back() [2/2]

```
template<typename T >
const_reference Vector< T >::back () const  [inline]
```

Accesses the last element.

Returns

Constant reference to the last element.

4.2.3.8 begin() [1/2]

```
template<typename T >
iterator Vector< T >::begin ()  [inline]
```

Returns an iterator to the beginning.

Returns

Iterator to the beginning.

4.2.3.9 begin() [2/2]

```
template<typename T >
const_iterator Vector< T >::begin () const  [inline]
```

Returns a constant iterator to the beginning.

Returns

Constant iterator to the beginning.

4.2.3.10 capacity()

```
template<typename T >
size_type Vector< T >::capacity () const  [inline]
```

Returns the number of elements that can be held in currently allocated storage.

Returns

Capacity of the vector.

4.2.3.11 data() [1/2]

```
template<typename T >
const value_type * Vector< T >::data () const  [inline], [noexcept]
```

Returns a constant pointer to the underlying data.

Returns

Constant pointer to the underlying data.

4.2.3.12 data() [2/2]

```
template<typename T >
value_type * Vector< T >::data ()  [inline], [noexcept]
```

Returns a pointer to the underlying data.

Returns

Pointer to the underlying data.

4.2.3.13 empty()

```
template<typename T >
bool Vector< T >::empty () const  [inline], [noexcept]
```

Checks if the vector is empty.

Returns

`true` if the vector is empty, `false` otherwise.

4.2.3.14 end() [1/2]

```
template<typename T >
iterator Vector< T >::end ()  [inline]
```

Returns an iterator to the end.

Returns

Iterator to the end.

4.2.3.15 end() [2/2]

```
template<typename T >
const_iterator Vector< T >::end () const  [inline]
```

Returns a constant iterator to the end.

Returns

Constant iterator to the end.

4.2.3.16 erase() [1/2]

```
template<typename T >
iterator Vector< T >::erase (
            iterator first,
            iterator last)  [inline]
```

Erases elements in the range [first, last).

Parameters

first	Start of the range.
last	End of the range.

Returns

Iterator pointing to the next element.

Exceptions

std::out_of_range	If the range is invalid.

4.2.3.17 erase() [2/2]

```
template<typename T >
iterator Vector< T >::erase (
            iterator position)  [inline]
```

Erases the element at `position`.

Parameters

position	Position of the element to erase.

Returns

Iterator pointing to the next element.

Exceptions

std::out_of_range	If `position` is out of range.

4.2.3.18 front() [1/2]

```
template<typename T >
reference Vector< T >::front ()  [inline]
```

Accesses the first element.

Returns

Reference to the first element.

4.2.3.19 front() [2/2]

```
template<typename T >
const_reference Vector< T >::front () const  [inline]
```

Accesses the first element.

Returns

Constant reference to the first element.

4.2.3.20 insert() [1/2]

```
template<typename T >
iterator Vector< T >::insert (
            const_iterator position,
            const value_type & val)  [inline]
```

Inserts `val` before `position`.

Parameters

position	Position to insert before.
val	Value to insert.

Returns

Iterator pointing to the inserted value.

4.2.3.21 insert() [2/2]

```
template<typename T >
iterator Vector< T >::insert (
            iterator position,
            size_type n,
            const value_type & val)  [inline]
```

Inserts `n` elements of `val` before `position`.

Parameters

position	Position to insert before.
n	Number of elements to insert.
val	Value to insert.

Returns

Iterator pointing to the first inserted value.

Exceptions

std::out_of_range	If `position` is out of range.

4.2.3.22 max_size()

```
template<typename T >
size_type Vector< T >::max_size () const  [inline]
```

Returns the maximum possible number of elements.

Returns

Maximum number of elements.

4.2.3.23 operator"!=()

```
template<typename T >
bool Vector< T >::operator!= (
            const Vector< T > & other) const  [inline]
```

Checks if two Vectors are not equal.

Parameters

other	Vector to compare.

Returns

`true` if the Vectors are not equal, `false` otherwise.

4.2.3.24 operator<()

```
template<typename T >
bool Vector< T >::operator< (
            const Vector< T > & other) const  [inline]
```

Checks if this Vector is less than another Vector.

Parameters

other	Vector to compare.

Returns

`true` if this Vector is less than the other Vector, `false` otherwise.

4.2.3.25 operator<=()

```
template<typename T >
bool Vector< T >::operator<= (
            const Vector< T > & other) const  [inline]
```

Checks if this Vector is less than or equal to another Vector.

Parameters

other	Vector to compare.

Returns

`true` if this Vector is less than or equal to the other Vector, `false` otherwise.

4.2.3.26 operator=() [1/2]

```
template<typename T >
Vector & Vector< T >::operator= (
            const Vector< T > & other)  [inline]
```

Copy assignment operator.

Assigns the contents of another Vector to this Vector.

Parameters

other	Vector to copy.

Returns

Reference to this Vector.

4.2.3.27 operator=() [2/2]

```
template<typename T >
Vector & Vector< T >::operator= (
            Vector< T > && other)  [inline]
```

Move assignment operator.

Moves the contents of another Vector to this Vector.

Parameters

other	Vector to move.

Returns

Reference to this Vector.

4.2.3.28 operator==()

```
template<typename T >
bool Vector< T >::operator== (
            const Vector< T > & other) const  [inline]
```

Checks if two Vectors are equal.

Parameters

other	Vector to compare.

Returns

`true` if the Vectors are equal, `false` otherwise.

4.2.3.29 operator>()

```
template<typename T >
bool Vector< T >::operator> (
            const Vector< T > & other) const  [inline]
```

Checks if this Vector is greater than another Vector.

Parameters

other	Vector to compare.

Returns

`true` if this Vector is greater than the other Vector, `false` otherwise.

4.2.3.30 operator>=()

```
template<typename T >
bool Vector< T >::operator>= (
            const Vector< T > & other) const  [inline]
```

Checks if this Vector is greater than or equal to another Vector.

Parameters

other	Vector to compare.

Returns

`true` if this Vector is greater than or equal to the other Vector, `false` otherwise.

4.2.3.31 operator[]() [1/2]

```
template<typename T >
reference Vector< T >::operator[] (
            size_type n)  [inline]
```

Accesses the element at position `n`.

Parameters

n	Position of the element.

Returns

Reference to the element.

4.2.3.32 operator[]() [2/2]

```
template<typename T >
const_reference Vector< T >::operator[] (
            size_type n) const  [inline]
```

Accesses the element at position `n`.

Parameters

n	Position of the element.

Returns

Constant reference to the element.

4.2.3.33 push_back() [1/2]

```
template<typename T >
void Vector< T >::push_back (
            const value_type & t)  [inline]
```

Appends `t` to the end.

Parameters

t	Value to append.

4.2.3.34 push_back() [2/2]

```
template<typename T >
void Vector< T >::push_back (
            value_type && val)  [inline]
```

Appends `val` to the end.

Parameters

val	Value to append.

4.2.3.35 reserve()

```
template<typename T >
void Vector< T >::reserve (
            size_type n)  [inline]
```

Requests that the vector capacity be at least enough to contain `n` elements.

Parameters

n	Minimum capacity requested.

4.2.3.36 resize() [1/2]

```
template<typename T >
void Vector< T >::resize (
            size_type sz)  [inline]
```

Resizes the vector to contain `sz` elements.

Parameters

sz	New size of the vector.

4.2.3.37 resize() [2/2]

```
template<typename T >
void Vector< T >::resize (
            size_type sz,
            const value_type & value)  [inline]
```

Resizes the vector to contain `sz` elements, each initialized to `value`.

Parameters

sz	New size of the vector.
value	Value to initialize elements with.

4.2.3.38 size()

```
template<typename T >
size_type Vector< T >::size () const  [inline]
```

Returns the number of elements.

Returns

Number of elements.

4.2.3.39 swap()

```
template<typename T >
void Vector< T >::swap (
            Vector< T > & x)  [inline]
```

Swaps the contents with another Vector.

Parameters

x	Vector to swap with.

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

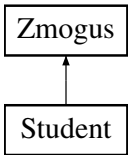
- vektorius.h

4.3 Zmogus Class Reference

Abstract base class representing a person.

`#include <student.h>`

Inheritance diagram for Zmogus:



Public Member Functions

- virtual void setName (const string &vardas)=0
  Sets the name of the person.
- virtual string getName () const =0
  Gets the name of the person.
- virtual void setSurname (const string &pavarde)=0
  Sets the surname of the person.
- virtual string getSurname () const =0
  Gets the surname of the person.
- virtual ~**Zmogus** ()
  Virtual destructor.

Protected Attributes

- string name
- string surname

4.3.1 Detailed Description

Abstract base class representing a person.

4.3.2 Member Function Documentation

4.3.2.1 getName()

```
virtual string Zmogus::getName () const  [pure virtual]
```

Gets the name of the person.

Returns

The name of the person.

Implemented in Student.

4.3.2.2 getSurname()

```
virtual string Zmogus::getSurname () const  [pure virtual]
```

Gets the surname of the person.

Returns

The surname of the person.

Implemented in Student.

4.3.2.3 setName()

```
virtual void Zmogus::setName (
            const string & vardas)  [pure virtual]
```

Sets the name of the person.

Parameters

vardas	The name to set.

Implemented in Student.

4.3.2.4 setSurname()

```
virtual void Zmogus::setSurname (
            const string & pavarde)  [pure virtual]
```

Sets the surname of the person.

Parameters

pavarde	The surname to set.

Implemented in Student.

4.3.3 Member Data Documentation

4.3.3.1 name

```
string Zmogus::name  [protected]
```

Name of the person

4.3.3.2 surname

```
string Zmogus::surname  [protected]
```

Surname of the person

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

- student.h

Chapter 5

File Documentation

5.1 main.cpp File Reference

A program for managing student data.

```
#include "student.h"
#include "vektorius.h"
```

Functions

- int main ()

  Main function of the program.

5.1.1 Detailed Description

A program for managing student data.

5.1.2 Function Documentation

5.1.2.1 main()

```
int main ()
```

Main function of the program.

Returns

0 on successful execution.

Perform actions for case '2'.

This function prompts the user to input the number of students in a group and the number of homework assignments. It then resizes a vector to accommodate the specified number of students, prompts for each student's name, generates random grades for the students, and displays either the average or median grades based on user choice.

Parameters

students	A vector containing objects representing students.
hw	The number of homework assignments.

Handles the case when user chooses option 3.

This function prompts the user to input the number of students in the group and the number of homework assignments. It then generates random data for the students and displays either their average grades or medians, based on user input.

Parameters

students	A vector containing Student objects.

$<$ Number of students in the group.

$<$ Number of homework assignments.

$<$ User's choice to display average (A) or median (M).

Handles case '4' of the main program menu.

This function prompts the user to input the number of students' data to read, then reads the data from a file, prompts the user to choose between displaying average or median grades, and then sorts and displays the student data accordingly.

Parameters

students	Vector of Student objects to store student data.
hw	Vector of vectors to store homework grades.

Perform operations for case '5'.

This function handles operations for case '5' which includes reading data from a file, sorting and displaying student information based on user input, and writing the results to a file.

Parameters

students	Vector of Student objects to store student data.
hw	Vector of vectors to store homework grades.

Case '6' logic for generating and processing student data.

This case handles the generation, processing, and sorting of student data. It prompts the user to input the number of students to generate, generates student data, reads the data from file, calculates results, sorts the students based on their results, and writes the sorted data to files.

Returns

1 if an invalid input is provided, otherwise 0.

Perform actions for case '7'.

This function executes the actions specific to case '7', which includes invoking the `testas()` function.

Returns

0 upon successful completion.

Case '8' of the program.

This case compares the performance of std::vector and a custom Vector implementation with different sizes of elements.

It measures the time taken to create vectors of Student objects using both std::vector and custom Vector.

- The sizes of the vectors tested are: 10000, 100000, 1000000, and 10000000.
- For each size, it measures the time taken to create vectors using both std::vector and custom Vector.
- The time measurements are printed in seconds.

A switch statement handling user choices.

This switch statement handles user input choices within a loop until the user chooses to exit. It provides messages for specific choices.

Parameters

choice	The user's input choice.

A loop continuing until the user chooses to exit.

This loop continues until the user inputs '9' to exit the program. It measures the execution time of the program and displays it when the program ends.

Main function of the program.

This function executes the main logic of the program, including user interaction and execution time measurement. It returns 0 upon successful execution.

Returns

0 upon successful execution.

5.2 student.h File Reference

Header file containing the definitions for the Zmogus and Student classes, and associated functions.

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <limits>
#include <string>
#include <vector>
#include <sstream>
#include <algorithm>
#include <numeric>
#include <random>
#include <ctime>
#include <cstring>
#include <cassert>
#include <utility>
#include <chrono>
```

Classes

- class Zmogus

  Abstract base class representing a person.
- class Student

  Class representing a student, derived from Zmogus.

Functions

- void generateRandomGrades (Vector< Student > &students, double hw)

  Generates random grades for each student in the provided vector.
- void generateRandomData (Vector< Student > &students, double hw)

  Generates random data for each student in the provided vector, including names and grades.
- double calculateMedian (Vector< int > &arr)

  Calculates the median of the given vector of integers.
- void readDataFromFile (Vector< Student > &students, double &hw, int N)

  Reads student data from a file and stores it in the provided vector.
- void enterDataManually (Vector< Student > &students, double hw)

  Manually enter data for each student.
- bool compareByName (const Student &a, const Student &b)

  Compares two students by their names.
- bool compareBySurname (const Student &a, const Student &b)

  Compares two students by their surnames.
- bool compareByMedian (const Student &a, const Student &b)

  Compares two students by their final median grades.
- bool compareByAvg (const Student &a, const Student &b)

  Compares two students by their final average grades.
- int generateRandomNumber (int min, int max)

  Generates a random number between given min and max values.
- double calculateAverage (const Vector< int > &pazymiai)

Calculates the average of grades in a vector.

- void failuGeneravimas (int studentu_kiekis, const std::string &failo_pavadinimas)

  Generates student data and writes it to a file.
- bool Nuskaitymas (const std::string &failo_pavadinimas, Vector< Student > &students, int studentukiekis)

  Reads student data from a file.
- std::chrono::steady_clock::time_point DabartinisLaikas ()

  Returns the current time as a steady clock time point.
- double LaikoSkirtumas (const std::chrono::steady_clock::time_point &pradzia, const std::chrono::steady_clock::time_point &pabaiga)

  Calculates the difference in seconds between two steady clock time points.
- void **calculateResults** (Vector< Student > &stud)
- double skaiciuotiVidurki (const Vector< int > &pazymiai)

  Calculates the average based on student grades.
- bool arGerasStudentas (const Student &student)

  Checks if a student is considered good based on average grade.
- bool lygintiPagalVidurki (const Student &a, const Student &b)

  Compares two students based on their average grade.
- void rusiuoja_ir_raso_failus (Vector< Student > &students)

  Sorts students, separates them into good and bad, and writes them to files.
- void testas ()

  Test function for the Student class.
- void testStdVectorPerformance ()

  Test the performance of std::vector<Student> and Vector<Student>.
- void testCustomVectorPerformance ()

  Test the performance of Vector<Student>.

5.2.1 Detailed Description

Header file containing the definitions for the Zmogus and Student classes, and associated functions.

This file defines the abstract base class Zmogus and the derived class Student, along with various utility functions for handling student data.

5.2.2 Function Documentation

5.2.2.1 arGerasStudentas()

```
bool arGerasStudentas (
            const Student & student)
```

Checks if a student is considered good based on average grade.

Parameters

student	The student object.

Returns

true if the student is considered good (average grade >= 5.0), false otherwise.

5.2.2.2 calculateAverage()

```
double calculateAverage (
            const Vector< int > & pazymiai)
```

Calculates the average of grades in a vector.

Parameters

pazymiai	Vector containing grades.

Returns

The average of grades in the vector.

5.2.2.3 calculateMedian()

```
double calculateMedian (
            Vector< int > & arr)
```

Calculates the median of the given vector of integers.

Parameters

arr	Vector of integers for which the median will be calculated.

Returns

Median value of the vector.

5.2.2.4 compareByAvg()

```
bool compareByAvg (
            const Student & a,
            const Student & b)
```

Compares two students by their final average grades.

Parameters

a	The first student to compare.
b	The second student to compare.

Returns

true if the final average grade of student a is less than that of student b, false otherwise.

5.2.2.5 compareByMedian()

```
bool compareByMedian (
            const Student & a,
            const Student & b)
```

Compares two students by their final median grades.

Parameters

a	The first student to compare.
b	The second student to compare.

Returns

true if the final median grade of student a is less than that of student b, false otherwise.

5.2.2.6 compareByName()

```
bool compareByName (
            const Student & a,
            const Student & b)
```

Compares two students by their names.

Parameters

a	The first student to compare.
b	The second student to compare.

Returns

true if the name of student a is less than the name of student b, false otherwise.

5.2.2.7 compareBySurname()

```
bool compareBySurname (
            const Student & a,
            const Student & b)
```

Compares two students by their surnames.

Parameters

a	The first student to compare.
b	The second student to compare.

Returns

true if the surname of student a is less than the surname of student b, false otherwise.

5.2.2.8 DabartinisLaikas()

```
std::chrono::steady_clock::time_point DabartinisLaikas ()
```

Returns the current time as a steady clock time point.

Returns

std::chrono::steady_clock::time_point The current time.

5.2.2.9 enterDataManually()

```
void enterDataManually (
            Vector< Student > & students,
            double hw)
```

Manually enter data for each student.

This function prompts the user to enter the name, surname, homework grades, and exam result for each student in the provided vector. It calculates the final average for each student based on the homework grades and exam result, and updates the corresponding student objects.

Parameters

students	A vector of Student objects to be populated with data.
hw	The number of homework assignments per student.

Note

This function assumes that each Student object in the vector already has default or initialized values for other attributes (e.g., ID).

Exception handling is implemented to handle input errors gracefully.

5.2.2.10 failuGeneravimas()

```
void failuGeneravimas (
            int studentu_kiekis,
            const std::string & failo_pavadinimas)
```

Generates student data and writes it to a file.

This function generates student data consisting of names, surnames, and grades, and writes it to a specified file.

Parameters

studentu_kiekis	The number of students to generate data for.
failo_pavadinimas	The name of the file to write the data to.

5.2.2.11 generateRandomData()

```
void generateRandomData (
            Vector< Student > & students,
            double hw)
```

Generates random data for each student in the provided vector, including names and grades.

Parameters

students	Vector of students for whom random data will be generated.
hw	Number of homework assignments to generate grades for.

5.2.2.12 generateRandomGrades()

```
void generateRandomGrades (
            Vector< Student > & students,
            double hw)
```

Generates random grades for each student in the provided vector.

Parameters

students	Vector of students for whom random grades will be generated.
hw	Number of homework assignments to generate grades for.

5.2.2.13 generateRandomNumber()

```
int generateRandomNumber (
            int min,
            int max)
```

Generates a random number between given min and max values.

Parameters

min	The minimum value for the random number.
max	The maximum value for the random number.

Returns

A random number between min and max (inclusive).

5.2.2.14 LaikoSkirtumas()

```
double LaikoSkirtumas (
            const std::chrono::steady_clock::time_point & pradzia,
            const std::chrono::steady_clock::time_point & pabaiga)
```

Calculates the difference in seconds between two steady clock time points.

Parameters

pradzia	The starting time point.
pabaiga	The ending time point.

Returns

double The time difference in seconds.

5.2.2.15 lygintiPagalVidurki()

```
bool lygintiPagalVidurki (
            const Student & a,
            const Student & b)
```

Compares two students based on their average grade.

Parameters

a	First student object.
b	Second student object.

Returns

true if the average grade of student 'a' is greater than that of student 'b', false otherwise.

5.2.2.16 Nuskaitymas()

```
bool Nuskaitymas (
            const std::string & failo_pavadinimas,
            Vector< Student > & students,
            int studentukiekis)
```

Reads student data from a file.

This function reads student data from a specified file and populates a vector of Student objects.

Parameters

failo_pavadinimas	The name of the file to read the data from.
students	A vector to store the Student objects.
studentukiekis	The number of students in the file.

Returns

True if the file was successfully read, false otherwise.

5.2.2.17 readDataFromFile()

```
void readDataFromFile (
            Vector< Student > & students,
            double & hw,
            int N)
```

Reads student data from a file and stores it in the provided vector.

Parameters

students	Vector to store the read student data.
hw	Reference to store the number of homework assignments.
N	Number of students to read from the file.

5.2.2.18 rusiuoja_ir_raso_failus()

```
void rusiuoja_ir_raso_failus (
            Vector< Student > & students)
```

Sorts students, separates them into good and bad, and writes them to files.

Parameters

students	Vector of student objects.

5.2.2.19 skaiciuotiVidurki()

```
double skaiciuotiVidurki (
            const Vector< int > & pazymiai)
```

Calculates the average based on student grades.

Parameters

pazymiai	Vector of student grades.

Returns

double The calculated average.

5.2.2.20 testas()

```
void testas ()
```

Test function for the Student class.

This function tests the default constructor, setters, getters, copy constructor, move constructor, copy assignment, and move assignment of the Student class. It includes assertions to ensure that the class functions correctly.

5.2.2.21 testCustomVectorPerformance()

```
void testCustomVectorPerformance ()
```

Test the performance of Vector<Student>.

This function measures the time taken to initialize Vector<Student> with different sizes.

Note

Requires the definition of DabartinisLaikas() and LaikoSkirtumas() functions.

5.2.2.22 testStdVectorPerformance()

```
void testStdVectorPerformance ()
```

Test the performance of std::vector<Student> and Vector<Student>.

This function compares the performance of std::vector<Student> and Vector<Student> by measuring the time taken to initialize vectors of different sizes.

Note

Requires the definition of DabartinisLaikas() and LaikoSkirtumas() functions.

5.3 student.h

Go to the documentation of this file.

```
00001
00009 #ifndef STUDENT_H
00010 #define STUDENT_H
00011
00012 #include <iostream>
00013 #include <fstream>
00014 #include <iomanip>
00015 #include <limits>
00016 #include <string>
00017 #include <vector>
00018 #include <sstream>
00019 #include <algorithm>
00020 #include <numeric>
00021 #include <random>
00022 #include <ctime>
00023 #include <cstring>
00024 #include <cassert>
00025 #include <utility>
00026 #include <chrono>
00027
00028 using namespace std;
00029 using namespace std::chrono;
00030
00034 class Zmogus {
00035 protected:
00036     string name;
00037     string surname;
00039 public:
00044     virtual void setName(const string& vardas) = 0;
00045
00050     virtual string getName() const = 0;
00051
00056     virtual void setSurname(const string& pavarde) = 0;
00057
00062     virtual string getSurname() const = 0;
00063
00067     virtual ~Zmogus() {}
00068 };
00069
00073 class Student : public Zmogus {
00074 private:
00075     Vector<int> grades;
00076     int exam_result;
00077     double final_avg;
00078     double final_median;
00080 public:
00084     Student();
00085
00089     ~Student();
00090
00091     // Implementation of abstract methods from Zmogus
00092     void setName(const string& vardas) override;
00093     string getName() const override;
00094     void setSurname(const string& pavarde) override;
00095     string getSurname() const override;
00096
00101     void setExamResult(int egzaminas);
00102
00107     int getExamResult() const;
```

```
00108
00113     void setFinalAvg(double Gal_vid);
00114
00119     double getFinalAvg() const;
00120
00125     void setFinalMedian(double Gal_med);
00126
00131     double getFinalMedian() const;
00132
00137     void setSingleGrade(int naujasnd);
00138
00144     int getSingleGrade(int i) const;
00145
00150     void setGrades(const Vector<int>& ND);
00151
00156     Vector<int> getGrades() const;
00157
00161     void clearData();
00162
00163     // Friend functions for input and output operations
00164     friend std::istream& operator»(istream& in, Student& student);
00165     friend std::ostream& operator«(ostream& out, const Student& student);
00166 };
00167
00168 // Function declarations
00169 void generateRandomGrades(Vector<Student>& students, double hw);
00170 void generateRandomData(Vector<Student>& students, double hw);
00171 double calculateMedian(Vector<int>& arr);
00172 void readDataFromFile(Vector<Student>& students, double& hw, int N);
00173 void enterDataManually(Vector<Student>& students, double hw);
00174 bool compareByName(const Student& a, const Student& b);
00175 bool compareBySurname(const Student& a, const Student& b);
00176 bool compareByMedian(const Student& a, const Student& b);
00177 bool compareByAvg(const Student& a, const Student& b);
00178 int generateRandomNumber(int min, int max);
00179 double calculateAverage(const Vector<int>& pazymiai);
00180 void failuGeneravimas(int studentu_kiekis, const std::string& failo_pavadinimas);
00181 bool Nuskaitymas(const std::string& failo_pavadinimas, Vector<Student>& students, int studentukiekis);
00182 std::chrono::steady_clock::time_point DabartinisLaikas();
00183 double LaikoSkirtumas(const std::chrono::steady_clock::time_point& pradzia, const
      std::chrono::steady_clock::time_point& pabaiga);
00184 void calculateResults(Vector<Student>& stud);
00185 double skaiciuotiVidurki(const Vector<int>& pazymiai);
00186 bool arGerasStudentas(const Student& student);
00187 bool lygintiPagalVidurki(const Student& a, const Student& b);
00188 void rusiuoja_ir_raso_failus(Vector<Student>& students);
00189 void testas();
00190 void testStdVectorPerformance();
00191 void testCustomVectorPerformance();
00192
00193 #endif // STUDENT_H
```

5.4 vektorius.h

```
00001
00009 #pragma once
00010
00011 #include <iostream>
00012 #include <memory>
00013 #include <algorithm>
00014 #include <limits>
00015
00023 template <typename T>
00024 class Vector {
00025 public:
00026     // MEMBER TYPE
00027     typedef size_t size_type;
00028     typedef T value_type;
00029     typedef T& reference;
00030     typedef const T& const_reference;
00031     typedef T* iterator;
00032     typedef const T* const_iterator;
00033
00034     // MEMBER FUNCTIONS
00035
00038
00043     Vector() { create(); }
00044
00053     explicit Vector(size_type n, const T& t = T{}) { create(n, t); }
00054
00062     Vector(const Vector& v) { create(v.begin(), v.end()); }
00063
00073     template <class InputIterator>
```

```
00074     Vector(InputIterator first, InputIterator last) { create(first, last); }
00075
00083     Vector(Vector&& v) {
00084         create();
00085         swap(v);
00086         v.uncreate();
00087     }
00088
00096     Vector(const std::initializer_list<T> il) { create(il.begin(), il.end()); }
00098
00104     ~Vector() { uncreate(); }
00105
00108
00116     Vector& operator=(const Vector& other) {
00117         if (this != &other) {
00118             uncreate();
00119             create(other.begin(), other.end());
00120         }
00121         return *this;
00122     }
00123
00132     Vector& operator=(Vector&& other) {
00133         if (this != &other) {
00134             uncreate();
00135             swap(other);
00136             other.uncreate();
00137         }
00138         return *this;
00139     }
00141
00144
00151     template <class InputIterator>
00152     void assign(InputIterator first, InputIterator last) {
00153         uncreate();
00154         create(first, last);
00155     }
00156
00163     void assign(size_type n, const value_type& val) {
00164         uncreate();
00165         create(n, val);
00166     }
00167
00173     void assign(std::initializer_list<value_type> il) {
00174         uncreate();
00175         create(il.begin(), il.end());
00176     }
00178
00179     // Element access
00187     const_reference at(size_type n) const {
00188         if (n >= size() || n < 0)
00189             throw std::out_of_range("Index out of range");
00190         return dat[n];
00191     }
00192
00199     reference operator[](size_type n) { return dat[n]; }
00200
00207     const_reference operator[](size_type n) const { return dat[n]; }
00208
00216     reference at(size_type n) {
00217         if (n >= size() || n < 0)
00218             throw std::out_of_range("Index out of range");
00219         return dat[n];
00220     }
00221
00227     reference front() { return dat[0]; }
00228
00234     const_reference front() const { return dat[0]; }
00235
00241     reference back() { return dat[size() - 1]; }
00242
00248     const_reference back() const { return dat[size() - 1]; }
00249
00255     value_type* data() noexcept { return dat; }
00256
00262     const value_type* data() const noexcept { return dat; }
00263
00264     // Iterators
00270     iterator begin() { return dat; }
00271
00277     const_iterator begin() const { return dat; }
00278
00284     iterator end() { return avail; }
00285
00291     const_iterator end() const { return avail; }
00292
00293     // Capacity
00299     size_type size() const { return avail - dat; }
```

```
00300
00306     size_type max_size() const { return std::numeric_limits<size_type>::max(); }
00307
00313     void resize(size_type sz) {
00314         if (sz < size()) {
00315             iterator it = dat + sz;
00316             while (it != avail) {
00317                 alloc.destroy(it++);
00318             }
00319             avail = dat + sz;
00320         } else if (sz > capacity()) {
00321             grow(sz);
00322             std::uninitialized_fill(avail, dat + sz, value_type());
00323             avail = dat + sz;
00324         } else if (sz > size()) {
00325             std::uninitialized_fill(avail, dat + sz, value_type());
00326             avail = dat + sz;
00327         }
00328     }
00329
00336     void resize(size_type sz, const value_type& value) {
00337         if (sz > capacity()) {
00338             grow(sz);
00339         }
00340
00341         if (sz > size()) {
00342             insert(end(), sz - size(), value);
00343         } else if (sz < size()) {
00344             avail = dat + sz;
00345         }
00346     }
00347
00353     size_type capacity() const { return limit - dat; }
00354
00360     bool empty() const noexcept { return size() == 0; }
00361
00367     void reserve(size_type n) {
00368         if (n > capacity()) {
00369             grow(n);
00370         }
00371     }
00372
00376     void shrink_to_fit() {
00377         if (limit > avail) {
00378             limit = avail;
00379         }
00380     }
00381
00382     // Modifiers
00386     void clear() noexcept {
00387         uncreate();
00388     }
00389
00397     iterator insert(const_iterator position, const value_type& val) {
00398         return insert(position, 1, val);
00399     }
00400
00410     iterator insert(iterator position, size_type n, const value_type& val) {
00411         if (position < begin() || position > end()) {
00412             throw std::out_of_range("Index out of range");
00413         }
00414         if (avail + n > limit) {
00415             size_type index = position - begin();
00416             grow(n);
00417             position = begin() + index;
00418         }
00419         for (iterator it = end() + n - 1; it != position + n - 1; --it) {
00420             *it = std::move(*(it - n));
00421         }
00422         std::uninitialized_fill(position, position + n, val);
00423         avail += n;
00424
00425         return position;
00426     }
00427
00435     iterator erase(iterator position) {
00436         if (position < dat || position >= avail) {
00437             throw std::out_of_range("Index out of range");
00438         }
00439         std::move(position + 1, avail, position);
00440         alloc.destroy(--avail);
00441
00442         return position;
00443     }
00444
00453     iterator erase(iterator first, iterator last) {
00454         if (first < dat || last > avail || first > last) {
```

```
00455             throw std::out_of_range("Index out of range");
00456         }
00457         iterator new_avail = std::move(last, avail, first);
00458         for (iterator it = new_avail; it != avail; ++it) {
00459             alloc.destroy(it);
00460         }
00461         avail = new_avail;
00462 
00463         return first;
00464     }
00465 
00471     void push_back(const value_type& t) {
00472         if (avail == limit)
00473             grow();
00474         unchecked_append(t);
00475     }
00476 
00482     void push_back(value_type&& val) {
00483         if (avail == limit)
00484             grow();
00485         unchecked_append(std::move(val));
00486     }
00487 
00491     void pop_back() {
00492         if (avail != dat)
00493             alloc.destroy(--avail);
00494     }
00495 
00501     void swap(Vector& x) {
00502         std::swap(dat, x.dat);
00503         std::swap(avail, x.avail);
00504         std::swap(limit, x.limit);
00505     }
00506 
00507     // NON-MEMBER FUNCTIONS
00508 
00515     bool operator==(const Vector<T>& other) const {
00516         if (size() != other.size()) {
00517             return false;
00518         }
00519 
00520         return std::equal(begin(), end(), other.begin());
00521     }
00522 
00529     bool operator!=(const Vector<T>& other) const {
00530         return !(*this == other);
00531     }
00532 
00539     bool operator<(const Vector<T>& other) const {
00540         return std::lexicographical_compare(begin(), end(), other.begin(), other.end());
00541     }
00542 
00549     bool operator<=(const Vector<T>& other) const {
00550         return !(other < *this);
00551     }
00552 
00559     bool operator>(const Vector<T>& other) const {
00560         return std::lexicographical_compare(other.begin(), other.end(), begin(), end());
00561     }
00562 
00569     bool operator>=(const Vector<T>& other) const {
00570         return !(other > *this);
00571     }
00572 
00573 private:
00574     iterator dat;
00575     iterator avail;
00576     iterator limit;
00577     std::allocator<T> alloc;
00578 
00582     void create() { dat = avail = limit = nullptr; }
00583 
00590     void create(size_type n, const T& val) {
00591         dat = alloc.allocate(n);
00592         limit = avail = dat + n;
00593         std::uninitialized_fill(dat, limit, val);
00594     }
00595 
00602     void create(const_iterator i, const_iterator j) {
00603         dat = alloc.allocate(j - i);
00604         limit = avail = std::uninitialized_copy(i, j, dat);
00605     }
00606 
00610     void uncreate() {
00611         if (dat) {
00612             iterator it = avail;
00613             while (it != dat) {
```

```
00614                 alloc.destroy(--it);
00615             }
00616             alloc.deallocate(dat, limit - dat);
00617         }
00618         dat = limit = avail = nullptr;
00619     }
00620
00626     void grow(size_type new_capacity = 0) {
00627         size_type new_size = std::max(new_capacity, 2 * capacity());
00628         iterator new_data = alloc.allocate(new_size);
00629         iterator new_avail = std::uninitialized_copy(std::make_move_iterator(dat),
      std::make_move_iterator(avail), new_data);
00630         uncreate();
00631         dat = new_data;
00632         avail = new_avail;
00633         limit = dat + new_size;
00634     }
00635
00641     void unchecked_append(const T& val) {
00642         alloc.construct(avail++, val);
00643     }
00644
00650     void unchecked_append(T&& val) {
00651         alloc.construct(avail++, std::move(val));
00652     }
00653 };
```

Index