

MSVector

2.0

Generated by Doxygen 1.9.5

1 Class Index	1
1.1 Class List	1
2 File Index	3
2.1 File List	3
3 Class Documentation	5
3.1 MSVector< T > Class Template Reference	5
3.1.1 Detailed Description	6
3.1.2 Constructor & Destructor Documentation	6
3.1.2.1 MSVector() [1/4]	6
3.1.2.2 MSVector() [2/4]	8
3.1.2.3 MSVector() [3/4]	8
3.1.2.4 MSVector() [4/4]	9
3.1.2.5 ~MSVector()	9
3.1.3 Member Function Documentation	10
3.1.3.1 begin()	10
3.1.3.2 Capacity()	10
3.1.3.3 Clear()	11
3.1.3.4 Empty()	11
3.1.3.5 end()	11
3.1.3.6 erase() [1/2]	12
3.1.3.7 erase() [2/2]	12
3.1.3.8 insert()	13
3.1.3.9 operator<()	13
3.1.3.10 operator=() [1/2]	14
3.1.3.11 operator=() [2/2]	14
3.1.3.12 operator==()	15
3.1.3.13 operator[]()	15
3.1.3.14 pop_back()	16
3.1.3.15 push_back()	16
3.1.3.16 Resize()	17
3.1.3.17 Size()	17
4 File Documentation	19
4.1 msVector.hpp File Reference	19
4.1.1 Detailed Description	19
4.1.2 Function Documentation	20
4.1.2.1 operator<<()	20
4.2 msVector.hpp	20
Index	25

Chapter 1

Class Index

1.1 Class List

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

MSVector< T >	
MSVector Class	5

Chapter 2

File Index

2.1 File List

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

msVector.hpp	
This is the Header file For msVector	19

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

<i>T</i>	
----------	--

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

<i>T</i>	
----------	--

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

<i>T</i>	
----------	--

Parameters

<i>cap</i>	
------------	--

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

<i>T</i>	
----------	--

Parameters

<i>other</i>	
--------------	--

Parameters

<i>n</i>	
----------	--

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

<i>T</i>	
----------	--

Parameters

<i>other</i>	
--------------	--

Returns

MSVector<T>

3.1.2.5 ~MSVector()

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

Destroy the [MSVector](#) object.

Template Parameters

<i>T</i>	
----------	--

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

<i>T</i>	
----------	--

Returns

MSVector<T>

3.1.3.2 Capacity()

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

Get capacity of [MSVector](#).

Returns

int

Template Parameters

<i>T</i>	
----------	--

Returns

int

3.1.3.3 Clear()

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

Clear [MSVector](#).

Clear all elements in [MSVector](#).

Template Parameters

<i>T</i>	
----------	--

3.1.3.4 Empty()

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

Check if [MSVector](#) is empty or not.

Returns

true
false

Template Parameters

<i>T</i>	
----------	--

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

<i>T</i>	
----------	--

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

<i>T</i>	
----------	--

Parameters

<i>pos</i>	
------------	--

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

<i>T</i>	
----------	--

Parameters

<i>start</i>	
<i>end</i>	

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

<i>T</i>	
----------	--

Parameters

<i>pos</i>	
<i>value</i>	

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

<i>T</i>	
----------	--

Parameters

<i>other</i>	
--------------	--

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

<i>T</i>	
----------	--

Parameters

<i>other</i>	
--------------	--

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

<i>T</i>	
----------	--

Parameters

<i>other</i>	
--------------	--

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

<i>T</i>	
----------	--

Parameters

<i>other</i>	
--------------	--

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

<i>T</i>	
----------	--

Parameters

<i>indx</i>	
-------------	--

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

<i>T</i>	
----------	--

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

<i>T</i>	
----------	--

Parameters

<i>item</i>	
-------------	--

Returns

int

3.1.3.16 Resize()

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

Resize [MSVector](#).

Returns

int

Template Parameters

<i>T</i>	
----------	--

Returns

int

3.1.3.17 Size()

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

Get size of [MSVector](#).

Returns

int

Template Parameters

<i>T</i>	
----------	--

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

<i>T</i>	
----------	--

Parameters

<i>out</i>	
<i>vec</i>	

Returns

ostream&

4.2 msVector.hpp

[Go to the documentation of this file.](#)

```
1 #ifndef _MSVECTOR_HPP
2 #define _MSVECTOR_HPP
10 #include <iostream>
11 using namespace std;
12
13 template <class T> class MSVector;
14 template <class T> ostream& operator<<(ostream&, MSVector<T>);
21 template <class T>
22 class MSVector{
23 // Friends
24 // Overload the exertion operator to print all vector content
25 friend ostream& operator<< <T> (ostream&, MSVector<T>);
26 // Private data members
27 private:
28     int size, capacity;
29     T* data;
30 // Public methods
31 public:
32
33     typedef T* iterator;
34
35     MSVector();
36
37     MSVector(int);
38
39     MSVector(T*, int);
40
41     MSVector(const MSVector<T>&);
42
43     MSVector<T>& operator=(const MSVector<T>&);
44
45     MSVector<T>& operator=(MSVector<T>&&);
72
```



```

77     ~MSVector();
78
84     T& operator[](int);
85
91     iterator begin();
92
98     iterator end();
99
105    int push_back(T);
106
112    T pop_back();
113
118    void insert(iterator, T);
119
124    void erase(iterator);
129    void erase(iterator, iterator);
130
136    int Size() const;
142    int Capacity() const;
143
149    int Resize();
150
157    bool Empty();
158
163    void Clear();
164
171    bool operator==(const MSVector<T>&);
172
179    bool operator<(const MSVector<T>&);
180 };
181
188 template <class T>
189 MSVector<T> :: MSVector() {
190     size = 0;
191     capacity = 2;
192     data = new T[capacity];
193 }
194
202 template <class T>
203 MSVector<T> :: MSVector(int cap) {
204     size = 0;
205     capacity = cap;
206     data = new T[capacity];
207 }
208
217 template <class T>
218 MSVector<T> :: MSVector(T* other, int n) {
219     size = n;
220     capacity = n;
221     data = new T[capacity];
222     for(int i = 0; i < size; i++){
223         data[i] = other[i];
224     }
225 }
226
234 template <class T>
235 MSVector<T> :: MSVector(const MSVector<T>& other) {
236     size = other.size;
237     capacity = other.capacity;
238     data = new T[capacity];
239     for(int i = 0; i < other.size; i++){
240         data[i] = other.data[i];
241     }
242 }
243
251 template <class T>
252 MSVector<T>& MSVector<T> :: operator= (const MSVector<T>& other) {
253     // Check if not self
254     if(this != &other){
255         delete[] data;
256         size = other.size;
257         capacity = other.capacity;
258         data = new T[capacity];
259         for(int i = 0; i < size; i++){
260             data[i] = other.data[i];
261         }
262     }
263     else
264         cerr << "Cannot copy self\n";
265     return *this;
266 }
267
275 template <class T>
276 MSVector<T>& MSVector<T> :: operator= (MSVector<T>&& other) {
277     if(this != &other){
278         size = other.size;
279         capacity = other.capacity;

```

```

280         data = other.data;
281         other.data = nullptr;
282         other.size = 0;
283         other.capacity = 0;
284     }
285     else
286         cerr << "Cannot move self\n";
287     return *this;
288 }
289
290 template <class T>
291 MSVector<T>::~MSVector() { delete[] data; }
292
293 template <class T>
294 typename MSVector<T>::iterator MSVector<T>::begin() { return data; }
295
296 template <class T>
297 typename MSVector<T>::iterator MSVector<T>::end() { return &data[size]; }
298
299 template <class T>
300 T& MSVector<T>::operator[] (int indx) {
301     try{
302         if(indx > size - 1 or indx < 0)
303             throw out_of_range("\n\tException Error: Index is OUT of range!\n");
304     } catch(out_of_range error){
305         cerr << error.what(); exit(-1);
306     }
307     return data[indx];
308 }
309
310 template <class T>
311 int MSVector<T>::push_back(T item) {
312     // Check if the vector is full and if so
313     // resize to double the capacity
314     if(size >= capacity){ this->Resize(); }
315     data[size++] = item;
316     return size;
317 }
318
319 template <class T>
320 T MSVector<T>::pop_back() {
321     try{
322         if(this->Empty()){
323             throw "\n\tException Error: Vector is already EMPTY!\n";
324         }
325     } catch(const char* error){ cerr << error; exit(-1); }
326     return data[--size];
327 }
328
329 template <class T>
330 void MSVector<T>::insert(iterator pos, T value) {
331     try{
332         if(pos > MSVector<T>::end() || pos < MSVector<T>::begin())
333             throw out_of_range("\n\tException Error: Position is out of range!\n");
334     } catch(out_of_range error){
335         cerr << error.what(); exit(-1);
336     }
337     if(pos == MSVector<T>::end()){
338         this->push_back(value);
339     }
340     else {
341         if(size + 1 > capacity)
342             capacity *= 2;
343         int idx = 0;
344         T *newData = new T[capacity];
345         for (auto it = MSVector<T>::begin(); it != MSVector<T>::end(); ) {
346             if (it != pos)
347                 newData[idx++] = *(it++);
348             else {
349                 newData[idx++] = value;
350                 newData[idx++] = *(it++);
351             }
352         }
353         delete[] data;
354         size++;
355         data = newData;
356         newData = nullptr;
357     }
358 }
359
360 template <class T>
361 void MSVector<T>::erase(iterator pos) {
362     try{
363         if(pos < MSVector<T>::begin() || pos >= MSVector<T>::end())
364             throw out_of_range("\n\tException Error: Position is out of range!\n");
365     } catch(out_of_range error){
366

```

```

418         cerr << error.what(); exit(-1);
419     }
420     int idx = 0;
421     T* newData = new T[capacity];
422     for (auto it = MSVector<T>::begin(); it != MSVector<T>::end(); ++it) {
423         if(it != pos)
424             newData[idx++] = *it;
425     }
426     delete [] data;
427     size--;
428     data = newData;
429     newData = nullptr;
430 }
431
432 template <class T>
433 void MSVector<T> :: erase(iterator start, iterator end) {
434     try{
435         if((start < MSVector<T>::begin() || start > MSVector<T>::end())
436            && (end < MSVector<T>::begin() || end > MSVector<T>::end())){
437             throw out_of_range("\n\tException Error: Position is out of range!\n");
438         }
439     } catch(out_of_range error){
440         cerr << error.what(); exit(-1);
441     }
442     try {
443         if (start > end)
444             throw "Position of start is greater than Position of end";
445     }
446     catch (const char* e){
447         cerr << "Program Terminated due to: " << e;
448         exit(-1);
449     }
450     int idx = 0;
451     int LB = start - MSVector<T>::begin();
452     int UB = end - MSVector<T>::begin() - 1;
453     int newSize = size - (UB - LB + 1);
454     if(newSize <= 0){
455         MSVector<T>::Clear();
456         return;
457     }
458     T* newData = new T[capacity];
459     for (auto it = MSVector<T>::begin(); it != MSVector<T>::end(); ++it) {
460         if(it < start || it > end - 1)
461             newData[idx++] = *it;
462     }
463     delete [] data;
464     size = newSize;
465     data = newData;
466     newData = nullptr;
467 }
468
469 template <class T>
470 int MSVector<T> :: Size() const { return size; }
471
472 template <class T>
473 int MSVector<T> :: Capacity() const { return capacity; }
474
475 template <class T>
476 int MSVector<T> :: Resize() {
477     // If the vector is not full return 0
478     if(size < capacity){ return capacity; }
479     // Otherwise double the capacity
480     capacity *= 2;
481     T* newData = new T[capacity];
482     // Copy the old vector in a new one
483     // with double the capacity
484     for(int i = 0; i < size; ++i){
485         newData[i] = data[i];
486     }
487     delete [] data; // Delete the old vector
488     data = newData; // Copy the new to the old
489     return capacity;
490 }
491
492 template <class T>
493 bool MSVector<T> :: Empty() {
494     if(size == 0) return true;
495     return false;
496 }
497
498 template <class T>
499 void MSVector<T> :: Clear() {
500     size = 0;
501     delete [] data;
502     data = nullptr;
503 }
504
505
506

```

```
550 template <class T>
551 bool MSVector<T> :: operator== (const MSVector<T> &other) {
552     if(this->Size() != other.Size())
553         return false;
554     else{
555         for (int i = 0; i < size; ++i) {
556             if(this->data[i] != other.data[i])
557                 return false;
558         }
559     }
560     return true;
561 }
562
563 template <class T>
564 bool MSVector<T> :: operator< (const MSVector<T> &other) {
565     if(this->Size() != other.Size())
566         return false;
567     else{
568         for (int i = 0; i < size; ++i) {
569             if(this->data[i] >= other.data[i])
570                 return false;
571         }
572     }
573     return true;
574 }
575
576 template <class T>
577 ostream& operator<< (ostream& out, MSVector<T> vec) {
578     if (!vec.Empty()){
579         out << "\t==> ";
580         for (int i = 0; i < vec.Size(); ++i)
581             out << vec[i] << " ";
582         out << "<=";
583     }
584     out << "\n";
585     return out;
586 }
587
588 #endif
```

Index

- ~MSVector
 - MSVector< T >, [9](#)
- begin
 - MSVector< T >, [10](#)
- Capacity
 - MSVector< T >, [10](#)
- Clear
 - MSVector< T >, [10](#)
- Empty
 - MSVector< T >, [11](#)
- end
 - MSVector< T >, [11](#)
- erase
 - MSVector< T >, [12](#)
- insert
 - MSVector< T >, [12](#)
- MSVector
 - MSVector< T >, [6](#), [8](#), [9](#)
- MSVector< T >, [5](#)
 - ~MSVector, [9](#)
 - begin, [10](#)
 - Capacity, [10](#)
 - Clear, [10](#)
 - Empty, [11](#)
 - end, [11](#)
 - erase, [12](#)
 - insert, [12](#)
 - MSVector, [6](#), [8](#), [9](#)
 - operator<, [13](#)
 - operator=, [13](#), [14](#)
 - operator==, [15](#)
 - operator[], [15](#)
 - pop_back, [16](#)
 - push_back, [16](#)
 - Resize, [17](#)
 - Size, [17](#)
- msVector.hpp, [19](#)
 - operator<<, [20](#)
- operator<
 - MSVector< T >, [13](#)
- operator<<
 - msVector.hpp, [20](#)
- operator=
 - MSVector< T >, [13](#), [14](#)
- operator==
 - MSVector< T >, [15](#)
- operator[]
 - MSVector< T >, [15](#)
- pop_back
 - MSVector< T >, [16](#)
- push_back
 - MSVector< T >, [16](#)
- Resize
 - MSVector< T >, [17](#)
- Size
 - MSVector< T >, [17](#)