MyVector

Generated by Doxygen 1.13.2

Chapter 1

Class Index

1.1 Class List

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

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Class Documentation

3.1 myVector< T > Class Template Reference

```
#include <myvector.h>
```

Public Member Functions

- myVector ()

  Creates an empty vector with default capacity of 10.
- myVector (const int &num, const T &val)

  Creates a vector with given size and fills it with given value

  .
- myVector (const T ∗begin, const T ∗end)

  Creates a vector with values from given range

  .
- myVector (const myVector &other)

  Creates a vector with values from given vector. Values in original vector are not moved

  .
- myVector (myVector &&other)

  Creates a vector with values from given vector and "moves" the values from it

  .
- myVector (std::initializer_list< T > list)

  Creates a vector with values from given initializer list

  .
- myVector operator= (std::initializer_list< T > list)

  Assigns values from given initializer list to the vector

  .
- myVector & operator= (const myVector &other)

  Assigns values from given vector to the vector. Values in original vector are not moved

  .
- myVector & operator= (myVector &&other)

  Assigns values from given vector to the vector and "moves" the values from it

  .
- ∼myVector ()

  Frees the memory and resets size_, capacity_, begin_ and end_.
- int size () const

Returns the size of the vector

.

- int capacity () const

  Returns the capacity of the vector

  .
- T ∗ begin ()

  Returns the pointer to the beginning of the vector

  .
- T ∗ cbegin () const

  Returns the pointer to the beginning of the vector, but as const

  .
- T ∗ end ()

  Returns the pointer to element past the end of the vector

  .
- T ∗ cend () const

  Returns the pointer to element past the end of the vector, but as const

  .
- T & front () const

  Returns the first element of the vector

  .
- T & back () const

  Returns the last element of the vector

  .
- T & operator[ ] (const int &index) const noexcept

  Returns the element at given index

  .
- T & at (const int &index) const

  Returns the element at given index (with bounds checking)

  .
- void push_back (const T &value)

  Adds an element to the end of the vector. If the vector is full, it doubles the capacity.

  .
- void pop_back ()

  Removes the last element of the vector.
- bool empty () const

  Checks if the vector is empty

  .
- void reserve (const int &num)

  Reserves memory for the vector. If the given number is smaller than the current capacity, it does nothing

  .
- void shrink_to_fit ()

  Shrinks the vector to fit the current size. If the capacity is equal to the size, it does nothing.
- void clear ()

  Clears the vector. Frees the old memory, but the capacity remains the same.
- std::wstring output () const

  Outputs the vector to a string (for testing purposes)

  .
- bool operator== (const myVector &second) const

  Compares two vectors. If the sizes are different, it returns false. If the sizes are the same, it compares the elements.

  .
- bool operator!= (const myVector &second) const

  Compares two vectors. If the sizes are different, it returns true. If the sizes are the same, it compares the elements.

  .
- void assign (const int &num, const T &val)

Replaces the elements with num copies of val value.

.

- void assign (const T ∗start, const T ∗end)

  Replaces the elements with elements from range from start to end

  .

- T ∗ data () const

  Returns pointer to the underlying array.

- void insert (const T ∗pos, const T &val)

  Inserts an copy of element val into place before position pos.

- void insert (const T ∗pos, const int num, const T &val)

  Inserts a num copies of element val into place before position pos.

3.1.1 Constructor & Destructor Documentation

3.1.1.1 myVector() [1/6]

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

Creates an empty vector with default capacity of 10.

3.1.1.2 myVector() [2/6]

```
template<typename T>
myVector< T >::myVector (
            const int & num,
            const T & val)  [inline]
```

Creates a vector with given size and fills it with given value

.

Parameters

num	- size of the vector
val	- value to fill the vector with

3.1.1.3 myVector() [3/6]

```
template<typename T>
myVector< T >::myVector (
            const T * begin,
            const T * end)  [inline]
```

Creates a vector with values from given range

.

Parameters

begin	- pointer to the beginning of the range
end	- pointer to the end of the range

3.1.1.4 myVector() [4/6]

```
template<typename T>
myVector< T >::myVector (
            const myVector< T > & other)  [inline]
```

Creates a vector with values from given vector. Values in original vector are not moved
.

Parameters

other	- vector to copy from

3.1.1.5 myVector() [5/6]

```
template<typename T>
myVector< T >::myVector (
            myVector< T > && other)  [inline]
```

Creates a vector with values from given vector and "moves" the values from it
.

Parameters

other	- vector to move from

3.1.1.6 myVector() [6/6]

```
template<typename T>
myVector< T >::myVector (
            std::initializer_list< T > list)  [inline]
```

Creates a vector with values from given initializer list
.

Parameters

list	- initializer list to copy from

3.1.1.7 ∼myVector()

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

Frees the memory and resets size_, capacity_, begin_ and end_.

3.1.2 Member Function Documentation

3.1.2.1 assign() [1/2]

```
template<typename T>
void myVector< T >::assign (
            const int & num,
            const T & val)  [inline]
```

Replaces the elements with num copies of val value.

.

Parameters

num	- number of elements to replace
val	- value to replace with

3.1.2.2 assign() [2/2]

```
template<typename T>
void myVector< T >::assign (
            const T * start,
            const T * end)  [inline]
```

Replaces the elements with elements from range from start to end

.

Parameters

start	- pointer to a start of a range
end	- pointer to an end of a range

3.1.2.3 at()

```
template<typename T>
T & myVector< T >::at (
            const int & index) const  [inline]
```

Returns the element at given index (with bounds checking)

.

Parameters

index	- index of the element

Returns

Element at given index (template typename T)

3.1.2.4 back()

```
template<typename T>
T & myVector< T >::back () const  [inline]
```

Returns the last element of the vector

.

Returns

Last element of the vector (template typename T)

3.1.2.5 begin()

```
template<typename T>
T * myVector< T >::begin ()  [inline]
```

Returns the pointer to the beginning of the vector

.

Returns

Pointer to the beginning of the vector (template typename T*)

3.1.2.6 capacity()

```
template<typename T>
int myVector< T >::capacity () const  [inline]
```

Returns the capacity of the vector

.

Returns

Capacity of vector (integer)

3.1.2.7 cbegin()

```
template<typename T>
T * myVector< T >::cbegin () const  [inline]
```

Returns the pointer to the beginning of the vector, but as const
.

Returns

Pointer to the beginning of the vector (template typename T*)

3.1.2.8 cend()

```
template<typename T>
T * myVector< T >::cend () const  [inline]
```

Returns the pointer to element past the end of the vector, but as const
.

Returns

Pointer to element past the end of the vector (template typename T*)

3.1.2.9 clear()

```
template<typename T>
void myVector< T >::clear ()  [inline]
```

Clears the vector. Frees the old memory, but the capacity remains the same.

3.1.2.10 data()

```
template<typename T>
T * myVector< T >::data () const  [inline]
```

Returns pointer to the underlying array.

Returns

Pointer to start of the data_ array

3.1.2.11 empty()

```
template<typename T>
bool myVector< T >::empty () const  [inline]
```

Checks if the vector is empty
.

Returns

True if the vector is empty, false otherwise (bool)

3.1.2.12 end()

```
template<typename T>
T * myVector< T >::end ( ) [inline]
```

Returns the pointer to element past the end of the vector

.

Returns

Pointer to element past the end of the vector (template typename T∗)

3.1.2.13 front()

```
template<typename T>
T & myVector< T >::front ( ) const [inline]
```

Returns the first element of the vector

.

Returns

First element of the vector (template typename T)

3.1.2.14 insert() [1/2]

```
template<typename T>
void myVector< T >::insert (
            const T * pos,
            const int num,
            const T & val) [inline]
```

Inserts a num copies of element val into place before position pos.

Parameters

pos	- pointer to position
num	- number of copies to insert
val	- value to insert

3.1.2.15 insert() [2/2]

```
template<typename T>
void myVector< T >::insert (
            const T * pos,
            const T & val) [inline]
```

Inserts an copy of element val into place before position pos.

Parameters

pos	- pointer to position
val	- value to insert

3.1.2.16 operator"!=()

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

Compares two vectors. If the sizes are different, it returns true. If the sizes are the same, it compares the elements.
.

Parameters

second	- vector to compare with

Returns

True if the vectors are not equal, false otherwise (bool)

3.1.2.17 operator=() [1/3]

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

Assigns values from given vector to the vector. Values in original vector are not moved
.

Parameters

other	- vector to copy from

Returns

the vector

See also

myVector(const myVector& other)

3.1.2.18 operator=() [2/3]

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

Assigns values from given vector to the vector and "moves" the values from it
.

Parameters

other	- vector to move from

Returns

the vector

See also

myVector(myVector&& other)

3.1.2.19 operator=() [3/3]

```
template<typename T>
myVector myVector< T >::operator= (
            std::initializer_list< T > list)  [inline]
```

Assigns values from given initializer list to the vector
.

Parameters

list	- initializer list to copy from

Returns

the vector

3.1.2.20 operator==()

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

Compares two vectors. If the sizes are different, it returns false. If the sizes are the same, it compares the elements.
.

Returns

True if the vectors are equal, false otherwise (bool)

3.1.2.21 operator[]()

```
template<typename T>
T & myVector< T >::operator[] (
            const int & index) const  [inline], [noexcept]
```

Returns the element at given index
.

Parameters

index	- index of the element

Returns

Element at given index (template typename T)

3.1.2.22 output()

```
template<typename T>
std::wstring myVector< T >::output () const  [inline]
```

Outputs the vector to a string (for testing purposes)
.

Returns

String with the vector values (std::wstring)

3.1.2.23 pop_back()

```
template<typename T>
void myVector< T >::pop_back ()  [inline]
```

Removes the last element of the vector.

3.1.2.24 push_back()

```
template<typename T>
void myVector< T >::push_back (
             const T & value)  [inline]
```

Adds an element to the end of the vector. If the vector is full, it doubles the capacity.
.

Parameters

value	- value to add

3.1.2.25 reserve()

```
template<typename T>
void myVector< T >::reserve (
             const int & num)  [inline]
```

Reserves memory for the vector. If the given number is smaller than the current capacity, it does nothing
.

Parameters

num	- number of elements to reserve memory for

3.1.2.26 shrink_to_fit()

```
template<typename T>
void myVector< T >::shrink_to_fit ()  [inline]
```

Shrinks the vector to fit the current size. If the capacity is equal to the size, it does nothing.

3.1.2.27 size()

```
template<typename T>
int myVector< T >::size () const  [inline]
```

Returns the size of the vector
.

Returns

Size of vector (integer)

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

- myvector.h

Chapter 4

File Documentation

4.1 myvector.h File Reference

```
#include <iostream>
#include <stdexcept>
#include <memory>
#include <string>
#include <algorithm>
#include <initializer_list>
```

Classes

- class myVector< T >

4.2 myvector.h

Go to the documentation of this file.

```
#pragma once
#include <iostream>
#include <stdexcept>
#include <memory>
#include <string>
#include <algorithm>
#include <initializer_list>

template<typename T>
class myVector {
private:
    int size_;
    int capacity_;
    T* data_;
    T* begin_;
    T* end_;
public:
    myVector() { // basic constructor
        size_ = 0;
        capacity_ = 10;
        data_ = new T[capacity_];
        begin_ = data_;
        end_ = data_;
    }
    myVector(const int& num, const T& val) { // fill constructor
        size_ = num;
        capacity_ = num;
        data_ = new T[capacity_];
        for (int i = 0; i < num; i++) {
            data_[i] = val;
        }
        begin_ = &data_[0];
        end_ = &data_[size_];
    }
    myVector(const T* begin, const T* end) { //range constructor
        size_ = end - begin;
        capacity_ = size_;
        data_ = new T[capacity_];
        for (int i = 0; i < size_; i++) {
            data_[i] = *(begin + i);
        }
        begin_ = &data_[0];
        end_ = &data_[size_];
    }
    myVector(const myVector& other) { // copy constructor
        size_ = other.size_;
        capacity_ = other.capacity_;
        data_ = new T[capacity_];
        for (int i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
        begin_ = &data_[0];
        end_ = &data_[size_];
    }
    myVector(myVector&& other) { //move constructor
        size_ = other.size_;
        capacity_ = other.capacity_;
        data_ = other.data_;
        begin_ = &data_[0];
        end_ = &data_[size_];
        other.data_ = nullptr;
        other.size_ = 0;
        other.capacity_ = 0;
        other.begin_ = nullptr;
        other.end_ = nullptr;
    }
    myVector(std::initializer_list<T> list) { //initalizer list constructor
        size_ = static_cast<int>(list.size());
        capacity_ = size_;
        data_ = new T[capacity_];
        std::copy(list.begin(), list.end(), data_);
        begin_ = &data_[0];
        end_ = &data_[size_];
    }
    myVector operator=(std::initializer_list<T> list) { // operator = with initalizer list
        this->~myVector();
        size_ = static_cast<int>(list.size());
        capacity_ = size_;
        data_ = new T[capacity_];
        std::copy(list.begin(), list.end(), data_);
        begin_ = &data_[0];
        end_ = &data_[size_];
        return *this;
    }
    myVector& operator= (const myVector& other) { // copy assignment
        if (this == &other) return *this;
        this->~myVector();
        size_ = other.size_;
        capacity_ = other.capacity_;
        data_ = new T[capacity_];
        for (int i = 0; i < size_; i++) {
            data_[i] = other.data_[i];
        }
        begin_ = &data_[0];
        end_ = &data_[size_];
        return *this;
    }
    myVector& operator= (myVector&& other) { // move assignment
        if (this == &other) return *this;
        this->~myVector();
        size_ = other.size_;
        capacity_ = other.capacity_;
        data_ = other.data_;
        begin_ = &data_[0];
        end_ = &data_[size_];
        other.data_ = nullptr;
        other.size_ = 0;
        other.capacity_ = 0;
        other.end_ = nullptr;
        other.begin_ = nullptr;
        return *this;
    }
    ~myVector() {
        size_ = 0;
        capacity_ = 0;
        delete[] data_;
        begin_ = nullptr;
        end_ = nullptr;
    }
    int size() const {
        return size_;
    }
    int capacity() const {
        return capacity_;
    }
    T* begin() {
        return begin_;
    }
    T* cbegin() const {
        return begin_;
    }
    T* end() {
        return end_;
    }
    T* cend() const {
        return end_;
    }
    T& front() const {
        if (this->empty()) throw std::out_of_range("Vector is empty");
        return *begin_;
    }
    T& back() const {
        if (this->empty()) throw std::out_of_range("Vector is empty");
        return *(end_ - 1);
    }
    T& operator[] (const int& index) const noexcept {
        //if (index < 0 || index >= size_) throw std::out_of_range("Out of range!");
        return this->data_[index];
    }
    T& at(const int& index) const {
        if (index < 0 || index >= size_) {
            throw std::out_of_range("Out of range!");
            std::wcerr « "Out of range!";
            std::terminate();
        }
        return data_[index];
    }
    void push_back(const T& value) {
        if (capacity_ == 0) {
            capacity_ = 10;
            data_ = new T[capacity_];
        }
        if (size_ == 0) {
            data_[0] = value;
            begin_ = &data_[0];
            end_ = &data_[0] + 1;
            size_ = 1;
        }
        else {
            if (size_ >= capacity_) {
                capacity_ = capacity_ * 2;
                T* temp = new T[capacity_];
                for (int i = 0; i < size_; i++) {
                    temp[i] = data_[i];
                }
                delete[] data_;
                data_ = temp;
                begin_ = &data_[0];
                end_ = &data_[size_];
                temp = nullptr;
            }
            data_[size_] = value;
            end_ = &data_[size_] + 1;
            size_++;

        }
    }
    void pop_back() {
        if (size_ == 0) return;
        ~data_[size_ - 1];
        size_ = size_-1;
        end_ = end_-1;
    }
    bool empty() const {
        if (begin_ == end_) return true;
        return false;
    }
    void reserve(const int& num) {
        if (num <= capacity_) return;
        capacity_ = num;
        T* temp = new T[capacity_];
        for (int i = 0; i < size_; i++) {
            temp[i] = data_[i];
        }
        delete[] data_;
        data_ = temp;
        begin_ = &data_[0];
        end_ = &data_[size_];
        temp = nullptr;
    }
    void shrink_to_fit() {
        if (capacity_ <= size_) return;
        capacity_ = size_;
        T* temp = new T[capacity_];
        for (int i = 0; i < size_; i++) {
            temp[i] = data_[i];
        }
        delete[] data_;
        data_ = temp;
        begin_ = &data_[0];
        end_ = &data_[size_];
        temp = nullptr;
    }
    void clear() {
       if (size_ == 0) return;
       delete[] data_;
       data_ = new T[capacity_];
       size_ = 0;
       begin_ = data_;
       end_ = begin_;
    }
    std::wstring output() const {
        std::wstringstream out;
        out « size_;
        if (size_ != 0) {
            out « *begin_ « *(end_ - 1);
            for (int i = 0; i < size_; i++) {
                out « data_[i];
            }
        }
        std::wstring str;
        str = out.str();
        return str;
    }
    bool operator==(const myVector& second) const {
        if (size_ != second.size_) return false;
        for (int i = 0; i < size_; i++) {
            if (data_[i] != second.data_[i]) return false;
        }
        return true;
    }
    bool operator!=(const myVector& second) const {
        if (size_ != second.size_) return true;
        for (int i = 0; i < size_; i++) {
            if (data_[i] != second.data_[i]) return true;
        }
        return false;
    }
    void assign(const int& num, const T& val) {
        if (num <= capacity_) {
            if (num > size_) {
                size_ = num;
                end_ = &data_[size_];
            }
            for (int i = 0; i < num; i++) {
                ~data_[i];
                data_[i] = val;
            }
        }
        else {
            this->~myVector();
            capacity_ = num;
            size_ = num;
            data_ = new T[capacity_];
            for (int i = 0; i < size_; i++) {
                data_[i] = val;
            }
            begin_ = &data_[0];
            end_ = &data_[size_];
        }
    }

    void assign(const T* start, const T* end) {
        int num = end - start;
        if (num <= capacity_) {
            if (num > size_) {
                size_ = num;
                end_ = &data_[size_];
            }
            for (int i = 0; i < num; i++) {
                ~data_[i];
                data_[i] = *(start+i);
            }
        }
        else {
            this->~myVector();
            capacity_ = num;
            size_ = num;
            data_ = new T[capacity_];
            for (int i = 0; i < size_; i++) {
                data_[i] = *(start + i);
            }
            begin_ = &data_[0];
            end_ = &data_[size_];
        }
    }
    T* data() const {
        return &data_[0];
    }
    void insert(const T* pos,  const T& val) {
        int index = pos - begin_;
        if (size_ + 1 <= capacity_) {
            for (auto it = end_; it != pos; it--) {
                *it = *(it - 1);
            }
            data_[index] = val;
            size_++;
            end_ = &data_[size_];
        }
        else {
            capacity_ = capacity_ * 2;
            T* temp = new T[capacity_];
            for (int i = 0; i != index; i++) {
                temp[i] = data_[i];
            }
            temp[index] = val;
            for (int i = index; i < size_; i++) {
                temp[i+1] = data_[i];
            }
            delete[] data_;
            data_ = temp;
            begin_ = data_;
            size_++;
            end_ = &data_[size_];
        }
    }
    void insert(const T* pos, const int num, const T& val) {
        int index = pos - begin_;
        if (size_ + num <= capacity_) {
            for (auto it = end_ + num - 1; it != pos; it--) {
                *it = *(it - num);
            }
            for (int i = num; i > 0; i--) {
                data_[index + i - 1] = val;
            }
            size_ = size_ + num;
            end_ = &data_[size_];
        }
        else {
            capacity_ = capacity_ + num;
            T* temp = new T[capacity_];
            for (int i = 0; i != index; i++) {
                temp[i] = data_[i];
            }
            for (int i = index; i < index + num; i++) {
                temp[i] = val;
            }
            for (int i = index; i < size_; i++) {
                temp[i + num] = data_[i];
            }
            delete[] data_;
            data_ = temp;
            begin_ = data_;
            size_ = size_ + num;
            end_ = &data_[size_];
        }
    }
};
```

Index