MSVector

2.0

Generated by Doxygen 1.9.5

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 documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 MSVector< T > Class Template Reference

MSVector Class.

```
#include <msVector.hpp>
```

Public Types

- typedef T $*$ **iterator**

Public Member Functions

- MSVector ()

  Construct a new MSVector object.
- MSVector (int)

  Construct a new MSVector object.
- MSVector (T $*$, int)

  Construct a new MSVector object from Pre-defined container.
- MSVector (const MSVector< T > &)

  Construct a new MSVector object from another object.
- MSVector< T > & operator= (const MSVector< T > &)

  Redefine MSVector object form another object.
- MSVector< T > & operator= (MSVector< T > &&)

  Move MSVector object to another object.
- ∼MSVector ()

  Destroy the MSVector object.
- T & operator[ ] (int)

  Get value at given index.
- iterator begin ()

  Get address of first element in MSVector.
- iterator end ()

  Get address of position after last element.
- int push_back (T)

Push Element at end of MSVector.
- T pop_back ()

  Remove last element form MSVector.
- void insert (iterator, T)

  Insert value at given Position.
- void erase (iterator)

  Erase element at given Position.
- void erase (iterator, iterator)

  Erase element from start till before end.
- int Size () const

  Get size of MSVector.
- int Capacity () const

  Get capacity of MSVector.
- int Resize ()

  Resize MSVector.
- bool Empty ()

  Check if MSVector is empty or not.
- void Clear ()

  Clear MSVector.
- bool operator== (const MSVector< T > &)

  Compare two MSVector element by element.
- bool operator< (const MSVector< T > &)

  Compare two MSVector element by element.

Friends

- ostream & **operator** (ostream &, MSVector< T >)

3.1.1 Detailed Description

template<class T>
class MSVector< T >

MSVector Class.

Template Parameters

T	

3.1.2 Constructor & Destructor Documentation

3.1.2.1 MSVector() `[1/4]`

```
template<class T >
MSVector< T >::MSVector
```

Construct a new MSVector object.

Template Parameters

T	

Returns

MSVector<T>

3.1.2.2 MSVector() `[2/4]`

```
template<class T >
MSVector< T >::MSVector (
              int cap )
```

Construct a new MSVector object.

Template Parameters

T	

Parameters

cap	

Returns

MSVector<T>

3.1.2.3 MSVector() `[3/4]`

```
template<class T >
MSVector< T >::MSVector (
              T * other,
              int n )
```

Construct a new MSVector object from Pre-defined container.

Template Parameters

T	

Parameters

other	

Parameters

n	

Returns

MSVector<T>

3.1.2.4 MSVector() [4/4]

```
template<class T >
MSVector< T >::MSVector (
             const MSVector< T > & other )
```

Construct a new MSVector object from another object.

Template Parameters

T	

Parameters

other	

Returns

MSVector<T>

3.1.2.5 ∼MSVector()

```
template<class T >
MSVector< T >::∼MSVector
```

Destroy the MSVector object.

Template Parameters

T	

Returns

MSVector<T>

3.1.3 Member Function Documentation

3.1.3.1 begin()

```
template<class T >
MSVector< T >::iterator MSVector< T >::begin
```

Get address of first element in MSVector.

Returns

iterator

Template Parameters

T	

Returns

MSVector<T>

3.1.3.2 Capacity()

```
template<class T >
int MSVector< T >::Capacity
```

Get capacity of MSVector.

Returns

int

Template Parameters

T	

Returns

int

3.1.3.3 Clear()

```
template<class T >
void MSVector< T >::Clear
```

Clear MSVector.

Clear all elements in MSVector.

Template Parameters

T	

3.1.3.4 Empty()

```
template<class T >
bool MSVector< T >::Empty
```

Check if MSVector is empty or not.

Returns

true

false

Template Parameters

T	

Returns

true

false

3.1.3.5 end()

```
template<class T >
MSVector< T >::iterator MSVector< T >::end
```

Get address of position after last element.

Returns

iterator

Template Parameters

T	

Returns

MSVector<T>

3.1.3.6 erase() [1/2]

```
template<class T >
void MSVector< T >::erase (
             iterator pos )
```

Erase element at given Position.

Template Parameters

T	

Parameters

pos	

3.1.3.7 erase() [2/2]

```
template<class T >
void MSVector< T >::erase (
             iterator start,
             iterator end )
```

Erase element from start till before end.

Template Parameters

T	

Parameters

start	
end	

3.1.3.8 insert()

```
template<class T >
void MSVector< T >::insert (
             iterator pos,
             T value )
```

Insert value at given Position.

insert value at given Position

Template Parameters

T	

Parameters

pos	
value	

3.1.3.9 operator<()

```
template<class T >
bool MSVector< T >::operator< (
             const MSVector< T > & other )
```

Compare two MSVector element by element.

Returns

true

false

Template Parameters

T	

Parameters

other	

Returns

true

false

3.1.3.10 operator=() [1/2]

```
template<class T >
MSVector< T > & MSVector< T >::operator= (
            const MSVector< T > & other )
```

Redefine MSVector object form another object.

Returns

MSVector<T>&

Template Parameters

T	

Parameters

other	

Returns

MSVector<T>&

3.1.3.11 operator=() [2/2]

```
template<class T >
MSVector< T > & MSVector< T >::operator= (
            MSVector< T > && other )
```

Move MSVector object to another object.

Returns

MSVector<T>&

Template Parameters

T	

Parameters

other	

Returns

MSVector<T>&

3.1.3.12 operator==()

```
template<class T >
bool MSVector< T >::operator== (
             const MSVector< T > & other )
```

Compare two MSVector element by element.

Returns

true

false

Template Parameters

T	

Parameters

other	

Returns

true

false

3.1.3.13 operator[]()

```
template<class T >
T & MSVector< T >::operator[] (
             int indx )
```

Get value at given index.

Returns

T&

Template Parameters

T	

Parameters

indx	

Returns

T&

3.1.3.14 pop_back()

```
template<class T >
T MSVector< T >::pop_back
```

Remove last element form MSVector.

Returns

T

Template Parameters

T	

Returns

T

3.1.3.15 push_back()

```
template<class T >
int MSVector< T >::push_back (
             T item )
```

Push Element at end of MSVector.

Returns

int

Template Parameters

T	

Parameters

item	

Returns

int

3.1.3.16 Resize()

```
template<class T >
int MSVector< T >::Resize
```

Resize MSVector.

Returns

int

Template Parameters

T	

Returns

int

3.1.3.17 Size()

```
template<class T >
int MSVector< T >::Size
```

Get size of MSVector.

Returns

int

Template Parameters

T	

Returns

int

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

- msVector.hpp

Chapter 4

File Documentation

4.1 msVector.hpp File Reference

This is the Header file For msVector.

```
#include <iostream>
```

Classes

- class MSVector< T >

  MSVector Class.

Functions

- template<class T >
  ostream & operator<< (ostream &out, MSVector< T > vec)

  Print MSVector element by element in console.

4.1.1 Detailed Description

This is the Header file For msVector.

Authors

Mohamed Amgad and Seif Yahia

Version

2.0

Date

2022-12-22

4.1.2 Function Documentation

4.1.2.1 operator<<()

```
template<class T >
ostream & operator<< (
             ostream & out,
             MSVector< T > vec )
```

Print MSVector element by element in console.

Template Parameters

T	

Parameters

out	
vec	

Returns

ostream&

4.2 msVector.hpp

Go to the documentation of this file.

```
#ifndef _MSVECTOR_HPP
#define _MSVECTOR_HPP
#include <iostream>
using namespace std;

template <class T> class MSVector;
template <class T> ostream& operator«(ostream&, MSVector<T>);
template <class T>
class MSVector{
// Friends
    // Overload the exertion operator to print all vector content
    friend ostream& operator« <T> (ostream&, MSVector<T>);
// Private data members
private:
    int size, capacity;
    T* data;
// Public methods
public:

    typedef T* iterator;

    MSVector();

    MSVector(int);

    MSVector(T*, int);

    MSVector(const MSVector<T>&);

    MSVector<T>& operator=(const MSVector<T>&);

    MSVector<T>& operator=(MSVector<T>&&);

    ~MSVector();

    T& operator[](int);

    iterator begin();

    iterator end();

    int push_back(T);

    T pop_back();

    void insert(iterator, T);

    void erase(iterator);
    void erase(iterator,iterator);

    int Size() const;
    int Capacity() const;

    int Resize();

    bool Empty();

    void Clear();

    bool operator==(const MSVector<T>&);

    bool operator<(const MSVector<T>&);
};

template <class T>
MSVector<T> :: MSVector() {
    size = 0;
    capacity = 2;
    data = new T[capacity];
}

template <class T>
MSVector<T> :: MSVector(int cap) {
    size = 0;
    capacity = cap;
    data = new T[capacity];
}

template <class T>
MSVector<T> :: MSVector(T* other, int n) {
    size = n;
    capacity = n;
    data = new T[capacity];
    for(int i = 0; i < size; i++){
        data[i] = other[i];
    }
}

template <class T>
MSVector<T> :: MSVector(const MSVector<T>& other) {
    size = other.size;
    capacity = other.capacity;
    data = new T[capacity];
    for(int i = 0; i < other.size; i++){
        data[i] = other.data[i];
    }
}

template <class T>
MSVector<T>& MSVector<T> :: operator= (const MSVector<T>& other) {
    // Check if not self
    if(this != &other){
        delete[] data;
        size = other.size;
        capacity = other.capacity;
        data = new T[capacity];
        for(int i = 0; i < size; i++){
            data[i] = other.data[i];
        }
    }
    else
        cerr « "Cannot copy self\n";
    return *this;
}

template <class T>
MSVector<T>& MSVector<T> :: operator= (MSVector<T>&& other) {
    if(this != &other){
        size = other.size;
        capacity = other.capacity;
        data = other.data;
        other.data = nullptr;
        other.size = 0;
        other.capacity = 0;
    }
    else
        cerr « "Cannot move self\n";
    return *this;
}

template <class T>
MSVector<T> :: ~MSVector() { delete[] data; }

template <class T>
typename MSVector<T> :: iterator MSVector<T> :: begin(){ return data; }

template <class T>
typename MSVector<T> :: iterator MSVector<T> :: end() { return &data[size]; }

template <class T>
T& MSVector<T> :: operator[] (int indx) {
    try{
        if(indx > size - 1 or indx < 0)
            throw out_of_range("\n\tException Error: Index is OUT of range!\n");
    } catch(out_of_range error){
        cerr « error.what(); exit(-1);
    }
    return data[indx];
}

template <class T>
int MSVector<T> :: push_back(T item) {
    // Check if the vector is full and if so
    // resize to double the capacity
    if(size >= capacity){ this->Resize(); }
    data[size++] = item;
    return size;
}

template <class T>
T MSVector<T> :: pop_back() {
    try{
        if(this->Empty()){
            throw "\n\tException Error: Vector is already EMPTY!\n";
        }
    } catch(const char* error){ cerr « error; exit(-1); }
    return data[--size];
}

template <class T>
void MSVector<T> :: insert(iterator pos, T value) {
    try{
        if(pos > MSVector<T>::end() || pos < MSVector<T>::begin())
            throw out_of_range("\n\tException Error: Position is out of range!\n");
    } catch(out_of_range error){
        cerr « error.what(); exit(-1);
    }
    if(pos == MSVector<T>::end()){
        this->push_back(value);
    }
    else {
        if(size + 1 > capacity)
            capacity *= 2;
        int idx = 0;
        T *newData = new T[capacity];
        for (auto it = MSVector<T>::begin(); it != MSVector<T>::end();) {
            if (it != pos)
                newData[idx++] = *(it++);
            else {
                newData[idx++] = value;
                newData[idx++] = *(it++);
            }
        }
        delete[] data;
        size++;
        data = newData;
        newData = nullptr;
    }
}


template <class T>
void MSVector<T> :: erase(iterator pos) {
    try{
        if(pos < MSVector<T>::begin() || pos >= MSVector<T>::end())
            throw out_of_range("\n\tException Error: Position is out of range!\n");
    } catch(out_of_range error){
        cerr « error.what(); exit(-1);
    }
    int idx = 0;
    T* newData = new T[capacity];
    for (auto it = MSVector<T>::begin(); it != MSVector<T>::end(); ++it) {
        if(it != pos)
            newData[idx++] = *it;
    }
    delete [] data;
    size--;
    data = newData;
    newData = nullptr;
}

template <class T>
void MSVector<T> :: erase(iterator start, iterator end) {
    try{
        if((start < MSVector<T>::begin() || start > MSVector<T>::end())
        && (end < MSVector<T>::begin() || end > MSVector<T>::end())){
            throw out_of_range("\n\tException Error: Position is out of range!\n");
        }
    } catch(out_of_range error){
        cerr « error.what(); exit(-1);
    }
    try {
        if (start > end)
            throw "Position of start is greater than Position of end";
    }
    catch (const char* e){
        cerr « "Program Terminated due to: " « e;
        exit(-1);
    }
    int idx = 0;
    int LB = start - MSVector<T>::begin();
    int UB = end - MSVector<T>::begin() - 1;
    int newSize = size - (UB - LB + 1);
    if(newSize <= 0){
        MSVector<T>::Clear();
        return;
    }
    T* newData = new T[capacity];
    for (auto it = MSVector<T>::begin(); it != MSVector<T>::end(); ++it) {
        if(it < start || it > end - 1)
            newData[idx++] = *it;
    }
    delete [] data;
    size = newSize;
    data = newData;
    newData = nullptr;
}

template <class T>
int MSVector<T> :: Size() const { return size; }

template <class T>
int MSVector<T> :: Capacity() const { return capacity; }

template <class T>
int MSVector<T> :: Resize() {
    // If the vector is not full return 0
    if(size < capacity){ return capacity; }
    // Otherwise double the capacity
    capacity *= 2;
    T* newData = new T[capacity];
    // Copy the old vector in a new one
    // with double the capacity
    for(int i = 0; i < size; ++i){
        newData[i] = data[i];
    }
    delete [] data; // Delete the old vector
    data = newData; // Copy the new to the old
    return capacity;
}

template <class T>
bool MSVector<T> :: Empty() {
    if(size == 0) return true;
    return false;
}

template <class T>
void MSVector<T> :: Clear() {
    size = 0;
    delete [] data;
    data = nullptr;
}

```

```
template <class T>
bool MSVector<T> :: operator== (const MSVector<T> &other) {
    if(this->Size() != other.Size())
        return false;
    else{
        for (int i = 0; i < size; ++i) {
            if(this->data[i] != other.data[i])
                return false;
        }
    }
    return true;
}

template <class T>
bool MSVector<T> :: operator< (const MSVector<T> &other) {
    if(this->Size() != other.Size())
        return false;
    else{
        for (int i = 0; i < size; ++i) {
            if(this->data[i] >= other.data[i])
                return false;
        }
    }
    return true;
}

template <class T>
ostream& operator« (ostream& out, MSVector<T> vec) {
    if (!vec.Empty()){
        out « "\t==> ";
        for (int i = 0; i < vec.Size(); ++i)
            out « vec[i] « " ";
        out « "<==";
    }
    out « "\n";
    return out;
}

#endif
```

Index