v3.0

Generated by Doxygen 1.10.0

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

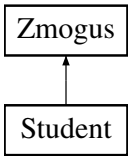
Here is a list of all documented files with brief descriptions:

Chapter 4

Class Documentation

4.1 Student Class Reference

Inheritance diagram for Student:



Public Member Functions

- void ppp () const override
- **Student** (const string &fName, const string &lName, const Vector< int > &grades, int finalExamGrade, double median, double average)
- **Student** (const Student &other)
- **Student** (Student &&other) noexcept
- Student & **operator=** (const Student &other)
- Student & **operator=** (Student &&other) noexcept
- const Vector< int > & **getGrades** () const
- int **getFinalExamGrade** () const
- double **getMedian** () const
- double **getAverage** () const
- double **getFinalMedian** () const
- double **getFinalAverage** () const
- double **getFinalGrade** () const
- void **setGrades** (const Vector< int > &newGrades)
- void **setFinalExamGrade** (int examGrade)
- void **setMedian** (double medianValue)
- void **setAverage** (double averageValue)
- void **setFinalMedian** (double finalMedian)
- void **setFinalAverage** (double finalAverage)
- void **setFinalGrade** (double finalGradeValue)

Public Member Functions inherited from Zmogus

- string **getFirstName** () const
- string **getLastName** () const
- void **setFirstName** (const string &fName)
- void **setLastName** (const string &lName)

Friends

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

Additional Inherited Members

Protected Member Functions inherited from Zmogus

- **Zmogus** (const string &fName, const string &lName)

Protected Attributes inherited from Zmogus

- string **firstName**
- string **lastName**

4.1.1 Member Function Documentation

4.1.1.1 ppp()

```
void Student::ppp ( ) const  [inline], [override], [virtual]
```

Implements Zmogus.

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

- sources/student.h

4.2 Vector< T > Class Template Reference

Public Types

- using **size_type** = size_t
- using **value_type** = T
- using **reference** = T&
- using **const_reference** = const T&
- using **iterator** = T*
- using **const_iterator** = const T*

Public Member Functions

- **Vector** (std::initializer_list< T > il)
- **Vector** (size_type n)
- **Vector** (const Vector &v)
- **Vector** (Vector &&v) noexcept
- Vector & **operator=** (const Vector &other)
- Vector & **operator=** (Vector &&other) noexcept
- reference **at** (size_type n)
- const_reference **at** (size_type n) const
- reference **operator[ ]** (size_type n)
- const_reference **operator[ ]** (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
- iterator **begin** () noexcept
- const_iterator **begin** () const noexcept
- iterator **end** () noexcept
- const_iterator **end** () const noexcept
- bool **empty** () const noexcept
- size_type **capacity** () const noexcept
- size_type **size** () const noexcept
- size_type **max_size** () const noexcept
- void **reserve** (size_type n)
- void **resize** (size_type sz)
- void **resize** (size_type sz, const value_type &value)
- void **shrink_to_fit** ()
- void **clear** () noexcept
- void **push_back** (const value_type &val)
- void **push_back** (value_type &&val)
- void **pop_back** ()
- iterator **insert** (iterator position, const value_type &val)
- iterator **insert** (iterator position, size_type n, const value_type &val)
- iterator **erase** (iterator position)
- iterator **erase** (iterator first, iterator last)

Friends

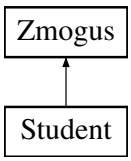
- bool **operator==** (const Vector< T > &lhs, const Vector< T > &rhs)
- bool **operator!=** (const Vector< T > &lhs, const Vector< T > &rhs)

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

- sources/vector.h

4.3 Zmogus Class Reference

Inheritance diagram for Zmogus:



Public Member Functions

- string **getFirstName** () const
- string **getLastName** () const
- void **setFirstName** (const string &fName)
- void **setLastName** (const string &lName)

Protected Member Functions

- **Zmogus** (const string &fName, const string &lName)
- virtual void **ppp** () const =0

Protected Attributes

- string **firstName**
- string **lastName**

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

- sources/student.h

Chapter 5

File Documentation

5.1 failu-generavimas.h

```
00001 #ifndef FAILU_GENERAVIMAS_H
00002 #define FAILU_GENERAVIMAS_H
00003
00004 #include <iostream>
00005 #include <fstream>
00006 #include <iomanip>
00007 #include <vector>
00008 #include <chrono>
00009 #include <thread>
00010 #include <string>
00011 #include <ctime>
00012 #include <cstdlib>
00013 #include "vektoriai.h"
00014 #include "student.h"
00015
00016 using namespace std;
00017
00018 void writeCategorizedStudents(const Vector<Student>& students, const string& filename);
00019 void generateFiles();
00020 void sortAndWriteToFile(const string& inputFilename);
00021 void generatingFinal();
00022
00023 #endif
```

5.2 student.h

```
00001 #ifndef STUDENT_H
00002 #define STUDENT_H
00003
00004 #include "vektoriai.h"
00005 #include "vector.h"
00006
00007 #include <vector>
00008 #include <string>
00009 #include <iomanip>
00010 #include <algorithm>
00011
00012 using namespace std;
00013
00014 class Zmogus {
00015 protected:
00016     string firstName;
00017     string lastName;
00018
00019     Zmogus() = default;
00020     Zmogus(const string& fName, const string& lName) : firstName(fName), lastName(lName) {}
00021     virtual void ppp() const = 0;
00022
00023 public:
00024     virtual ~Zmogus() {}
00025     string getFirstName() const { return firstName; }
00026     string getLastName() const { return lastName; }
00027
00028     void setFirstName(const string& fName) { firstName = fName; }
```

```
00029     void setLastName(const string& lName) { lastName = lName; }
00030 };
00031
00032 class Student : public Zmogus {
00033 private:
00034     Vector<int> grades;
00035     int finalExamGrade;
00036     double median, average;
00037     double fin_median, fin_average, finalGrade;
00038
00039 public:
00040     void ppp() const override {}
00041     Student() : finalExamGrade(0), median(0.0), average(0.0), fin_median(0.0), fin_average(0.0),
      finalGrade(0.0) {}
00042     Student(const string& fName, const string& lName, const Vector<int>& grades, int finalExamGrade,
      double median, double average)
00043      : Zmogus(fName,lName), grades(grades), finalExamGrade(finalExamGrade), median(median),
      average(average), fin_median(0.0), fin_average(0.0), finalGrade(0.0) {}
00044
00045    // Destructor
00046    ~Student() {
00047     grades.clear();
00048     firstName.clear();
00049     lastName.clear();
00050     }
00051
00052     // Copy Constructor
00053     Student(const Student& other)
00054     : Zmogus(other.firstName, other.lastName), grades(other.grades),
      finalExamGrade(other.finalExamGrade), median(other.median), average(other.average),
      fin_median(other.fin_median), fin_average(other.fin_average), finalGrade(other.finalGrade) {}
00055
00056     // Move Constructor
00057     Student(Student&& other) noexcept
00058     : Zmogus(move(other.firstName), move(other.lastName)),
00059       grades(move(other.grades)),
00060       finalExamGrade(move(other.finalExamGrade)),
00061       median(move(other.median)),
00062       average(move(other.average)),
00063       fin_median(move(other.fin_median)),
00064       fin_average(move(other.fin_average)),
00065       finalGrade(move(other.finalGrade)) {
00066
00067     other.finalExamGrade = 0;
00068     other.median = 0;
00069     other.average = 0;
00070     other.fin_median = 0;
00071     other.fin_average = 0;
00072     other.finalGrade = 0;
00073     other.firstName.clear();
00074     other.lastName.clear();
00075     other.grades.clear();
00076 }
00077
00078     // Copy Assignment Operator
00079     Student& operator=(const Student& other) {
00080         if (this != &other) { // self-assignment check
00081         Zmogus::setFirstName(other.getFirstName());
00082         Zmogus::setLastName(other.getLastName());
00083             grades = other.grades;
00084             finalExamGrade = other.finalExamGrade;
00085             median = other.median;
00086             average = other.average;
00087             fin_median = other.fin_median;
00088             fin_average = other.fin_average;
00089             finalGrade = other.finalGrade;
00090         }
00091         return *this;
00092     }
00093
00094     // Move Assignment Operator
00095     Student& operator=(Student&& other) noexcept {
00096         if (this != &other) {
00097         Zmogus::setFirstName(move(other.getFirstName()));
00098         Zmogus::setLastName(move(other.getLastName()));
00099             grades = std::move(other.grades);
00100             finalExamGrade = std::move(other.finalExamGrade);
00101             median = std::move(other.median);
00102             average = std::move(other.average);
00103             fin_median = std::move(other.fin_median);
00104             fin_average = std::move(other.fin_average);
00105             finalGrade = std::move(other.finalGrade);
00106
00107             other.finalExamGrade = 0;
00108             other.median = 0;
00109             other.average = 0;
00110             other.fin_median = 0;
```

```
00111             other.fin_average = 0;
00112             other.finalGrade = 0;
00113             other.firstName.clear();
00114             other.lastName.clear();
00115             other.grades.clear();
00116 
00117         }
00118         return *this;
00119     }
00120     // Input Operator
00121 friend std::istream& operator»(istream& i, Student& student) {
00122     string firstName, lastName;
00123     i » firstName » lastName;
00124     student.setFirstName(firstName);
00125     student.setLastName(lastName);
00126 
00127     Vector<int> grades;
00128     for (int j = 0; j < 15; ++j) {
00129         int grade;
00130         i » grade;
00131         grades.push_back(grade);
00132     }
00133     student.setGrades(grades);
00134 
00135     i » student.finalExamGrade;
00136 
00137     // final average
00138     double sum = 0;
00139     for (int grade : grades) {
00140         sum += grade;
00141     }
00142     double average = sum / grades.size();
00143     double finalAverage = average * 0.4 + student.finalExamGrade * 0.6;
00144     student.setFinalAverage(finalAverage);
00145 
00146     // final median
00147     sort(grades.begin(), grades.end());
00148     double finalMedian;
00149     if (grades.size() % 2 == 0) {
00150         finalMedian = (grades[grades.size() / 2 - 1] + grades[grades.size() / 2]) / 2.0;
00151     } else {
00152         finalMedian = grades[grades.size() / 2];
00153     }
00154     finalMedian = finalMedian * 0.4 + student.finalExamGrade * 0.6;
00155     student.setFinalMedian(finalMedian);
00156 
00157     return i;
00158 }
00159 
00160 // Output Operator
00161 friend std::ostream& operator«(std::ostream& os, const Student& student) {
00162     os « setw(10) « student.getFirstName() « setw(20) « student.getLastName();
00163     double average = student.getAverage() * 0.4 + student.getFinalExamGrade() * 0.6;
00164     double median = student.getMedian() * 0.4 + student.getFinalExamGrade() * 0.6;
00165     os « fixed « setw(25) « setprecision(2) « average;
00166     os « fixed « setw(25) « setprecision(2) « median « '\n';
00167     return os;
00168 }
00169 
00170     const Vector<int>& getGrades() const { return grades; }
00171     int getFinalExamGrade() const { return finalExamGrade; }
00172     double getMedian() const { return median; }
00173     double getAverage() const { return average; }
00174     double getFinalMedian() const { return fin_median; }
00175     double getFinalAverage() const { return fin_average; }
00176     double getFinalGrade() const { return finalGrade; }
00177 
00178 
00179     void setGrades(const Vector<int>& newGrades) { grades = newGrades; }
00180     void setFinalExamGrade(int examGrade) { finalExamGrade = examGrade; }
00181     void setMedian(double medianValue) { median = medianValue; }
00182     void setAverage(double averageValue) { average = averageValue; }
00183     void setFinalMedian(double finalMedian) { fin_median = finalMedian; }
00184     void setFinalAverage(double finalAverage) { fin_average = finalAverage; }
00185     void setFinalGrade(double finalGradeValue) { finalGrade = finalGradeValue; }
00186 };
00187 
00188 #endif
```

5.3 vector.h

```
00001 #pragma once
00002 
```

```
00003 #include <iostream>
00004 #include <memory>
00005 #include <initializer_list>
00006 #include <limits>
00007 #include <algorithm>
00008 #include <stdexcept>
00009
00010 template <typename T>
00011 class Vector {
00012 public:
00013     using size_type = size_t;
00014     using value_type = T;
00015     using reference = T&;
00016     using const_reference = const T&;
00017     using iterator = T*;
00018     using const_iterator = const T*;
00019
00020     // Constructors
00021     Vector() noexcept : dat(nullptr), avail(nullptr), limit(nullptr) {}
00022     Vector(std::initializer_list<T> il) { create(il.begin(), il.end()); }
00023
00024     // Fill constructor
00025     explicit Vector(size_type n) { create(n, T{}); }
00026
00027     // Destructor
00028     ~Vector() { uncreate(); }
00029
00030     // Copy constructor
00031     Vector(const Vector& v) { create(v.begin(), v.end()); }
00032
00033     // Move constructor
00034     Vector(Vector&& v) noexcept { move_from(std::move(v)); }
00035
00036     // Copy assignment
00037     Vector& operator=(const Vector& other) {
00038         if (this != &other) {
00039             uncreate();
00040             create(other.begin(), other.end());
00041         }
00042         return *this;
00043     }
00044
00045     // Move assignment
00046     Vector& operator=(Vector&& other) noexcept {
00047         if (this != &other) {
00048             uncreate();
00049             move_from(std::move(other));
00050         }
00051         return *this;
00052     }
00053
00054     // Comparison operators
00055     friend bool operator==(const Vector<T>& lhs, const Vector<T>& rhs) {
00056         return lhs.size() == rhs.size() && std::equal(lhs.begin(), lhs.end(), rhs.begin());
00057     }
00058
00059     friend bool operator!=(const Vector<T>& lhs, const Vector<T>& rhs) {
00060         return !(lhs == rhs);
00061     }
00062
00063     // Element access
00064     reference at(size_type n) {
00065         if (n >= size())
00066             throw std::out_of_range("Index out of range");
00067         return dat[n];
00068     }
00069
00070     const_reference at(size_type n) const {
00071         if (n >= size())
00072             throw std::out_of_range("Index out of range");
00073         return dat[n];
00074     }
00075
00076     reference operator[](size_type n) { return dat[n]; }
00077     const_reference operator[](size_type n) const { return dat[n]; }
00078
00079     reference front() { return dat[0]; }
00080     const_reference front() const { return dat[0]; }
00081     reference back() { return dat[size() - 1]; }
00082     const_reference back() const { return dat[size() - 1]; }
00083
00084     value_type* data() noexcept { return dat; }
00085     const value_type* data() const noexcept { return dat; }
00086
00087     // Iterators
00088     iterator begin() noexcept { return dat; }
00089     const_iterator begin() const noexcept { return dat; }
```

```
00090     iterator end() noexcept { return avail; }
00091     const_iterator end() const noexcept { return avail; }
00092
00093     // Capacity
00094     bool empty() const noexcept { return size() == 0; }
00095     size_type capacity() const noexcept { return limit - dat; }
00096     size_type size() const noexcept { return avail - dat; }
00097     size_type max_size() const noexcept { return std::numeric_limits<size_type>::max(); }
00098     void reserve(size_type n) {
00099         if (n > capacity()) {
00100             reallocate(n);
00101         }
00102     }
00103
00104         void resize(size_type sz) {
00105         if (sz < size()) {
00106             destroy_elements(dat + sz, avail);
00107             avail = dat + sz;
00108         } else {
00109             if (sz > capacity()) {
00110                 reserve(sz);
00111             }
00112             std::uninitialized_fill(avail, dat + sz, T());
00113             avail = dat + sz;
00114         }
00115     }
00116
00117     void resize(size_type sz, const value_type& value) {
00118         if (sz < size()) {
00119             destroy_elements(dat + sz, avail);
00120             avail = dat + sz;
00121         } else {
00122             if (sz > capacity()) {
00123                 reserve(sz);
00124             }
00125             std::uninitialized_fill(avail, dat + sz, value);
00126             avail = dat + sz;
00127         }
00128     }
00129
00130     void shrink_to_fit() {
00131         if (limit > avail) {
00132             reallocate(size());
00133         }
00134     }
00135
00136     // Modifiers
00137     void clear() noexcept { uncreate(); }
00138
00139     void push_back(const value_type& val) {
00140         if (avail == limit) {
00141             grow();
00142         }
00143         alloc.construct(avail++, val);
00144     }
00145
00146     void push_back(value_type&& val) {
00147         if (avail == limit) {
00148             grow();
00149         }
00150         alloc.construct(avail++, std::move(val));
00151     }
00152
00153     void pop_back() {
00154         if (avail != dat) {
00155             alloc.destroy(--avail);
00156         }
00157     }
00158
00159     iterator insert(iterator position, const value_type& val) {
00160         return insert(position, 1, val);
00161     }
00162
00163     iterator insert(iterator position, size_type n, const value_type& val) {
00164         size_type index = position - begin();
00165         if (avail + n > limit) {
00166             grow(size() + n);
00167             position = begin() + index;
00168         }
00169         std::move_backward(position, avail, avail + n);
00170         std::uninitialized_fill(position, position + n, val);
00171         avail += n;
00172         return position;
00173     }
00174
00175     iterator erase(iterator position) {
00176         if (position < dat || position >= avail) {
```

```
00177             throw std::out_of_range("Index out of range");
00178         }
00179         std::move(position + 1, avail, position);
00180         alloc.destroy(--avail);
00181         return position;
00182     }
00183
00184     iterator erase(iterator first, iterator last) {
00185         if (first < dat || last > avail || first > last) {
00186             throw std::out_of_range("Index out of range");
00187         }
00188         iterator new_avail = std::move(last, avail, first);
00189         destroy_elements(new_avail, avail);
00190         avail = new_avail;
00191         return first;
00192     }
00193
00194 private:
00195     iterator dat = nullptr;
00196     iterator avail = nullptr;
00197     iterator limit = nullptr;
00198     std::allocator<T> alloc;
00199
00200     void create() noexcept { dat = avail = limit = nullptr; }
00201
00202     template <class InputIterator>
00203     void create(InputIterator first, InputIterator last) {
00204         dat = alloc.allocate(last - first);
00205         avail = limit = std::uninitialized_copy(first, last, dat);
00206     }
00207
00208     void uncreate() noexcept {
00209         if (dat) {
00210             destroy_elements(dat, avail);
00211             alloc.deallocate(dat, limit - dat);
00212         }
00213         dat = avail = limit = nullptr;
00214     }
00215
00216     void grow(size_type new_capacity = 1) {
00217         size_type new_size = std::max(new_capacity, 2 * capacity());
00218         reallocate(new_size);
00219     }
00220
00221     void reallocate(size_type new_size) {
00222         iterator new_data = alloc.allocate(new_size);
00223         iterator new_avail = std::uninitialized_copy(dat, avail, new_data);
00224         destroy_elements(dat, avail);
00225         alloc.deallocate(dat, limit - dat);
00226         dat = new_data;
00227         avail = new_avail;
00228         limit = dat + new_size;
00229     }
00230
00231     void destroy_elements(iterator first, iterator last) noexcept {
00232         while (first != last) {
00233             alloc.destroy(--last);
00234         }
00235     }
00236
00237     void move_from(Vector&& other) noexcept {
00238         dat = other.dat;
00239         avail = other.avail;
00240         limit = other.limit;
00241         other.dat = other.avail = other.limit = nullptr;
00242     }
00243
00244 };
00245
```

5.4 vektoriai.h

```
00001 #ifndef VEKTORIAI_H
00002 #define VEKTORIAI_H
00003
00004
00005 #include "vector.h"
00006 #include <iostream>
00007 #include <fstream>
00008 #include <string>
00009 #include <algorithm>
00010 #include <iomanip>
00011 #include <ctime>
```

```
00012 #include <cstdlib>
00013 #include <sstream>
00014 #include <limits>
00015 #include <numeric>
00016 #include <chrono>
00017 #include <cassert>
00018
00019
00020 using namespace std;
00021
00022 class Student;
00023 bool isValidName(const string& name);
00024 bool isValidGrade(const string& grade);
00025 double calculateAverage(const Student& student);
00026 double calculateMedian(const Student& student);
00027 void randomGradeGenerator(int number, Student& student);
00028 void generateNames(Student& student);
00029 void readFromFile(const string& filename, Vector<Student>& students);
00030 void tests();
00031 void checkVector();
00032 void vectorVsVector();
00033
00034 #endif
```

Index