The SST Common Object Library Manual

Gunar Schorcht
Technical University of Ilmenau
Department for Computer Science and Automation
Gunar@TheoInf.TU-Ilmenau.DE

Version 0.2

Technical Report SST-01/96

12. March 1996

1 Introduction

Programming either library or application software requires a common library with functionality for general tasks. In the object-oriented programming domain a class library is needed. Such a library should comprise class definitions that allow the instantiation of objects that satisfy requirements for such general tasks like

- object persistence handling,
- inter-object communication, and
- efficient data handling, e.g., by container classes.

Furthermore it is necessary to design the classes of general task object libraries so that specialized application dependent classes can be derived in an easy way. A lot of work was spend to develop a standard object library. As the result the STL - Standard Template Library [1] was proposed. It is a very comprehensive template library and provides a set of generic components. It uses function templates [3] extensively to realize generic operations on type parameterized container objects. Most parts of the STL were declared as the future standard for the C++ standard library [2]. The most important property of these libraries is the genericity, that is, defined classes and algorithms have type parameters. The problem of the STL as well as the future C++ standard library is that they are very complex. You have to use many template classes to use one single class. Furthermore the high sophisticated template class definitions cannot be used with any of todays compilers. The separation between generic algorithms and generic class definitions through their orthogonal design breaks down the consequent object-oriented programming methodology. Moreover, even such libraries realize a lot of classes for general tasks there is a lack of classes whose use is essential for our daily programming work. Therefore we have decided to develop a very small common object library, that contains only the most common classes, especially only those classes that are often used for our software. The library can either be used in extension to the C++ standard library or exclusively.

2 The Class Hierarchy

Since the library defines only general classes, the hierarchy shown in Fig. 1 is very flat. There are no more than two hierarchy levels. The most common class is the class *SstObjectC*. It provides instanced objects with a name and the facility to communicate with other objects by their names. Furthermore, it defines methods for persistence handling for derived classes. The second common class is the class *SstVectorC*. It is a template class and defines the handling of data objects with same type within an classical array. From this class the more specialized classes SstArrayC and SstMatrixC are derived. They are also generic, that is, the handled type is given as class parameter. The class *SstArrayC* realizes dynamic array objects based on the common generic vector object class. The matrix object class *SstMatrixC* realizes vector objects in two dimensions. The more specialized class SstNMatrixC realizes matrices for numerical data objects.

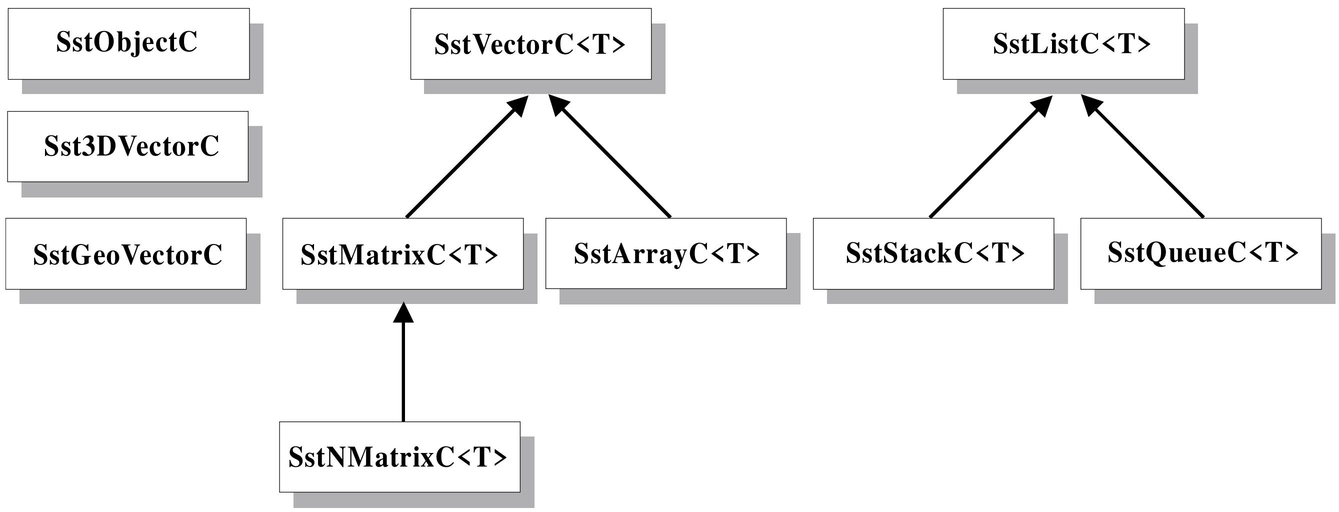


Fig. 1: SST common object library class hierarchy

The third common class *SstListC* realizes a generic double linked list. The type of data objects is given as a parameter during the instantiation. From this class the more specialized classes *SstStackC* and *SstQueueC* are derived. They realizes LIFO and FIFO lists respectively by using methods of basis class. To avoid the access to basis class methods, they are derived protected from class *SstListC*. The other classes *Sst3DVectorC* and *SstGeoVectorC* are often used specialized vector object classes.

3 Library Classes

3.1 Notation Rules

Identifiers used in the class library underlay the following rules.

- Class names have the prefix ***Sst***.
- Data members have the prefix ***m*** with the exception of public members. In this case the data member have not to have a special prefix.
- Formal parameters of member functions have the prefix ***p*** in most cases.
- Global variables carry a ***s*** as prefix.
- Identifier with more than one character are starting with uppercase letters.
- Underscore characters are not allowed within an identifier.

3.2 Class *SstObjectC*

This class provides objects of derived classes with common data elements and methods that are needed by each object within an application. The main facilities of the class *SstObjectC* are

- name handling,
- error handling,
- persistence handling, and
- inter-object communication.

The idea behind this class is, that each object of this or an derived class have a unique name. The object can be identified and addressed by its name. So the objects can interact by addressing each other by their names. ...

3.2.1 Class Declaration

```
#include <SST/Object.h.

class SstObjectC
{
    char *mName;        // default NULL

  public:

    SstObjectC(char *pName = NULL);
    SstObjectC(char *pName, int pNum);

    virtual ~SstObjectC();

    bool SetName(char *pName);                // TRUE -> success
    bool SetName(char *pName, int pNum);      // TRUE -> success

    virtual char *GetProject(){ return "SST";    }
    virtual char *GetClass  (){ return "SstObjectC"; }
    virtual char *GetObject (){ return GetName(); }
    virtual char *GetName   (){ if (mName) return mName; else return "NoName";}

    virtual void printf (FILE *pFile, char *pFormat, ...);
    virtual void eprintf(FILE *pFile, char *pFormat, ...);

    virtual ostream &operator << (ostream& os) { os << GetName(); return os;}
    virtual istream &operator >> (istream& is) { return is; }
};
```

3.2.2 Important Data Members

*char *mName;*

Description: This member holds the pointer to the memory that contains the name of the object. The memory for the name is allocated during the construction and the setting of the name, respectively.

Values: String

3.2.3 Usable Methods/Operators

SstObjectC (char *pName = NULL);
SstObjectC (char *pName, int pNum);

Description: The constructors use the methods *SstObjectC::SetName* to allocate the needed memory and to copy the given name *pName* to this memory. If a number is given by using parameter *pNum*, the name is extended by this number in ASCII format. By means of these numbers a series of equally named objects can be realized that are distinguished by their number extension.

Parameters: *pName* - object name
pNum - number by which the name is extended

Results: None

virtual ~SstObjectC ();

Description: The destructor releases the allocated memory.
Parameters: None
Results: None

bool SetName (char *pName);
bool SetName (char *pName, int pNum);

Description: This methods set the name of the object. The name is given by the parameter *pName*. Additionally, the name can be extended by a number given by the parameter *pNum*. The methods reallocate the needed memory dependent on the length of name and the numbering extension. After this, the method copies the whole name to the allocated memory.

Parameters: pName - the name of the object
pNum - the number by which the name should be extended

Results: *TRUE* - on success
FALSE - on error

virtual char *GetProject ()
virtual char *GetClass ()
virtual char *GetObject ()
virtual char *GetName ()

Description: These methods deliver several information about an instanced object

- the *Project* to which belongs the objects class definition,
- the *Class* from that the object is an instance, and
- the *Object* name

These methods can be used to identify the object in an unique way. In derived classes these methods have to be overwritten. The method *SstObjectC::GetName* is the same as *SstObjectC::GetObject*. An object is fully identified by *Project.Class.Object*.

Parameters: None

Results: *Project* - by default „SST“
Class - by default „SstObjectC“

Object - by default „NoName“

virtual void printf (FILE *pFile, char *pFormat, ...);

Description: This method realizes the C *printf* function with the exception that the object is identified automatically by using the methods *SstObjectC::GetProject*, *SstObjectC::GetClass*, and *SstObjectC::GetObject*. The format of the printed message is „Proj Xxx Class Yyyy, Obj Zzzz:\n\t ....“. By default (when the parameter *pFile* is *NULL*) the messages are written to the file given by the variable *sOutFile*, see below.

Parameters: *pFile* - file descriptor of the destination
pFormat- the printf format string
... - parameters in accordance to the format string.

Result None

virtual void eprintf (FILE *pFile, char *pFormat, ...);

Description: This method realizes a C like *printf* function to print out error messages. The object in which the error is detected and that calls this method to print out the error message is identified by using the methods *SstObjectC::GetProject*, *SstObjectC::GetClass*, and *SstObjectC::GetObject*. The format of the printed error message is „ERROR in Proj Xxx Class Yyyy, Obj Zzzz:\n\t ....“. By default (when the parameter *pFile* is *NULL*) the messages are written to the file given by the variable *sErrFile*, see below. The output of this method can be switched off by the global variable sError.

Parameters: *pFile* - File descriptor of the destination
pFormat- the printf format string
. - parameters in accordance to the format string.

Result None

virtual ostream &operator << (ostream& os)
virtual istream &operator >> (istream& is)

Description: These methods realizes the stream operators for this class. The output stream operator print out the object name to the output stream by using the method *SstObjectC::GetObject*. The input stream get a name string from the input stream and set objects name with it.

Parameters: *os* - Output stream
is - Input stream

Result: Reference to the output stream
Reference to the input stream

3.2.4 Associated Global Variables

extern bool sError;

Description: This variable determines whether the call of the method *SstObjectC::eprintf* should lead to an output or not. With this switch error messages can be switched off.

Values: *TRUE* | *FALSE*.

extern FILE* sErrFile;

Description: This variable determines the destination of error outputs generated by the method *SstObjectC::eprintf*. By default this variable is set to the *stderr* device. The variable have to be set by using applications.

Values: File descriptor

extern FILE* sOutFile;

Description: This variable determines the destination of outputs generated by the method *SstObjectC::printf*. By default this variable is set to the *stdout* device. The variable have to be set by using applications.

Values: File descriptor

3.3 Class *SstListC<T>*

This class defines double linked list objects. To realize the list object class a further class *SstListEleC* that realizes list element objects. However, this class cannot be used directly. The list element objects hold only references, that is, addresses of data objects of any type, Fig. 2. This has the advantage that existing data objects can be added to list objects without copying the whole data object contents. Thus the changing of data object contents does not lead to inconsistencies between data objects stored in the list object and the data objects itself, since the addresses of data objects are not changed over their lifetime. Furthermore, the data objects stored in one list object can be copied to another list object in a efficient way by copying only their references. Duplicated list objects hold their own instances of list element objects with the same references to same data objects. This has the advantage that a duplicated list object can be handled independently from each other. The disadvantage of this policy is that for each data object that should be put into the list, an extra list element object has to be created in addition to the data object. This leads to more memory allocation inefficiency. A list object has a current element pointer. This current element pointer can be manipulated by several methods. With this methods one can step through the elements of a list object. Each list element object is associated with an index. The index corresponds to the position of the element in the list object. The class is generic, that is, the type of stored data is given as a parameter during the instantiation.

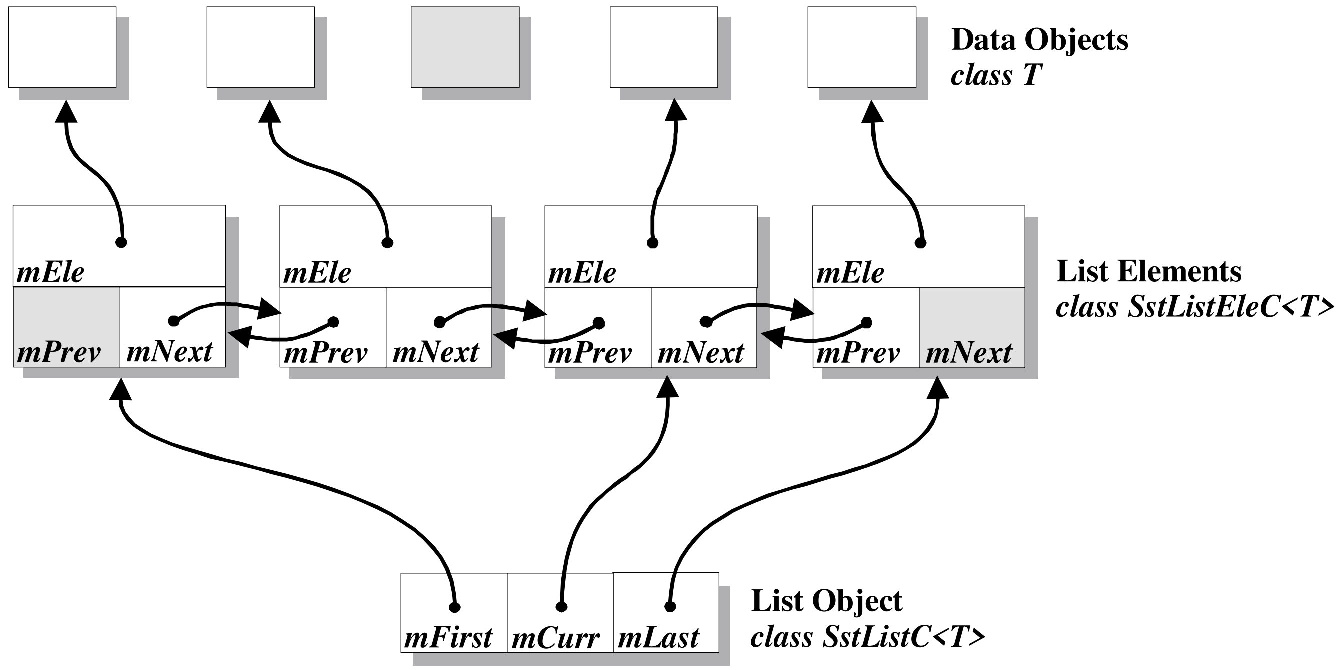


Fig. 2: The class *SstListC<T>*

3.3.1 Class Declaration

```
#include <SST/List.h>

template <class T> class SstListEleC;

template <class T> class SstListC
{
    SstListEleC<T> *mFirst, // pointer to the first element
                   *mLast,  // pointer to the last element
                   *mCurr;  // pointer to the current element
    int mEleNum;            // number of elements in list

    inline SstListEleC<T> *GetListEle(int pInd);
    inline SstListEleC<T> *GetListEle(T *pData);

    inline int Del     (SstListEleC<T> *pEle);
    inline T*  TakeOut (SstListEleC<T> *pEle);
    inline int Kill    (SstListEleC<T> *pEle);

  public:

    // constructors

    inline SstListC () : mFirst(NULL), mLast(NULL), mCurr(mFirst)
                                              , mEleNum(0) {}
    inline SstListC (const SstListC<T> &pList);

    // destructor

    inline  virtual ~SstListC();

    // following methods makes member variables accessible
```

```
int GetEleNum () const { return mEleNum;  }  // value of mEleNum

// miscellaneous methods

inline  int GetIndex(T *pData); // index of an Element (-1 -> not in list)

// methods that handle concatenation of lists

inline  SstListC<T>  operator+ (const SstListC<T> &pList) const;
inline  SstListC<T>  operator- (const SstListC<T> &pList) const;

inline  void  operator = (const SstListC<T> &pList);
inline  void  operator+= (const SstListC<T> &pList);
inline  void  operator-= (const SstListC<T> &pList);
inline  bool  operator== (const SstListC<T> &pList) const;

// methods that control the current position in the list

inline  T *SetFirst   ();
inline  T *SetLast    ();
inline  T *Set        (int pInd);
inline  T *Set        (T *pData);

inline  T *operator++ (int);     // postfix ++
inline  T *operator-- (int);     // postfix --
inline  T *operator++ ();        // prefix ++
inline  T *operator-- ();        // prefix --


// methods that add an element to the list (delivers the position)
// delivers number of elements after processing

inline  int AddFirst (T *pData);              // as the first
inline  int AddLast  (T *pData);              // as the last
inline  int Add      (T *pData);              // at the current pos
inline  int Add      (T *pData, int pInd);    // at an indexed pos

// methods that get an element from the list
// delivers NULL on error

inline  T *GetFirst ();          // from first position
inline  T *GetLast  ();          // from last position
inline  T *Get      ();          // from current position
inline  T *Get      (int pInd);  // from an indexed position

// methods that get and delete elements from the list
// delivers NULL on error

inline  T *TakeOutFirst ();         // from the first pos
inline  T *TakeOutLast  ();         // from the last pos
inline  T *TakeOut      ();         // from the current pos
inline  T *TakeOut      (int pInd); // from an indexed pos

// methods that delete an element from the list without destroying
// delivers an error code

inline  int DelFirst ();         // from the first pos
inline  int DelLast  ();         // from the last pos
inline  int Del      ();         // from the current pos
inline  int Del      (int pInd); // from the indexed pos
inline  int Del      (T *pData); // the element
```

```
    inline  int DelAll   ();        // all elements

    // methods that kill elements of the list, they destroy them !!
    // delivers an error code

    inline  int KillFirst ();           // at the first
    inline  int KillLast  ();           // at the last
    inline  int Kill      (int pInd);   // at an indexed position
    inline  int Kill      (T *pData);   // the element
    inline  int KillAll   ();           // all elements
};
```

3.3.2 Important Data Members

SstListEleC<T> *mFirst;

Description: This data member holds the pointer to the first list element.
Values: Address - the list has least one element
NULL - the list is empty

SstListEleC<T> *mLast;

Description: The data member holds the pointer to the last list element.
Values: Address - the list has least one element
NULL - the list is empty

SstListEleC<T> *mCurr;

Description: The data member holds the pointer to the current list element.
Values: Address - the list has least one element
NULL - the list is empty or the pointer is outside the list

int mEleNum;

Description: The data member holds the current number of list element stored in the list object.
Values: Integer greater/equal 0

3.3.3 Usable Methods/Operators

SstListC ();

Description: The method is the default constructor of the list object. It initializes only its data members.
Parameters: None
Results: None

SstListC (const SstListC<T> &pList);

Description: This method is the copy constructor. For each list element object of the given list object *pList* an list element object of *this* list object is created. After this the references stored in elements of the given list object *pList* are copied to these created elements. This has the advantage that one list object can be destroyed without the confusion between duplications of *this* list object.

Parameters: None
Results: None

virtual ~SstListC ();

Description: The method destroys the list object by deleting all list element objects without destroying referenced data objects.
Parameters: None
Results: None

int GetEleNum () const;

Description: This method delivers the number of list element objects in the list object.
Parameters: None
Results: number of elements

int GetIndex (T *pData) ;

Description: The method gets the index of the list element object that contains the reference to the given data object *pData*.
Parameters: *pData* - address of the given data object
Results: On Success - index of the list element object that references the data object
On Error - gets -1, means that the data object is not stored in a list element

SstListC<T> operator+ (const SstListC<T> &pList) const;

Description: The operator appends the list element objects of a given list object *pList* to *this* list object and delivers a new concatenated list object. The list element objects are concatenated in the same order as they are stored in the original list objects.
Parameters: *pList* - list object
Results: concatenated list object

SstListC<T> operator- (const SstListC<T> &pList) const;

Description: The operator delivers a new list object that contains all list elements that are in *this* list object but not in the given other list object *pList*. With other words, the methods delivers the difference of *this* list objet and the given list object *pList*.
Parameters: *pList* - list object
Results: difference of *this* list and the given list object

void operator = (const SstListC<T> &pList) ;

Description: The operator is the copy operator. First, list element objects that are already in *this* list object are deleted from *this* list object. After this all list element objects of the given list *pList* are copied by creating new list elements and copying only the references to the data objects stored in the elements of the given list *pList*.
Parameters: *pList* - list object
Results: None

void operator+= (const SstListC<T> &pList) ;

Description: The operator appends the elements of a given list object *pList* to *this* list object. The elements are appended in the same order as they are stored in the given list object *pList*

Parameters: *pList* - list object

Results: None

void operator-= (const SstListC<T> &pList) ;

Description: The method delete all elements in *this* list object that are also in the given list object *pList*.

Parameters: *pList* - list object

Results: None

bool operator= = (const SstListC<T> &pList) const;

Description: The operator checks whether *this* list object is equal to the given list object *pList*. A list object is equal to another list object *pList* when

- the number of elements is the same,
- the data objects referenced by list elements are the same (same addresses), and
- the references are stored in the list elements in the same order.

Parameters: *pList* - list object

Results: *TRUE* - *this* list and the given list object are equal
FALSE - *this* list object is different to the given list object

T* SetFirst ();
T* SetLast ();

Description: These methods set the current element pointer to the first/last list element.

Parameters: None

Results: On Success - reference to the data object stored in first/last element
On Error - *NULL*

T* Set (int pInd) ;

Description: The method set the current element pointer to the element with index *pInd*.

Parameters: *pInd* - index

Results: On Success - reference to the data object stored in the indexed element
On Error - *NULL* (the list has no element with given index)

T* Set (T *pData);

Description: The method set the current element pointer to the element that hold the reference to a given data object.

Parameters: pData - data object

Results: On Success - the pointer to the data object
On Error - *NULL* (the data object is not stored in the list)

T* operator++ (int) ;
T* operator- - (int) ;

Description: These methods are the postfix increment/decrement operators. They delivers the reference to the data object stored in the element referenced by the current element pointer and set this pointer to the next/previous list element.
Parameters: None
Results: On Success - the data object reference stored in the former current element
On Error - *NULL* (no next/previous elements)

T* operator++ ();
T* operator- - ();

Description: These methods are the prefix increment/decrement operators. They set the current element pointer to the next/previous list element and delivers the reference to the data object stored in the new current list element.
Parameters: None
Results: On Success - the data object reference stored in the new current element
On Error - NULL (no next/previous elements)

int AddFirst (T *pData);
int AddLast (T *pData);
int Add (T *pData) ;
int Add (T *pData, int pInd);

Description: These methods add a new list element that contains the pointer to the given data object *pData* to the list. The new element is added to the list as
- the first element,
- the last element,
- the successor of the current element given by the current element pointer, or
- at the position with index *pInd*.

In the third case the current element pointer is still referencing the same element.
Parameters: *pData* - the reference of the data object
pInd - the index of the position
Results: the new number of elements in the list object

T* GetFirst ();
T* GetLast ();
T* Get ();
T* Get (int pInd);

Description: These methods get
- the first list element,
- the last list element,
- the current element referenced by the current element pointer, or
- the list element with index *pInd*.

The elements are not removed from the list.
Parameters: pInd - the index of the position
Results: On Success - pointer to the data object

On Error - *NULL*

T* TakeOutFirst ();
T* TakeOutLast ();
T* TakeOut ();
T* TakeOut (int pInd);

Description: These methods take out the
- the first list element,
- the last list element,
- the current element referenced by the current element pointer, or
- the list element with index *pInd*

and deliver the pointer to the data object stored in the element. This means that the elements are removed from the list by these methods.

Parameters: *pInd* - the index of the position

Results: On Success - pointer to the data object
On Error - *NULL*

int DelFirst ();
int DelLast ();
int Del ();
int Del (int pInd);
int Del (T *pData);
int DelAll ();

Description: These methods delete
- the first list element,
- the last list element,
- the current element referenced by the current element pointer,
- the list element with index *pInd*,
- the list element that holds the reference to the given data object *pData*, or
- all list elements.

Is the removed element the element referenced by the current element pointer, the current element pointer is set to the next element.

Parameters: *pData* - the pointer to the data object
pInd - the index of the position

Results: On Success - constant *SST_LST_NoError*
On Error - constant *SST_LST_Error*

int KillFirst ();
int KillLast ();
int Kill ();
int Kill (int pInd);
int Kill (T *pData);
int KillAll ();

Description: These methods delete
- the first list element,

- the last list element,
- the current element given by the current element pointer,
- the list element with index pInd,
- the list element that holds the reference to the given data object pData, or
- all list elements.

Furthermore these method destroy the data object referenced by the removed list element. Is the removed element the element referenced by the current element pointer, the current element pointer is set to the next element.

Parameters: *pData* - the pointer to the data object
pInd - the index of the position

Results: On Success - constant *SST_LST_NoError*
On Error - constant *SST_LST_Error*

3.4 Class *SstStackC<T>*

This class realizes a stack. A stack is a LIFO (Last In First Out) buffer. This means that the element that was put in as the last element is taken out as the first elements. This class provide only methods for push and pop operations.

3.4.1 Class Declaration

```
template <class T> class SstStackC : protected SstListC<T>
{
  public:
    int  *Push (T *pData) { return AddLast(pData); }
    T    *Pop  ()         { return TakeOutLast(); }
};
```

3.4.2 Important Data Members

There are no additional data members.

3.4.3 Usable Methods/Operators

int *Push (T *pData) ;

Description: This method create a new stack element that references the data object *pData* and put it onto the stack at its top.

Parameters: *pData* - pointer to data object

Results: Number of elements in the stack

T *Pop ();

Description: This method get the data object referenced by the element at the top of the stack. The element itself is taken from the stack and is destroyed.

Parameters: None

Results: On Success - Reference to data object
On Error - NULL (no more elements)

3.5 Class *SstQueueC<T>*

This class realizes a queue. A queue is a FIFO (First In First Out) buffer. This means that new elements are put into the queue at its end and can only be taken out from its front. This class provide only methods for put in and take out operations.

3.5.1 Class Declaration

```
template <class T> class SstQueueC : protected SstListC<T>
{
  public:
    int   *In   (T *pData) { return AddLast(pData);  }
    T     *Out  ()         { return TakeOutFirst(); }
};
```

3.5.2 Important Data Members

There are no additional data members.

3.5.3 Usable Methods/Operators

int *In (T *pData);

Description: This method create a new queue element that references the data object *pData* and put it into the queue at its end.
Parameters: *pData* - pointer to data object
Results: Number of elements in the queue

T *Out ();

Description: This method get the data object referenced by the first element at the front of the queue. The element itself is taken out from the queue and is destroyed.
Parameters: None
Results: On Success - Reference to data object
On Error - NULL (no more elements)

3.6 Class *SstVectorC<T>*

This class defines a simple vector object. It realizes basic methods that are necessary to handle with multidimensional data objects. Based on this class several other classes for handling with specific multidimensional data objects like dynamic array or matrices can be realized. Each vector object holds a number of data objects dependent on its dimension. In the simplest case the vector object holds a number of scalar data objects like real or integer values. The class is a generic one so that the elements can be of any type. The dimension of the vector can be changed during the lifetime of an vector object.

3.6.1 Class declaration

```
#include <SST/Vector.h>

template<class T>
class SstVectorC
{
  protected:

    T*   mVec;     // Vector memory
```

```
    int mDim;     // holds the vector dimension

    inline  void  mAlloc (int);
    inline  void  mFree  (void);
    inline  void  mInit  (T pVal);

  public:

    // constructors

    inline  SstVectorC (const SstVectorC<T>& pVec);
    inline  SstVectorC (int pDim = 0);

    // destructor

    inline  virtual ~SstVectorC();

    // methods that handle with the  dimension of the vector object

    inline  int   Dim () const { return mDim; }
    inline  void  Dim (int pDim);
    inline  void  Dim (int pDim, T pVal);

    // miscellaneous methods

    inline  virtual void operator=  (T pVal);
    inline  virtual void operator=  (const SstVectorC<T>& pVec);
    inline  virtual bool operator== (const SstVectorC<T>& pVec) const;

    inline  virtual T&            operator[] (int pInd) const;
    inline  virtual SstVectorC<T> operator() (int pInd, int pLen = 0) const;

    friend  ostream& operator<< (ostream& os, const SstVectorC<T>& pVec);
};
```

3.6.2 Important Data Members

T* mVec;

Description: This data member holds the pointer to the memory that holds the data objects.
Values: Address

int mDim;

Description: This data member holds the dimension of the vector object, that is, the number of data objects.
Values: Integer greater/equal 0

3.6.3 Usable Methods/Operators

SstVectorC (int pDim = 0);

Description: This method is the default constructor that initializes the data members. Furthermore it constructs the vector data objects dependent on the dimension. If the given dimension *pDim* is 0, no data objects are allocated and the vector memory pointer will be set to *NULL*.

Parameters: *mDim* - initial vector dimension

Results: None

SstVectorC (const SstVectorC<T>& pVec);

Description: This method is the copy constructor. It uses the copy operator. First the constructor allocates the same number of data objects. Than it copies the contents of data objects of the given vector to its own data objects. After the construction *this* object has the same dimension as the given vector *pVec* and its data objects holds the same contents as the data objects of the given vector.

Parameters: *pVec* - vector object

Results: None

virtual ~SstVectorC ();

Description: The method destructs the vector object by destroying the data objects.

Parameters: None

Results: None

int Dim () const;

Description: The method delivers the current dimension of the vector object.

Parameters: None

Results: None

void Dim (int pDim);
void Dim (int pDim, T pVal);

Description: The methods change the dimension of the vector object. This is done by creating additional data objects or releasing surplus data objects. The contents of existing data objects is unchanged. If the parameter *pVal* is given, additional data objects are initialized with this value. For this purpose the used type has to provide a copy operator.

Parameters: *pDim* - new vector dimension
pVal - initial value

Results: None

virtual void operator= (T pVal);

Description: The method initialize all data objects with the values given by the parameter *pVal*.

Parameters: *pVal* - initial value

Results: None

virtual void operator= (const SstVectorC<T>& pVec);

Description: This method is the copy operator. First it releases allocated data objects by destroying them. After this, it constructs a number of new data objects dependent on the dimension of the given vector object and copies the contents of the data objects of the given vector to its own data objects. After the execution of this operator *this* vector object has the same dimension as the given vector and its data objects has the same contents as the data objects of the given vector.

Parameters: *pVec* - vector object
Results: None

virtual bool operator= = (const SstVectorC<T>& pVec) const;

Description: This operator compares *this* vector object with the given one. Two vector objects are equal when they hold the same number of data objects and the contents of all data objects is the same.
Parameters: *pVec* - vector object
Results: *TRUE* - the vector objects are equal
FALSE - the vector objects are different

virtual T& operator[] (int pInd) const;

Description: This method delivers the data object with index *pInd*.
Parameters: *pInd* - index of the data object
Results: Reference to the indexed data objects

virtual SstVectorC<T> operator() (int pInd, int pLen = 0) const;

Description: This operator is the slice operator. It extracts *pLen* data objets starting at the index *pInd* from *this* vector object and delivers these data objects in a new one. If the parameter *pLen* is 0, all data objects starting from the index *pInd* are extracted. *This* vector object is unchanged by this operator.
Parameters: *pInd* - begin of the extraction
pLen - number of extracted data objects
Results: New vector object

friend ostream& operator<< (ostream& os, const SstVectorC<T>& pVec);

Description: This method is the stream operator. It put out all data objects of *this* vector separated by tabulator to the given output stream. For using this method the data objects have to provide the stream operator.
Parameters: *os* - output stream
pVec - vector object
Results: Reference to resulting output stream

3.7 Class *Sst3DVectorC*

This class defines a special class of vectors. These vectors are three-dimensional and are given by their three coordinates *x*, *y*, and *z*. The coordinates are real values. Since this class of vectors is often used, especially in conjunction with vectors given in geographic, this class was defined with specific functionality. There are methods for simple vector operations as well methods for transformations from or to its geographic coordinates. Furthermore, there are methods that permit the using either of three-dimensional or geographical vector objects as parameter.

3.7.1 Class Declaration

```
#include <SST/Vector.h>
```

```
class Sst3DVectorC
{
  public:

    double x, y, z;

    Sst3DVectorC(double px = 0, double py = 0
                              , double pz = 0) : x(px), y(py), z(pz) {}

    double mag() { return sqrt(x*x + y*y + z*z); }  // magnitude

    void   operator=  (Sst3DVectorC pVector);
    void   operator=  (SstGeoVectorC pVector);

    void operator+= (Sst3DVectorC pVector);
    void operator+= (SstGeoVectorC pVector);

    Sst3DVectorC operator+ (Sst3DVectorC  pVector);
    Sst3DVectorC operator- (Sst3DVectorC  pVector);
    Sst3DVectorC operator+ (SstGeoVectorC pVector);
    Sst3DVectorC operator- (SstGeoVectorC pVector);

    void   Transform(double CosineMatrix[3][3]);
};
```

3.7.2 Important Data Members

double x;
double y;
double z;

Description: These data members holds the coordinates of the vector. Since these data members are public, the direct access to it is possible.
Values: Double

3.7.3 Usable Methods/Operators

Sst3DVectorC (double px = 0, double py = 0, double pz = 0);

Description: This method constructs the vector given by its *xyz*-coordinates. At the same time this constructor is the default constructor.
Parameters: *px*, *py*, *pz* - vector coordinates
Result: None

double mag();

Description: This method delivers the magnitude of the vector.
Parameters: None
Result: Vector magnitude

void operator= (Sst3DVectorC pVector);
void operator= (SstGeoVectorC pVector);

Description: These operators are the copy operators. They take the given vector *pVector* and set the data members. If the vector is given in geographical coordinates it is transformed to three-dimensional coordinates.
Parameters: *pVector* - three-dimensional/geographical vector
Result: None

void operator+= (Sst3DVectorC pVector);
void operator+= (SstGeoVectorC pVector);

Description: These operators add the coordinates of a given vector *pVector* to the coordinates of *this* vector. If the vector is given in geographical coordinates it is transformed to the three-dimensional coordinates.
Parameters: *pVector* - three-dimensional/geographical vector
Result: None

Sst3DVectorC operator+ (Sst3DVectorC pVector);
Sst3DVectorC operator- (Sst3DVectorC pVector);
Sst3DVectorC operator+ (SstGeoVectorC pVector);
Sst3DVectorC operator- (SstGeoVectorC pVector);

Description: These operators add/subtract the coordinates of a given vector *pVector* and the coordinates of *this* vector. The result is delivered in a new three-dimensional vector. If the vector is given in geographical coordinates it is transformed to the three-dimensional coordinates.
Parameters: *pVector* - three-dimensional/geographical vector
Result: New three-dimensional vector

void Transform(double pCosineMatrix[3][3]);

Description: This method transforms *this* vector object to another three-dimensional coordinate system. The transformation matrix is given by the parameter *pCosineMatrix*.
Parameters: *pCosineMatrix* - transformation matrix
Result: None

3.8 Class *SstGeoVectorC*

This class defines a special class of vectors. These vectors are given by its coordinates in geographic coordinate system, that is by its longitude, latitude, and magnitude. The coordinates are real values. Since this class of vectors is often used, especially in conjunction with vectors given in three-dimensional coordinates, the class was defined with specific functionality. There are methods for simple vector operations as well methods for transformations from or to three-dimensional coordinates. Furthermore, there are methods that permit the using vectors given either of three-dimensional or geographical coordinates as parameter. For transformation from or to the three-dimensional coordinate system it is assumed that the x axis is along the longitude of 0 degree, the y axis is along longitude of 90 degree east and the z axis is pointing to the north pole.

3.8.1 Class Declaration

```
#include <SST/Vector.h>

class SstGeoVectorC
{
double Mag, Lon, Lat;

    SstGeoVectorC(double pMag = 0, double pLon = 0
                                 , double pLat = 0): Mag(pMag), Lon(pLon)
                                                              , Lat(pLat) {}
    double mag() { return Mag; }  // magnitude

    void   operator= (SstGeoVectorC pVector);
    void   operator= (Sst3DVectorC pVector);

    void   operator+= (SstGeoVectorC pVector);
    void   operator+= (Sst3DVectorC pVector);

};
```

3.8.2 Important Data Members

double Mag;
double Lon;
double Lat;

Description: These data members holds the coordinates magnitude, longitude, and latitude of the geographical vector. Since these data members are public the direct access to it is possible.
Values: Double

3.8.3 Usable Methods/Operators

SstGeoVectorC (double pMag = 0, double pLon = 0, double pLat = 0);

Description: This method constructs the vector object given by its geographical coordinates magnitude, longitude, and latitude. At the same time this constructor is the default constructor.
Parameters: *pMag*, - vector magnitude
pLon - vector longitude
pLat - vector latitude
Result: None

double mag ();

Description: This method delivers the magnitude of the vector.
Parameters: None
Result: Vector magnitude

void operator= (SstGeoVectorC pVector);
void operator= (Sst3DVectorC pVector);

Description: These operators are the copy operators. They take the given vector *pVector* and set the data members. If the vector is given in its three-dimensional coordinates it is transformed to geographic coordinates.
Parameters: *pVector* - three-dimensional/geographical vector
Result: None

void operator+= (Sst3DVectorC pVector);
void operator+= (SstGeoVectorC pVector);

Description: These operators add the coordinates of a given vector *pVector* to the coordinates of *this* vector. If the vector is given in three-dimensional coordinates it is transformed to the geographic coordinates.
Parameters: *pVector* - three-dimensional/geographical vector
Result: None

3.9 Class *SstMatrixC<Type>*

The class *SstMatrixC* defines matrix objects. Since a matrix can be considered as a vector of vectors, the class was derived from class *SstVectorC*. It uses the basic functionality of the class *SstVectorC* to provide methods that handles with special properties of matrix objects. The usage of methods and operators of the vector objects class *SstVectorC* can lead to confusion by their application to matrix objects. Therefore such methods and operators like *SstVectorC::Dim* and slice *operator()* are overwritten. A set of methods are realized that deal with specific properties of matrices, e.g., the properties that all row vectors have the same number of elements. The not overwritten methods and operators of the vector object class *SstVectorC* can be used without any restriction. For example, the use of the *operator[]* will get a row vector on which this *operator[]* is also applicable. This leads to the possibility that each matrix element is accessible by using of the *operator[]* for two times. That is, a matrix element can be got by using for example *M[i][j]*. This is corresponding to the normal *C* syntax for the access to elements in two-dimensional arrays. As with vector objects, matrix objects can be resized during their lifetime by using specific methods.

3.9.1 Class Declaration

```
#include <SST/Matrix.h>

template<class Type>
class SstMatrixC : public SstVectorC< SstVectorC<Type> >
{
  public:

    // constructors

    inline  SstMatrixC (int pm = 0, int pn = 0);   // n == 0 -> n = m
    inline  SstMatrixC (const SstMatrixC<Type>& pM)
                       :SstVectorC< SstVectorC<Type> >(pM) {}

    // methods that get/set information about matrix size
```

```
    inline  int   Rows() const { return mDim;          }   // get m dimension
    inline  int   Cols() const { return mVec[0].Dim(); }   // get n dimension
    inline  int   m   () const { return mDim;          }   // get m dimension
    inline  int   n   () const { return mVec[0].Dim(); }   // get n dimension
    inline  int   Rg  () const { return 0;  }              // get the rank

    inline  void  Dim (int pm, int pn);              // set m,n (n==0 -> n=m)
    inline  void  Dim (int pm) { Dim(pm,0); }        // set m = n

    // methods that get manipulated matrices

    inline  SstMatrixC<Type> T();
    inline  SstMatrixC<Type> R(int pDir= 1);
    inline  SstMatrixC<Type> S(int pi, int pj,
                               int pm, int pn) const;  // slice method
    inline  SstMatrixC<Type> S(int pi, int pj) const;  // strip method

    inline  SstMatrixC<Type> operator() (int pi, int pj,
                                         int pm, int pn) const; // slice op
    inline  SstMatrixC<Type> operator() (int pi, int pj) const; // strip op
    inline  SstMatrixC<Type> operator&  (const SstMatrixC<Type> &pM) const;

    // miscellaneous methods

    inline  virtual void  operator=  (Type pVal);
    inline  virtual void  operator&= (const SstMatrixC<Type> &pM);

    friend  ostream& operator<< (ostream& os, const SstMatrixC<Type>& pM);
};
```

3.9.2 Important Data Members

Matrix objects does not need any own data member. The usage of inherited data members of the class *SstVectorC* is sufficient for realization of the matrix object methods.

3.9.3 Usable Methods/Operators

SstMatrixC (int pm = 0, int pn = 0);

Description: This method is the default constructor. It constructs a matrix of data objects of the type *Type*. After construction the matrix object has *pm* rows and *pn* columns. If the parameter *pn* is 0, a square matrix object is constructed, that is, the number of columns is the same as the number of rows. Are both *pm* and *pn* 0, the matrix object contains no data object.

Parameters: *pm* - number of rows
pn - number of columns

Results: None

SstMatrixC (const SstMatrixC<Type>& pM);

Description: This method is the copy constructor. By using the copy constructor of the basis class *SstVectorC*, it constructs an matrix object with the same size as the given matrix object M. It constructs *m* x *n* new data objects and copies the contents of all data objects to its own data objects.

Parameters: *pM* - matrix object

Results: None

int Rows () const;
int m () const;

Description: These methods deliver the number of rows of *this* matrix object.
Parameters: None
Results: Number of rows

int Cols () const;
int n () const;

Description: These methods deliver the number of columns of *this* matrix object.
Parameters: None
Results: Number of columns

int Rg () const;

Description: The method delivers the rank of *this* matrix object.
Parameters: None
Results: Rank of the matrix

void Dim (int pm, int pn);
void Dim (int pm)

Description: These methods set the dimension of the matrix object. For this purpose the methods increase/decrease the number of rows and columns. The contents of additional created data objects during the increasing of the matrix object is undefined. If the parameter *pn* is not given, the matrix is resized to a square matrix.
Parameters: *pm* - number of rows
pn - number of columns
Results: None

SstMatrixC<Type> T();

Description: The method delivers the transposed matrix object
Parameters: None
Results: Transposed matrix object

SstMatrixC<Type> R(int pDir= 1);

Description: This method delivers the rotated/transposed matrix object. The rotation can be done with 90 degrees in the clockwise pDir=1 or counterclockwise pDir=-1 direction. If the given direction *pDir* is 0, the transposed matrix is delivered.
Parameters: *pDir* - direction (1 -> +90 deg, -1 -> -90 deg, 0 -> transposed)
Results: Rotated/Transposed matrix object
SstMatrixC<Type> S(int pi, int pj, int pm, int pn) const;

SstMatrixC<Type> operator() (int pi, int pj, int pm, int pn) const;

Description: This method/operator realizes slicing. The submatrix starting from data object *pi*, *pj* is extracted with a size *pm* x *pn*. If the size expires the matrix object the size parameters are set to the maximum possible value. The method/operator can be distinguished from the strip method (see below) by the number of parameters.

Parameters: *pi* - starting at row
pj - starting at column
pm - number of rows to extract
pn - number of columns to extract

Results: Extracted matrix object

SstMatrixC<Type> S(int pi, int pj) const; // strip method
SstMatrixC<Type> operator() (int pi, int pj) const; // strip op

Description: This method/operator realizes striping. The *pi*-th row and the *pj*-th column are striped. The method/operator can be distinguished from the strip method (see below) by the number of parameters.

Parameters: *pi* - row to strip
pj - column to strip

Results: Resulting matrix object

SstMatrixC<Type> operator& (const SstMatrixC<Type> &pM) const;

Description: This operator concatenates *this* matrix object and a given matrix object to a new matrix object. The number of rows of the resulting matrix object corresponds to the lower number of rows of the concatenated matrix objects. All other rows the bigger matrix are cut.

Parameters: *pM* - given matrix object

Results: Concatenated matrix object

virtual void operator= (Type pVal);

Description: With this method the contents of all data objects of *this* matrix object can be initialized with values given by the parameter *pVal*.

Parameters: *pVal* - initial value

Results: None

virtual void operator&= (const SstMatrixC<Type> &pM);

Description: This operator concatenates a given matrix object to *this* matrix object. The number of rows of *this* matrix object after the execution corresponds to the lower number of rows of the concatenated matrix objects.

Parameters: *pM* - given matrix object

Results: None

friend ostream& operator<< (ostream& os, const SstMatrixC<Type>& pM);

Description: This method is the stream operator. It put out all data objects of *this* vector matrix object to the given output stream. The rows of separated by new line. The output of data objects within one row are separated by tabulators. For using this method the data objects have to provide the stream operator.

Parameters: *os* - output stream
pM - matrix object

Results: Reference to resulting output stream

3.10 Class SstNMatrixC<Type>

This class defines numerical matrix objects, that is matrix objects with numerical data objects. A numerical data object is a data object that provide the basic numerical operations like addition, subtraction, multiplication, and division. That is, each data object that provide these basic numerical operations can be used as an element of a numerical matrix object. Numerical matrix objects are a specialization of normal matrix objects. Therefore the class *SstNMatrixC* was derived from the class *SstMatrixC*. The inherited methods from class *SstMatrixC* are those that are applicable to normal as well to numerical matrix objects. However, numerical matrix objects provide additional functionality. They realize methods for matrix algebra like matrix addition or matrix multiplication. Furthermore, the class *SstNMatrixC* has methods that can be used only for square numerical methods. These methods comprise the computation of the inverse matrix, the cofactor matrix, and the determinant of a matrix. For all of these methods the data objects have to be numerical. The class *SstNMatrixC* is also generic, that is, that the used type of data objects is given as a parameter during the instantiation on a numerical matrix object.

3.10.1 Class Declaration

```
#include <SST/Matrix.h>

template <class Type> class SstNMatrixC : public SstMatrixC<Type>
{
  public:

    SstNMatrixC(const SstNMatrixC <Type>& pM) : SstMatrixC<Type>(pM)     {}
    SstNMatrixC(const SstMatrixC  <Type>& pM) : SstMatrixC<Type>(pM)     {}
    SstNMatrixC(int pm = 0, int pn = 0)       : SstMatrixC<Type>(pm, pn) {}

    virtual void  operator=  (Type pVal) { SstMatrixC<Type>::operator= (pVal);}

    inline  void  operator+= (const SstNMatrixC<Type> &pM);
    inline  void  operator-= (const SstNMatrixC<Type> &pM);
    inline  void  operator*= (const SstNMatrixC<Type> &pM);
    inline  void  operator/= (Type pVal);
    inline  void  operator*= (Type pVal);

    inline  SstNMatrixC<Type>  operator+ (const SstNMatrixC<Type> &pM) const;
    inline  SstNMatrixC<Type>  operator- (const SstNMatrixC<Type> &pM) const;
    inline  SstNMatrixC<Type>  operator* (const SstNMatrixC<Type> &pM) const;
    inline  SstNMatrixC<Type>  operator/ (Type pVal) const;
    inline  SstNMatrixC<Type>  operator* (Type pVal) const;

    // following methods can only be used for square matrices

    SstNMatrixC<Type>  I () const;
    SstNMatrixC<Type>  Ad() const;
    Type               D () const;

    friend  ostream& operator<< (ostream& os, const SstNMatrixC<Type>& pM);
};
```

3.10.2 Important Data Members

There are no additional data members in this class.

3.10.3 Usable Methods/Operators

SstNMatrixC (int pm = 0, int pn = 0);

Description: This method is the default constructor. It constructs a matrix of numerical data objects of the type *Type*. After construction the numerical matrix object has *pm* rows and *pn* columns. If the parameter *pn* is 0, a square numerical matrix object is constructed, that is, the number of columns is the same as the number of rows. Are both *pm* and *pn* 0, the matrix object contains no data object. This constructor is only using the constructor of the basis matrix class *SstMatrixC*.

Parameters: *pm* - number of rows
pn - number of columns

Results: None

SstNMatrixC (const SstNMatrixC <Type>& pM)
SstNMatrixC (const SstMatrixC <Type>& pM)

Description: These method are copy constructors. The copied matrix can be a numerical or a normal matrix provided that the data objects of the normal matrix are also numerically. By using the copy constructor of the basis class *SstMatrixC*, it constructs an numerical matrix object with the same size as the given matrix object M. It constructs m x n new numerical data objects and copies the contents of all data objects to its own data objects.

Parameters: *pM* - matrix object

Results: None

virtual void operator= (Type pVal);

Description: With this method the contents of all data objects of *this* matrix object can be initialized with values given by the parameter *pVal*.

Parameters: *pVal* - initial value

Results: None

SstNMatrixC<Type> operator+ (const SstNMatrixC<Type> &pM) const;
SstNMatrixC<Type> operator- (const SstNMatrixC<Type> &pM) const;
SstNMatrixC<Type> operator* (const SstNMatrixC<Type> &pM) const;

Description: These operators computes the sum/difference/product of *this* numerical matrix object and a given further numerical matrix object. The resulting numerical matrix object is delivered in a new numerical matrix object. For the addition and subtraction, the number of rows of the resulting matrix object corresponds to the lower number of rows of *this* and the given matrix object. All other rows of the bigger matrix are cut. For multiplying *this* and the given matrix object the number of columns of *this* matrix object has to correspond to the number of rows of the given matrix object and vice versa. Otherwise the multiplying is not executed.

Parameters: *pM* - matrix object

Results: Resulting numerical matrix object

SstNMatrixC<Type> operator/ (Type pVal) const;
SstNMatrixC<Type> operator* (Type pVal) const;

Description: These operators divide/multiply all data objects of *this* numerical matrix object through/with a given value. The resulting matrix object is delivered in a new matrix object.
Parameters: *pVal* - given value
Results: Resulting numerical matrix object

void operator+= (const SstNMatrixC<Type> &pM);
void operator-= (const SstNMatrixC<Type> &pM);
void operator*= (const SstNMatrixC<Type> &pM);

Description: These operators computes the adds/subtract/multiply *this* numerical matrix object with a given further numerical matrix object. For the addition and subtraction, the number of rows of the resulting matrix object corresponds to the lower number of rows of *this* and the given matrix object. All other rows of the bigger matrix are cut. For multiplying *this* and the given matrix object the number of columns of *this* matrix object has to correspond to the number of rows of the given matrix object and vice versa. Otherwise the multiplying is not executed.
Parameters: *pM* - matrix object
Results: None

void operator/= (Type pVal);
void operator*= (Type pVal);

Description: These operators divide/multiply all data objects of *this* numerical matrix object through/with a given value.
Parameters: *pVal* - given value
Results: None

SstNMatrixC<Type> I () const;

Description: This method delivers the inverse of *this* matrix object.
Parameters: None
Results: Inverse Matrix

SstNMatrixC<Type> Ad() const;

Description: This method delivers cofactor matrix object of *this* matrix object.
Parameters: None
Results: Cofactor Matrix

Type D () const;

Description: This method delivers cofactor matrix object of *this* matrix object.
Parameters: None
Results: Cofactor Matrix

friend ostream& operator<< (ostream& os, const SstNMatrixC<Type>& pM);

Description: This method uses the output stream operator of the basis class to put out all data objects of *this* matrix object to the given output stream.

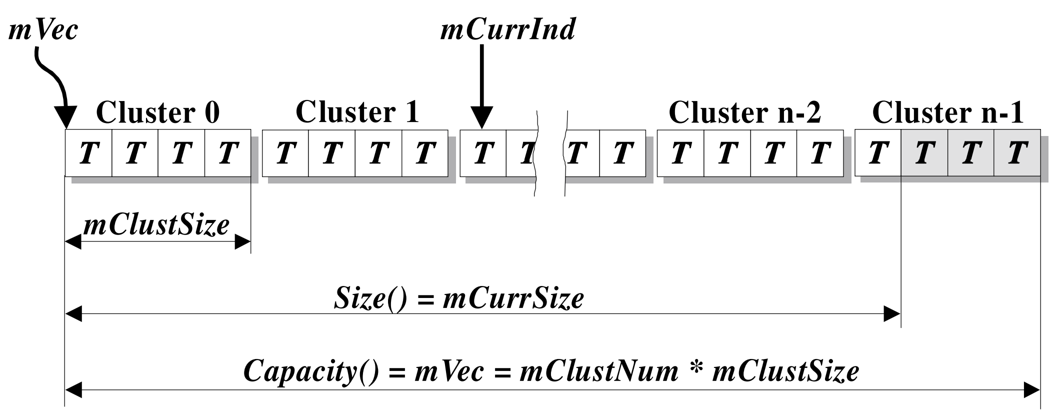
Parameters: *os* - output stream

pM - matrix object

Results: Reference to resulting output stream

3.11 Class *SstArrayC<T>*

The class *SstArrayC* defines dynamic array objects. Since the dynamic array is a special kind of vectors this class is derived from the class *SstVectorC*. However, this class is much more complex than the simple vector class *SstVectorC* and have a lot of additionally features like dynamics. It uses only the data members and basic methods of the vector class to realize its own functionality. Dynamically means that the size of such an array is changed automatically when it is tried to access to an element outside of the existing array. Furthermore, the size of the array can be changed by methods more than once over the lifetime. To avoid the fragmentation of the heap due to the allocation of small memory blocks, array elements can be allocated in clusters with a given cluster size. From this it results that each dynamic array object has a size and a capacity. The size is the number of used elements whereas the capacity is the possible number of element without memory reallocation determined by the product of the number of allocated clusters and the cluster size. The cluster size is fixed over the lifetime of the dynamic array object with the exception of the copy operator. Furthermore, a number of allocable clusters can be specified during the array object construction. By default the number of allocable clusters is not limited. The allocated memory of an array can be initialized with specified value. The dynamic array object has a current element index. The current element index can manipulated by methods and can be used to step through the array. The class *SstArrayC* is a generic class, that is, the type of an array element is given as a parameter on instantiation.



3.11.1 Class Declaration

```
#include <SST/Array.h>

template<class T> class SstArrayC : public SstVectorC<T>
{
    void Allocate (int pSize, T pValue, int pClustSize, int pClustMax);
    void Free     ();

    T   mInitValue;  // initial value of allocated memory
    int mClustNum;   // current number of clusters
```

```
    int mClustSize;  // cluster size
    int mClustMax;   // maximum allowed number of clusters
    int mCurrSize;   // current number of used elements
    int mCurrInd;    // last used index

  public:

    // constructors

    inline  SstArrayC (int pSize, T pValue, int pClustSize, int pClustMax);
    inline  SstArrayC (int pSize = 0, int pClustSize = 16, int pClustMax = 0);
    inline  SstArrayC (const SstArrayC<T>& pArray);
    inline  SstArrayC (const SstVectorC<T>& pVector);

    // destructor

    virtual ~SstArrayC() { mFree(); }

    // methods that control the access

    inline  T& operator[] (int pIndex);
    inline  T& operator++ ();
    inline  T& operator-- ();

    // methods that merge array objects

    inline  void          operator=  (const SstArrayC<T>  &pArray);
    inline  void          operator=  (const SstVectorC<T> &pVector);
    inline  void          operator+= (const SstArrayC<T>  &pArray);
    inline  void          operator+= (const SstVectorC<T> &pArray);
    inline  SstArrayC<T>  operator+  (const SstArrayC<T>  &pArray);
    inline  SstArrayC<T>  operator+  (const SstVectorC<T> &pVector);

    // Methods that manipulate the contents of the dynamic array

    inline  void Init (T pValue);
    inline  void Init ();

    void operator=  (T pVal) { operator[](mCurrInd)    = pVal; }
    void operator+  (T pVal) { operator[](mCurrSize)   = pVal; }
    void operator-  (T pVal) { operator[](mCurrSize-1) = pVal; mCurrSize--; }

    // Methods that deliver and set information about the array

    inline  int Capacity() const  { return mDim;      }
    inline  int Size    () const  { return mCurrSize; }
    inline  int Index   () const  { return mCurrInd;  }
    inline  int i       () const  { return Index();   }

    inline  int Size  (int pSize);
    inline  int Index (int pIndex) { operator[](pIndex); return mCurrInd; }
    inline  int i     (int pIndex) { return Index(pIndex); }

};
```

3.11.2 Important Data Members

T mInitValue;

Description: This data member holds the initial value of elements. The initial value is used to bring the elements to a defined value. The elements are set to this value during their allocation and The value can be set and used by methods.

Values: Any Value of type *T*

int mClustNum;

Description: This data member holds the number of allocated clusters. The value of this data member is determined during construction and the resizing of the array object. The product of the number of allocated clusters with the cluster size determines the capacity of the array, that is, the number of possible elements without new memory allocations.

Values: Integer value greater than 0

int mClustSize;

Description: This data member holds the cluster size, that is, the number of elements per cluster. The cluster size is specified during the construction of the array object and is fixed over its lifetime.

Values: Integer value greater than 0

int mClustMax;

Description: This data member holds the number of allocable clusters. The value is specified during the construction of the array object and is fixed over its lifetime. If this value is 0, there is no limit of allocable clusters.

Values: Integer value greater/equal 0

int mCurrSize;

Description: This data member holds the current size of the array object, that is, the number of used elements.

Values: Integer value greater/equal 0

int mCurrInd;

Description: This data member holds the index of the currently used element. After each access to an element the index of this element is stored in this data member. There are several methods that can manipulate the current element index.

Values: Integer value greater/equal 0

3.11.3 Usable Methods/Operators

SstArrayC (int pSize, T pValue, int pClustSize, int pClustMax);

Description: This method constructs an array object with size *pSize* (number of elements) and a cluster size *pClustSize* (number of elements per cluster). The initial number of clusters is calculated from the size and the cluster size of the array. All elements of allocated clusters are set to the initial value specified by *pValue*. The parameter *pClustMax* specifies the maximum number of allocable clusters. If this parameter 0, the number is not limited.

Parameters: *pSize* - initial size of the array
pValue - initial value
pClustSize - the cluster size (fixed over the lifetime)
pClustMax - number of allocable number of clusters (0 - not limited)

Results: None

SstArrayC (int pSize = 0, int pClustSize = 16, int pClustMax = 0);

Description: This method is the default constructor. It constructs an array object with size *pSize* (number of elements) and a cluster size *pClustSize* (number of elements per cluster). The initial number of clusters is calculated from the size and the cluster size of the array. All elements of allocated clusters are set to the value of the data member *mInitValue*. The parameter *pClustMax* specifies the maximum number of allocable clusters. If this parameter 0, the number is not limited.

Parameters: *pSize* - initial size of the array (by default 0)
pClustSize - the cluster size (fixed over the lifetime; by default 16 elements)
pClustMax - number of allocable number of clusters (0 - not limited)

Results: None

SstArrayC (const SstArrayC<T>& pArray);

SstArrayC (const SstVectorC<T>& pVector);

Description: These methods are the copy constructors. They construct an array object by copying the content of an array object *pArray* or an vector object *pVector*. For this purpose the *operator=* is used. In case of copying a vector object the values of the cluster size and the number of allocable clusters is set to the default values as with the default constructor.

Parameters: *pArray* - copied array

Results: None

virtual ~SstArrayC();

Description: The destructor deletes all array elements by calling of their destructors.

Parameters: None

Results: None

T &operator[] (int pIndex);

Description: This method delivers a reference to the element of the array with index *pIndex*. If the index is outside the array, the array is resized automatically. This means that additional clusters are reserved. These clusters are initialized with the value of the data member *mInitValue*. If the index is smaller than 0, the element referenced by the current element index is delivered. The current element index is set to *pIndex*.

Parameters: *pIndex* - index of the element (<0 - get the current element)

Results: Reference to the indexed element

T &operator++ ();
T &operator- - ();

Description: This method increments/decrements the current element index and delivers the new referenced element. If the current element index is reaching the size of the array/0, the current element index is not increment/decrement and the method delivers the initial value.
Parameters: None
Results: On success - reference to the current element
On error - initial value

SstArrayC<T> operator+ (const SstArrayC<T> &pArray);
SstArrayC<T> operator+ (const SstVectorC<T> &pVector);

Description: This operator concatenates the array object with a given array object *pArray* or a given vector object *pVector*. It delivers a reference to the concatenated array object. The cluster size as well as the number of allocable clusters of the concatenated array object are determined by these values of *this* array object. The size of the concatenated array is sum of *this* and the given array/vector object.
Parameters: *pArray* - array object
pVector - vector object
Results: Reference to the concatenated object

void operator= (const SstArrayC<T> &pArray);
void operator= (const SstVectorC<T> &pVector);

Description: This operator copies the contents of an array object *pArray* or a vector object *pVector*. For this purpose all data members are copied. Furthermore, the needed array memory is allocated and the contents of array/vector elements is copied one by one from the given array/vector. After the execution of this operator the array object holds his own array/vector memory, not that of the copied array/vector. If a vector is copied the values of the number of allocable cluster, the cluster size and the initial value of *this* array object are used. Otherwise these values of *this* object are set that of the copied array object.
Parameters: *pArray* - array object
pVector - vector object
Results: None

void operator+= (const SstArrayC<T> &pArray);
void operator+= (const SstVectorC<T> &pVector);

Description: This operator concatenates a given array object *pArray* or a given vector object *pVector* to *this* array object. The cluster size as well as the number of allocable clusters of the concatenated array object are determined by these values of *this* array object. The size of the concatenated array is the sum of *this* and the given array/vector object.
Parameters: *pArray* - given array object

pVector - vector object
Results: None

void Init (T pValue);
void Init ();

Description: These methods initialize all elements of allocated clusters with an initial value. If the parameter *pValue* is given the elements are initialized operator concatenates a given array object *pArray* to the array object. It delivers a reference to the concatenated array object. The size of the concatenated array is sum of *this* and the given array object.
Parameters: *pArray* - given array object
Results: None

int Capacity () const;

Description: This method delivers the current element capacity of an array object. The capacity is the product of the number of allocated clusters and the cluster size in elements.
Parameters: None
Results: Capacity

int Size () const;

Description: This method delivers the current size of an array object (number of elements).
Parameters: None
Results: Size

int Index () const;
int i () const;

Description: These methods delivers the current element index
Parameters: None
Results: Current element index

int Size (int pSize);

Description: This method change the size (number of elements) of the array object. If it is necessary, new clusters are allocated. The new allocated clusters are initialized with the value given by the data member *mInitValue*. If there are clusters that are not needed longer for the new array size, these clusters are released.
Parameters: *Size* - new array size
Results: Array size after the execution of this method (0 - the method failed)

int Index (int pIndex);
int i (int pIndex);

Description: These methods set the current element index to *pIndex*. For this purpose, the methods use the *operator[]*. This leads to resizing of the array, if the new current element index is laying outside of the array.
Parameters: *pIndex* - new current element index
Results: Current element index

4 References

[1] A. Stepanov, M. Lee, *The Standard Template Library*. Hewlett-Packard Labs, Palo Alto CA, October 1995

[2] ANSI X3J16 WG21, Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++. Doc.-No. X3J16/95-0087, April 1995

[3] M. A. Ellis, B. Stroustrup, The Annotated C++ Reference Manual. Addison-Wesley Reading, April 1995

5 Index

S