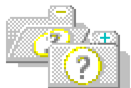
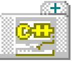
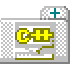


Class Library Reference

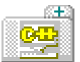
Click the icon above to open all folders. Click an icon below to open a folder or click the underlined text to see a specific topic.



Container Classes describe the classes for creating and manipulating common data structures, such as arrays, lists, and queues.

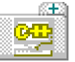


I/O Steam Classes describe the classes for creating and manipulating stream I/O.



Persistent Stream Classes describe the classes for creating and manipulating persistent objects.

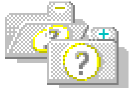
Mathematical Classes describe the bcd and complex mathematical classes.



Run-Time Support Classes describe the classes for exception handling and run-time type information support.

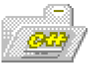
Class Diagnostic Macros describe the macros you can use for debugging your C++ code.

Service Classes describe the classes for handling date, file, string, thread, and time information.



Class Library Reference

Click the icon above to open all folders. Click an icon below to open or close a folder. Click the underlined text to see a specific topic.



Container Classes describe the classes for creating and manipulating common data structures, such as arrays, lists, and queues.

- Arrays
- Associations
- Bags
- Binary Trees

Deques

Dictionaries

Double Lists

Hash Tables

Lists

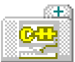
Queues

Sets

Stacks

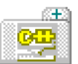
TShouldDelete

Vectors



I/O Steam Classes describe the classes for creating and manipulating stream I/O.

Persistent Stream Classes describe the classes for creating and manipulating persistent objects.

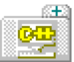


Mathematical Classes describe the bcd and complex mathematical classes.

Run-Time Support Classes describe the classes for exception handling and run-time type information support.

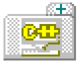
Class Diagnostic Macros describe the macros you can use for debugging your C++ code.

Service Classes describe the classes for handling date, file, string, thread, and time information.

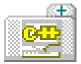


Class Library Reference

Click the icon above to open all folders. Click an icon below to open or close a folder. Click the underlined text to see a specific topic.



Container Classes describe the classes for creating and manipulating common data structures, such as arrays, lists, and queues.



I/O Steam Classes describe the classes for creating and manipulating stream I/O.

conbuf

constream

filebuf

fstreambase

fstream

ifstream

ios

iostream_withassign

iostream

istream_withassign

istream

istrstream

ofstream

ostream_withassign

ostream

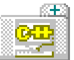
ostrstream

streambuf

strstreambase

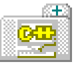
strstreambuf

strstream

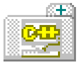


Persistent Stream Classes describe the classes for creating and manipulating persistent objects.

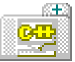
Mathematical Classes describe the bcd and complex mathematical classes.



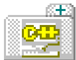
Run-Time Support Classes describe the classes for exception handling and run-time type information support.



Class Diagnostic Macros describe the macros you can use for debugging your C++ code.

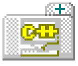


Service Classes describe the classes for handling date, file, string, thread, and time information.

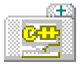


Class Library Reference

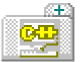
Click the icon above to open all folders. Click an icon below to open or close a folder. Click the underlined text to see a specific topic.



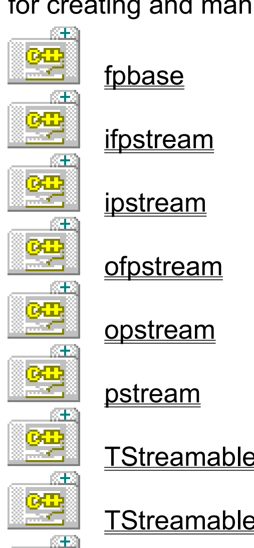
Container Classes describe the classes for creating and manipulating common data structures, such as arrays, lists, and queues.



I/O Steam Classes describe the classes for creating and manipulating stream I/O.

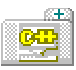


Persistent Stream Classes describe the classes for creating and manipulating persistent objects.

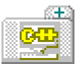


- fpbase
- ifpstream
- ipstream
- ofpstream
- opstream
- pstream
- TStreamableBase
- TStreamableClass
- TStreamer

- Streaming Macros



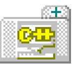
Mathematical Classes describe the bcd and complex mathematical classes.



Run-Time Support Classes describe the classes for exception handling and run-time type information support.

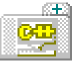
Class Diagnostic Macros describe the macros you can use for debugging your C++ code.

Service Classes describe the classes for handling date, file, string, thread, and time information.

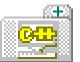


Class Library Reference

Click the icon above to open all folders. Click an icon below to open or close a folder. Click the underlined text to see a specific topic.

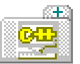


Container Classes describe the classes for creating and manipulating common data structures, such as arrays, lists, and queues.



I/O Steam Classes describe the classes for creating and manipulating stream I/O.

Persistent Stream Classes describe the classes for creating and manipulating persistent objects.



Mathematical Classes describe the bcd and complex mathematical classes.

bcd

complex

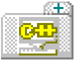
Run-Time Support Classes describe the classes for exception handling and run-time type information support.

Class Diagnostic Macros describe the macros you can use for debugging your C++ code.

Service Classes describe the classes for handling date, file, string, thread, and time information.

Class Library Reference

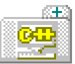
Click the icon above to open all folders. Click an icon below to open or close a folder. Click the underlined text to see a specific topic.



Container Classes describe the classes for creating and manipulating common data structures, such as arrays, lists, and queues.

I/O Steam Classes describe the classes for creating and manipulating stream I/O.

Persistent Stream Classes describe the classes for creating and manipulating persistent objects.



Mathematical Classes describe the bcd and complex mathematical classes.

Run-Time Support Classes describe the classes for exception handling and run-time type information support.

Bad_cast class

Bad_typeid class

set_new_handler()

set_terminate()

set_unexpected()

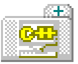
terminate()

typeinfo class

unexpected()

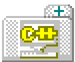
xalloc class

xmsg class



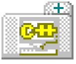
Class Diagnostic Macros describe the macros you can use for debugging your C++ code.

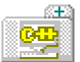
Service Classes describe the classes for handling date, file, string, thread, and time information.

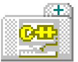


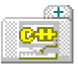
Class Library Reference

Click the icon above to open all folders. Click an icon below to open or close a folder. Click the underlined text to see a specific topic.


Container Classes describe the classes for creating and manipulating common data structures, such as arrays, lists, and queues.


I/O Steam Classes describe the classes for creating and manipulating stream I/O.

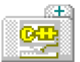

Persistent Stream Classes describe the classes for creating and manipulating persistent objects.


Mathematical Classes describe the bcd and complex mathematical classes.

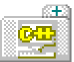
Run-Time Support Classes describe the classes for exception handling and run-time type information support.

Class Diagnostic Macros describe the macros you can use for debugging your C++ code.

- CHECK
- CHECKX
- DIAG_DECLARE_GROUP
- DIAG_DEFINE_GROUP
- DIAG_ENABLE
- DIAG_GETLEVEL
- DIAG_ISENABLED
- DIAG_SETLEVEL
- PRECONDITION
- PRECONDITIONX
- TRACE
- TRACEX
- WARN
- WARNX

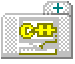


Service Classes describe the classes for handling date, file, string, thread, and time information.



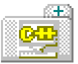
Class Library Reference

Click the icon above to open all folders. Click an icon below to open or close a folder. Click the underlined text to see a specific topic.

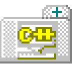


Container Classes describe the classes for creating and manipulating common data structures, such as arrays, lists, and queues.

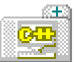
I/O Steam Classes describe the classes for creating and manipulating stream I/O.



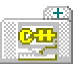
Persistent Stream Classes describe the classes for creating and manipulating persistent objects.



Mathematical Classes describe the bcd and complex mathematical classes.

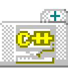


Run-Time Support Classes describe the classes for exception handling and run-time type information support.



Class Diagnostic Macros describe the macros you can use for debugging your C++ code.

Service Classes describe the classes for handling date, file, string, thread, and time information.



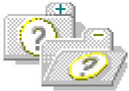
Date class

File class

String classes

Thread classes

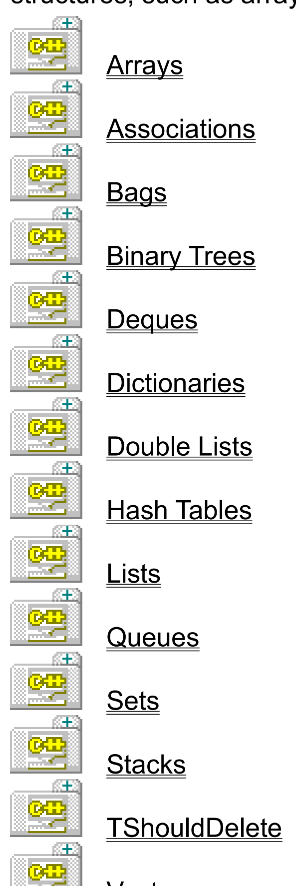
Time class



Class Library Reference

Click any icon to close all folders or click the underlined text to see a specific topic.

Container Classes describe the classes for creating and manipulating common data structures, such as arrays, lists, and queues.



Arrays

Associations

Bags

Binary Trees

Deques

Dictionaries

Double Lists

Hash Tables

Lists

Queues

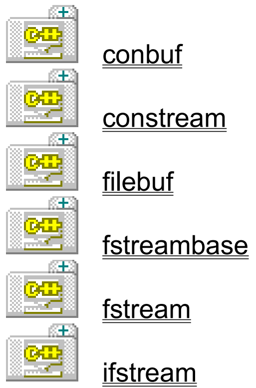
Sets

Stacks

TShouldDelete

Vectors

I/O Steam Classes describe the classes for creating and manipulating stream I/O.



conbuf

constream

filebuf

fstreambase

fstream

ifstream

ios

iostream_withassign

iostream

istream_withassign

istream

istrstream

ofstream

ostream_withassign

ostream

ostrstream

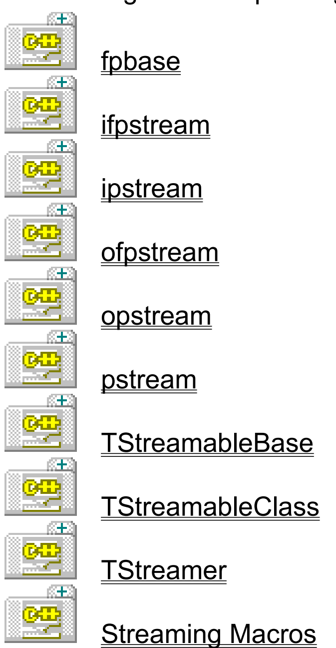
streambuf

strstreambase

strstreambuf

strstream

Persistent Stream Classes describe the classes for creating and manipulating persistent objects.



fpbase

ifpstream

ipstream

ofpstream

opstream

pstream

TStreamableBase

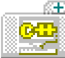
TStreamableClass

TStreamer

Streaming Macros

Mathematical Classes describe the bcd and complex mathematical classes.

bcd



complex

Run-Time Support Classes describe the classes for exception handling and run-time type information support.

Bad_cast class

Bad_typeid class

set_new_handler()

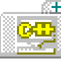
set_terminate()

set_unexpected()

terminate()

typeinfo class

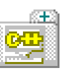
unexpected()



xalloc class

xmsg class

Class Diagnostic Macros describe the macros you can use for debugging your C++ code.



CHECK

CHECKX

DIAG_DECLARE_GROUP

DIAG_DEFINE_GROUP

DIAG_ENABLE

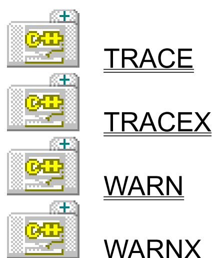
DIAG_GETLEVEL

DIAG_ISENABLED

DIAG_SETLEVEL

PRECONDITION

PRECONDITIONX

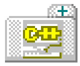


TRACE

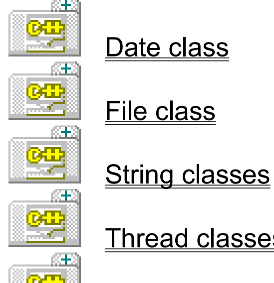
TRACEX

WARN

WARNX



Service Classes describe the classes for handling date, file, string, thread, and time information.



Date class

File class

String classes

Thread classes

Time class

Virtual Classes

See Also

A virtual class virtual is a base class that is passed to more than one derived class, as might happen with multiple inheritance.

You cannot specify a base class more than once in a derived class:

```
class B { ...};
class D : B, B { ... };  // ILLEGAL
```

However, you can indirectly pass a base class to the derived class more than once:

```
class X : public B { ... }
class Y : public B { ... }
class Z : public X, public Y { ... }  // OK
```

In this case, each object of class Z has two sub-objects of class B.

If this causes problems, add the keyword "virtual" to the base class specifier. For example,

```
class X : virtual public B { ... }
class Y : virtual public B { ... }
class Z : public X, public Y { ... }
```

B is now a virtual base class, and class Z has only one sub-object of class B.

Constructors for Virtual Base Classes

Constructors for virtual base classes are invoked before any non-virtual base classes.

If the hierarchy contains multiple virtual base classes, the virtual base class constructors invoke in the order they were declared.

Any non-virtual bases are then constructed before the derived class constructor is called.

If a virtual class is derived from a non-virtual base, that non-virtual base will be first, so that the virtual base class can be properly constructed. For example, this code

```
class X : public Y, virtual public Z
   X one;
```

produces this order:

```
Z();   // virtual base class initialization
Y();   // non-virtual base class
X();   // derived class
```

See Also
virtual (keyword)
Virtual Functions

Virtual Functions

See Also

Virtual functions let derived classes provide different versions of a base class function.

You can declare a virtual function in a base class, then redefine it in any derived class, even if the number and type of arguments are the same. The redefined function overrides the base class function.

You can also declare the functions

```
int Base::Function(int)
```

and

```
int Derived::Function(int)
```

even when they are not virtual.

When you declare virtual functions, keep these guidelines in mind:

- They can only be member functions.
- They can be declared a friend in another class.
- They cannot be a static member.

The base class version is available to derived class objects via scope override. If they are virtual, only the function associated with the actual type of the object is available.

If two functions with the same name and have different arguments, C++ considers them different, and the virtual function mechanism is ignored.

```
virtual void gork(void) = 0;
```

is a pure virtual function. This makes the class an abstract type. `gork()` must be defined by derived classes or redeclared as pure.

Generally, when redefining a virtual function, you cannot change just the function return type. To redefine a virtual function, the new definition (in some derived class) must exactly match the return type and formal parameters of the initial declaration. If two functions with the same name have different formal parameters, C++ considers them different, and the virtual function mechanism is ignored.

However, for certain virtual functions in a base class, their overriding version in a derived class can have a return type that is different from the overridden function. This is possible only when both of the following conditions are met:

- The overridden virtual function returns a pointer or reference to the base class.
- The overriding function returns a pointer or reference to the derived class.

If a base class B and class D (derived publicly from B) each contain a virtual function vf, then if vf is called for an object d of D, the call made is D::vf(), even when the access is via a pointer or reference to B. For example,

```
struct X {};           // Base class.
struct Y : X {};       // Derived class.

struct B {
   virtual void vf1();
   virtual void vf2();
   virtual void vf3();
   void f();
   virtual X* pf();    /* Return type is a pointer to base. */
                       /* This can be overridden. */
   };

class D : public B {
public:
   virtual void vf1(); /* Virtual specifier is legal but redundant. */
```

```
void vf2(int);      /* Not virtual, since it's using a different */
                    /* arg list. This hides B::vf2(). */
// char vf3();
// Illegal: return-type-only change!
void f();
Y*   pf();          /* Overriding function differs only  */
                    /* in return type. Returns a pointer */
                    /* to the derived class.             */
};

 void extf() {
 D d;            /* Instantiate D */
 B* bp = &d;     /* Standard conversion from D* to B*             */
                 /* Initialize bp with the table of functions     */
                 /* provided for object d. If there is no entry   */
                 /* for a function in thed-table, use the         */
                 /* function in the B-table.                      */
 bp->vf1();      /* Calls D::vf1 */
 bp->vf2();      /* Calls B::vf2 since D's vf2 has different args */
 bp->f();        /* Calls B::f (not virtual)                      */

       X* xptr = bp->pf();   /* Calls D::pf() and converts theresult
                                to a pointer to X. */

    D* dptr = &d;
    Y* yptr = dptr->pf();  /* Calls D::pf() and initializes yptr. */
                           /* No further conversion is done.      */
    }
```

The overriding function vf1 in D is automatically virtual. The virtual specifier can be used with an overriding function declaration in the derived class. If other classes will be derived from D, the virtual keyword is required. If no further classes will be derived from D, these of virtual is redundant.

The interpretation of a virtual function call depends on the type of the object it is called for; with nonvirtual function calls, the interpretation depends only on the type of the pointer or reference denoting the object it is called for.

virtual functions exact a price for their versatility: each object in the. derived class needs to carry a pointer to a table of functions in order to select the correct one at run time (late binding).

See Also

virtual (keyword)

Virtual Classes

Container Classes (C++)

These topics are an alphabetical reference guide to the Borland C++ container classes.

TArray

TArrayIterator

TArrayAsVector

TArrayAsVectorIterator

TBagAsVector

TBagAsVectorIterator

TBinarySearchTreeImp

TBinarySearchTreeIteratorImp

TCVectorImp

TCVectorIteratorImp

TDDAssociation

TDequeAsDoubleList

TDequeAsDoubleListIterator

TDequeAsVector

TDequeAsVectorIterator

TDIAssociation

TDictionary

TDictionaryAsHashTable

TDictionaryAsHashTableIterator

TDictionaryIterator

TDoubleListImp

TDoubleListIteratorImp

THashTableImp

THashTableIteratorImp

TIArrayAsVector

TIArrayAsVectorIterator

TIBagAsVector

TIBagAsVectorIterator

TIBinarySearchTreeImp

TIBinarySearchTreeIteratorImp

TICVectorImp
TICVectorIteratorImp
TIDAssociation
TIDequeAsDoubleList
TIDequeAsDoubleListIterator
TIDequeAsVector
TIDequeAsVectorIterator
TIDictionaryAsHashTable
TIDictionaryAsHashTableIterator
TIDoubleListImp
TIDoubleListIteratorImp
TIHashTableImp
TIHashTableIteratorImp
TIIAssociation
TIListImp
TIListIteratorImp
TIQueueAsDoubleList
TIQueueAsDoubleListIterator
TIQueueAsVector
TIQueueAsVectorIterator
TISArrayAsVector
TISArrayAsVectorIterator
TISDoubleListImp
TISDoubleListIteratorImp
TISetAsVector
TISetAsVectorIterator
TISListImp
TISListIteratorImp
TIStackAsList
TIStackAsListIterator
TIStackAsVector
TIStackAsVectorIterator
TISVectorImp
TISVectorIteratorImp
TIVectorImp
TIVectorIteratorImp
TListImp
TListIteratorImp
TMArrayAsVector
TMArrayAsVectorIterator
TMBagAsVector
TMBagAsVectorIterator
TMCVectorImp
TMCVectorIteratorImp

TMDDAssociation
TMDequeAsVector
TMDequeAsVectorIterator
TMDequeAsDoubleList
TMDequeAsDoubleListIterator
TMDIAssociation
TMDictionaryAsHashTable
TMDictionaryAsHashTableIterator
TMDoubleListElement
TMDoubleListImp
TMDoubleListIteratorImp
TMHashTableImp
TMHashTableIteratorImp
TMIArrayAsVector
TMIArrayAsVectorIterator
TMIBagAsVector
TMIBagAsVectorIterator
TMICVectorImp
TMICVectorIteratorImp
TMIDAssociation
TMIDequeAsDoubleList
TMIDequeAsDoubleListIterator
TMIDequeAsVector
TMIDequeAsVectorIterator
TMIDictionaryAsHashTable
TMIDictionaryAsHashTableIterator
TMIDoubleListImp
TMIDoubleListIteratorImp
TMIHashTableImp
TMIHashTableIteratorImp
TMIIAssociation
TMIListImp
TMIListIteratorImp
TMIQueueAsDoubleList
TMIQueueAsDoubleListIterator
TMIQueueAsVector
TMIQueueAsVectorIterator
TMISArrayAsVector
TMISDoubleListImp
TMISDoubleListIteratorImp
TMISetAsVector
TMISetAsVectorIterator
TMISListImp
TMISListIteratorImp

TMIStackAsList
TMIStackAsListIterator
TMIStackAsVector
TMIStackAsVectorIterator
TMISVectorImp
TMISVectorImp
TMISVectorIteratorImp
TMIVectorIteratorImp
TMListElement
TMListImp
TMListIteratorImp
TMListIteratorImp
TMQueueAsDoubleList
TMQueueAsDoubleListIterator
TMQueueAsVector
TMQueueAsVectorIterator
TMSArrayAsVector
TMSArrayAsVectorIterator
TMSDoubleListImp
TMSDoubleListIteratorImp
TMSetAsVector
TMSetAsVectorIterator
TMSListImp
TMSListIteratorImp
TMStackAsList
TMStackAsListIterator
TMStackAsVector
TMStackAsVectorIterator
TMSVectorImp
TMSVectorIteratorImp
TMVectorImp
TMVectorIteratorImp
TQueue
TQueueAsDoubleList
TQueueAsDoubleListIterator
TQueueAsVector
TQueueAsVectorIterator
TQueueIterator
TSArray
TSArrayAsVector
TSArrayAsVectorIterator
TSArrayIterator
TSDoubleListImp
TSDoubleListIteratorImp

TSet
TSetAsVector
TSetAsVectorIterator
TSetIterator
TShouldDelete
TSListImp
TSListIteratorImp
TStack
TStackAsList
TStackAsListIterator
TStackAsVector
TStackAsVectorIterator
TStackIterator
TSVectorImp
TSVectorIteratorImp
TVectorImp
TVectorIteratorImp

See Also

Categorical listing of Container classes

Container Classes (by Category)

See Also Summary

These topics are a reference guide to the Borland C++ container classes. Each container class belongs to one of the following groups:

Arrays	Dictionaries	Sets
Associations	Double Lists	Stacks
Bags	Hash Tables	TShouldDelete
Binary Trees	Lists	Vectors
Deques	Queues	

See Also

Alphabetical listing of Container classes

Container Classes Summary

You use containers to store objects. The underlying data structure used for these containers determines how objects are stored and retrieved.

Containers can store objects either directly or indirectly. Direct storage is done by copying the object into the container; while indirect storage uses a pointer to store the object in the container. The method you choose is dependant upon the size of the object you want to store. Smaller objects should be stored directly and larger objects indirectly.

Borland C++ includes a template-based container class library, that uses encapsulation so you can maximize container storage with a minimal amount of reprogramming.

These classes build containers from ADTs (abstract data types), and use FDS (fundamental data structure) as an underlying data structure. Together ADTs and FDSs determine a containers storage stategy.

Each template must be instantiated with a particular data type as the type of element that it will hold. These classes provide direct control over the types of objects stored in the container.

The syntax for using the container classes is:

```
template <class [T,I,V,Alloc,List,Stk,Vect]> class ContainerClassName
```

A template can use more than one class as the first class parameter declaration.

Class	What it means
T	template
I	indirect (pointer to a class or object)
V	virtual
Alloc	allocator
List	list
Stk	stack
Vect	vector

Fundamental Data Structures

Fundamental data types are low-level containers you can instantiate. Fundamental data types are the underlying data structures upon which you build containers and they do not necessarily need an accompanying abstract data type.

BTree	useful for storage and retrieval of large, dynamic objects
DoubleList	objects are stored in a linearly linked list which can be searched in both directions
HashTable	an unordered collection where storage and retrieval of objects are done by a hash value
List	objects are stored in linearly linked list which can be searched in one direction only
Vector	index-based storage beginning with 0; useful when the number of objects to be stored is known

Abstract Data Types

Abstract data types are high-level containers which have at least one pure virtual function. The existance of a pure virtual function means that the abstract data type container classes cannot be instantiated without an underlying fundamental data structure.

Array	index-based storage beginning with 1; array size need not be known
Bag	objects are stored in no particular order, objects are not sortable and you can have multiple instances
Deque	objects are stored or retrieved at either the head or tail of the container
Dictionary	ragged vector storage when objects are not the same size
Queue	objects are stored at the tail and retrieved from the head
Set	objects are stored in no particular order, objects are not sortable but you can have only one instance
Stack	objects are stored and retrieved from the same end of the list

Array classes

Array containers manage arrays of objects. This family includes nine container classes, together with their corresponding nine iterator classes.

Containers in this family manage objects and pointers to objects, and can sort objects. All arrays can be resized. Some arrays can be sorted. Most classes use the default memory manager TStandardAllocator, but you can pass your own memory manager class to managed container class templates.

Here is a list of container classes comprising this family:

TArray	TArrayIterator
TArrayAsVector	TMIArrayAsVector
TArrayAsVectorIterator	TMIArrayAsVectorIterator
TIArrayAsVector	TMISArrayAsVector
TIArrayAsVectorIterator	TMSArrayAsVector
TISArrayAsVector	TMSArrayAsVectorIterator
TISArrayAsVectorIterator	TSArrayAsVector
TMArrayAsVector	TSArrayAsVectorIterator
TMArrayAsVectorIterator	

Association classes

Association objects bind a key to a value. When given a specific key-type value, an object returns the value-type data associated with that key.

This class family includes eight classes. Individual classes in this family can contain data, or pointers to data. Most classes use the default memory manager TStandardAllocator, but you can pass your own memory manager class to managed association class templates.

Association classes are not containers. They can not hold an expanding number of elements of your user type. They are designed to hold only one key and one value.data item. Association classes are designed to be contained in a dictionary object, and to return *value* data, given *key* data as an input parameter.

Here is a list of container classes comprising this family:

TDDAssociation	TMDDAssociation
TDIAssociation	TMDIAssociation
TIDAssociation	TMIDAssociation
TIIAssociation	TMIIAssociation

Bag classes

Bag containers manage bags of objects. A bag is a container which holds any number of objects in any order, and of any value. The bag is the least structured of all the data structures supported by any container in the Container class families. The bag family includes four container classes, together with their corresponding four iterator classes.

Individual containers in this family manage objects and pointers to objects. Most classes use the default memory manager TStandardAllocator, but you can pass your own memory manager class to managed container class templates..

Here is a list of container classes comprising this family:

TBagAsVector	TMBagAsVector
TBagAsVectorIterator	TMBagAsVectorIterator
TIBagAsVector	TMIBagAsVector
TIBagAsVectorIterator	TMIBagAsVectorIterator

Binary Tree classes

Binary tree containers manage data placed into nodes, where each node can connect to one parent and up to two child nodes. This family includes two container classes, together with their corresponding two iterator classes.

Individual containers in this family manage objects and pointers to objects. Binary search tree templates use the default memory manager TStandardAllocator. You can not pass your own memory manager to a template in the binary tree family.

By default, binary trees connect nodes in an inorder sequence. This means that binary tree nodes are connected so that the value placed into a parent node is more than the value placed into the left child node and less than the value placed into the right child node. Repeated values are placed into right child nodes on the tree, and the tree is restructured if necessary. This strategy creates an inherently sorted tree that can be searched more quickly than can array, linked list, or hash table containers. You can change default sequencing to preorder or postorder node sequencing from the container constructor, when the container is instantiated.

Binary trees are unbalanced. The order in which you place data into the tree determines the shape of the tree. Binary tree objects do not adjust themselves in order to build more symmetrical shapes.

Binary tree objects adjust themselves automatically when you add or delete data from the tree. If a duplicate value is added or a parent node is deleted, the object promotes the appropriate child node and recursively adjusts the tree.without your intervention.

Here is a list of container classes comprising this family:

TBinarySearchTreeImp

TBinarySearchTreeIteratorImp

TIBinarySearchTreeImp

TIBinarySearchTreeIteratorImp

Dequeue classes

Deque objects manage a train of objects of type *T*, where objects can be placed or retrieved from either the head or the tail of the train. The queue structure stores head and tail objects in the order they were received. A call to the head or the tail surrenders the last object placed at the head or tail. The queue family includes nine container classes, together with their corresponding nine iterator classes.

Individual containers in this family manage objects and pointers to objects. Most classes use the default memory manager TStandardAllocator, but you can pass your own memory manager class to managed container class templates.

You can implement deque structures as vectors or as double-linked lists. Both kinds of class templates take the same parameters and present the same list of member functions. For example, if you change your class usertype from *TDequeAsVector* to *TDequeAsDoubleList, you* will not need to change calls you made to deque member functions in your source code.

A *DequeAsVector* class can not be resized; a *DequeAsList* class can be resized.

Here is a list of container classes comprising this family:

TDequeAsDoubleList	TMDequeAsDoubleList
TDequeAsDoubleListIterator	TMDequeAsDoubleListIterator
TDequeAsVector	TMDequeAsVector
TDequeAsVectorIterator	TMDequeAsVectorIterator
TIDequeAsDoubleList	TMIDequeAsDoubleList
TIDequeAsDoubleListIterator	TMIDequeAsDoubleListIterator
TIDequeAsVector	TMIDequeAsVector
TIDequeAsVectorIterator	TMIDequeAsVectorIterator

Dictionary classes

Dictionary containers manage objects holding data. A dictionary object usually stores objects instantiated from an association class, which returns value data given key data.

Individual dictionary containers can manage objects or pointers to objects. Two classes use the default memory manager TStandardAllocator. Two other classes can accept your own memory manager as an input parameter.

The dictionary class family includes four container classes, together with their corresponding four iterator classes, and two helper classes. Direct classes are base classes. Indirect classes derive from the TShouldDelete class.

Here is a list of container classes comprising this family:

TDictionaryAsHashTable	TIDictionaryAsHashTableIterator
TDictionaryAsHashTableIterator	TMDictionaryAsHashTable
TDictionary	TMDictionaryAsHashTableIterator
TDictionaryIterator	TMIDictionaryAsHashTable
TIDictionaryAsHashTable	TMIDictionaryAsHashTableIterator

Double List classes

A double list container manages a chain of nodes, where each node supports a pointer to the previous node and a pointer to the next node in the chain. This family includes eight container classes, together with their corresponding eight iterator classes.

Individual containers in this family manage node objects and pointers to node objects. Some classes can automatically maintain a sorted double list. Most classes use the default memory manager TStandardAllocator, but you can pass your own memory manager class to managed list class templates.

Normally, a list data structure contains a chain of nodes. To build a list, you must define a node. Containers in this family handle this task for you.

All containers in this family use a protected class, *TMDoubleListElement*, to define and instantiate a node object and manage pointers which position it within the list. You never need to instantiate an object of this class directly.

Here is a list of container classes comprising this family:

TDoubleListImp	TMIDoubleListImp
TDoubleListIteratorImp	TMIDoubleListIteratorImp
TIDoubleListImp	TMISDoubleListImp
TIDoubleListIteratorImp	TMISDoubleListIteratorImp
TISDoubleListImp	TMSDoubleListImp
TISDoubleListIteratorImp	TMSDoubleListIteratorImp
TMDoubleListElement	TSDoubleListImp
TMDoubleListImp	TSDoubleListIteratorImp
TMDoubleListIteratorImp	

Hash Table classes

Hash table containers define a hash table data structure which manages data of type *T*. All container classes in this family use a hash function to assign a unique key to data placed in the table.

The hash table class family includes four container classes, together with their corresponding four iterator classes. Two classes use the default memory manager TStandardAllocator. Two other classes can accept your own memory manager class as an input parameter.

Hash table classes are used to implement Dictionary classes. Hash table classes can be used independently, provided that your program provides either a global *HashValue* function, or a *HashValue* member within the class providing objects to be stored. The *HashValue* function for built-in types is already provided.

Here is a list of container classes comprising this family:

TMHashTableImp	TMIHashTableIm
TMHashTableIteratorImp	TMIHashTableIteratorImp
THashTableImp	TIHashTableImp
THashTableIteratorImp	TIHashTableIteratorImp

List classes

List containers manage a chain of nodes containing data of type *T*, where each node supports a pointer to the previous node in the chain. This family includes eight container classes, together with their corresponding eight iterator classes.

Individual containers in this family manage node objects and pointers to node objects. Some classes can automatically maintain a sorted list. Most classes use the default memory manager TStandardAllocator, but you can pass your own memory manager class to managed container class templates.

Normally, a list data structure contains a chain of nodes. To build a list, you must define a node. Containers in this family handle this task for you.

All containers in this family use a protected class, *TListElement*, to define and instantiate a node object and manage pointers which position it within the list. You never need to instantiate an object of this class directly.

Here is a list of container classes comprising this family:

TIListImp	TMListElement
TIListIteratorImp	TMListImp
TISListImp	TMListIteratorImp
TListImp	TMSListImp
TListIteratorImp	TMSListIteratorImp
TMIListImp	TSListImp
TMIListIteratorImp	TSListIteratorImp
TMISListImp	TSListIteratorImp
TMISListIteratorImp	

Queue classes

A queue container manages a train of objects, where objects are added to the train from the tail position, and removed from the train from the head position. This family includes nine container classes, together with their corresponding nine iterator classes.

Individual containers in this family manage objects and pointers to objects. Most classes use the default memory manager TStandardAllocator, but you can pass your own memory manager class to managed container class templates.

You can implement queue structures as vectors or as double-linked lists. Both kinds of class templates take the same parameters and present the same list of member functions. For example, if you change your class usertype from *TQueueAsVector* to *TQueueAsDoubleList, you* will not need to change calls you made to queue member functions in your source code.

A *QueueAsVector* class can not be resized; a *QueueAsList* class can be resized.

Here is a list of container classes comprising this family:

TIQueueAsDoubleList	TMQueueAsDoubleListIterator
TIQueueAsDoubleListIterator	TMQueueAsVector
TIQueueAsVector	TMQueueAsVectorIterator
TIQueueAsVectorIterator	TQueue
TMIQueueAsDoubleList	TQueueAsDoubleList
TMIQueueAsVector	TQueueAsDoubleListIterator
TMIQueueAsVectorIterator	TQueueAsVector
TMQueueAsDoubleList	TQueueAsVectorIterator
TMQueueAsDoubleListIterator	TQueueIterator

Set classes

Set containers manage data items as an unordered group of non-repeating objects. The set family includes four container classes, together with their corresponding four iterator classes.

Individual containers in this family manage objects and pointers to objects. Most classes use the default memory manager TStandardAllocator, but you can pass your own memory manager object to managed container classes upon instantiation.

Here is a list of container classes comprising this family:

TISetAsVector	TMSetAsVectorIterator
TISetAsVectorIterator	TSet
TMISetAsVector	TSetAsVector
TMISetAsVectorIterator	TSetAsVectorIterator
TMSetAsVector	TSetIterator

Stack classes

Stack containers manage an ordered chain of objects of type T, where objects are sequenced in the order they were placed onto the chain, and objects can be added or deleted only from the first (top) position in the chain. This is commonly called "Last-in, First-out" order. This family includes eight container classes, together with their corresponding eight iterator classes.

Individual containers in this family manage objects and pointers to objects. Most classes use the default memory manager TStandardAllocator, but you can pass your own memory manager object to managed container classes upon instantiation.

You can implement stack structures as vectors or as linked lists. Both kinds of class templates take the same parameters and present the same list of member functions. For example, if you change your class usertype from *TStackAsVector* to *TStackAsList, you* will not need to change calls you made to stack member functions in your source code.

A *StackAsVector* class can not be resized; a *StackAsList* class can be resized.

Here is a list of container classes comprising this family:

TIStackAsList	TMStackAsListIterator
TIStackAsListIterator	TMStackAsVector
TIStackAsVector	TMStackAsVectorIterator
TIStackAsVectorIterator	TStack
TMIStackAsList	TStackAsList
TMIStackAsListIterator	TStackAsListIterator
TMIStackAsVector	TStackAsVector
TMIStackAsVectorIterator	TStackAsVectorIterator
TMStackAsList	TStackIterator

Vector classes

Vector containers manage contiguous blocks of memory, where each block contains an object of type *T*. This family includes thirteen container classes, together with their corresponding thirteen iterator classes.

Individual containers in this family manage objects and pointers to objects, and can sort and count objects. Most classes use the default memory manager TStandardAllocator, but you can pass your own memory manager object to managed container object upon instantiation.

Here is a list of container classes comprising this family:

TCVectorImp	TMISVectorImp
TCVectorIteratorImp	TMISVectorIteratorImp
TICVectorImp	TMIVectorImp
TICVectorIteratorImp	TMIVectorIteratorImp
TISVectorImp	TMSVectorImp
TISVectorIteratorImp	TMSVectorIteratorImp
TIVectorImp	TMVectorImp
TIVectorIteratorImp	TMVectorIteratorImp
TMCVectorImp	TSVectorImp
TMCVectorIteratorImp	TSVectorIteratorImp
TMICVectorImp	TVectorImp
TMICVectorIteratorImp	TVectorIteratorImp

arrays.h

See Also   Header Files

This header file contains the template definitions for these classes, their data members and member functions:

TArray
TArrayAsVector
TArrayAsVectorIterator
TIArrayAsVector
TIArrayAsVectorIterator
TISArrayAsVector
TISArrayAsVectorIterator
TMArrayAsVector
TMArrayAsVectorIterator
TArrayIterator
TMIArrayAsVector
TMIArrayAsVectorIterator
TMISArrayAsVector
TMSArrayAsVector
TMSArrayAsVectorIterator
TSArrayAsVector
TSArrayAsVectorIterator

assoc.h

See Also   Header Files

This header file contains the template definitions for these classes, their data members and member functions:

TDDAssociation	TMDDAssociation
TDIAssociation	TMDIAssociation
TIDAssociation	TMIDAssociation
TIIAssociation	TMIIAssociation

bags.h

See Also Header Files

This header file contains the template definitions for these classes, their data members and member functions:

TBagAsVector	TMBagAsVector
TBagAsVectorIterator	TMBagAsVectorIterator
TIBagAsVector	TMIBagAsVector
TIBagAsVectorIterator	TMIBagAsVectorIterator

binimp.h

See Also Header Files

This header file contains the template definitions for these classes, their data members and member functions:

TBinarySearchTreeImp

TBinarySearchTreeIteratorImp

TIBinarySearchTreeImp

TIBinarySearchTreeIteratorImp

deques.h

See Also Header Files

This header file contains the template definitions for these classes, their data members and member functions:

TDequeAsDoubleListIterator	TMDequeAsDoubleListIterator
TDequeAsDoubleList	TMDequeAsDoubleList
TDequeAsVectorIterator	TMDequeAsVectorIterator
TDequeAsVector	TMDequeAsVector
TIDequeAsDoubleListIterator	TMIDequeAsDoubleListIterator
TIDequeAsDoubleList	TMIDequeAsDoubleList
TIDequeAsVectorIterator	TMIDequeAsVectorIterator
TIDequeAsVector	TMIDequeAsVector

dict.h

See Also   Header Files

This header file contains the template definitions for these classes, their data members and member functions:

TDictionaryAsHashTableIterator	TIDictionaryAsHashTable
TDictionaryAsHashTable	TMDictionaryAsHashTableIterator
TDictionaryIterator	TMDictionaryAsHashTable
TDictionary	TMIDictionaryAsHashTableIterator
TIDictionaryAsHashTableIterator	TMIDictionaryAsHashTable

dlistimp.h

See Also Header Files

This header file contains the template definitions for these classes, their data members and member functions:

TDoubleListImp	TMIDoubleListImp
TDoubleListIteratorImp	TMIDoubleListIteratorImp
TIDoubleListImp	TMISDoubleListImp
TIDoubleListIteratorImp	TMISDoubleListIteratorImp
TISDoubleListImp	TMSDoubleListImp
TISDoubleListIteratorImp	TMSDoubleListIteratorImp
TMDoubleListElement	TSDoubleListImp
TMDoubleListImp	TSDoubleListIteratorImp
TMDoubleListIteratorImp	

hashimp.h

See Also   Header Files

This header file contains the template definitions for these classes, their data members and member functions:

TMHashTableImp	TMIHashTableImp
TMHashTableIteratorImp	TMIHashTableIteratorImp
THashTableImp	TIHashTableImp
THashTableIteratorImp	TIHashTableIteratorImp

listimp.h

See Also   Header Files

This header file contains the template definitions for these classes, their data members and member functions:

TIListImp	TMListElement
TIListIteratorImp	TMListImp
TISListImp	TMListIteratorImp
TListImp	TMSListImp
TListIteratorImp	TMSListIteratorImp
TMIListImp	TSListImp
TMIListIteratorImp	TSListIteratorImp
TMISListImp	TSListIteratorImp
TMISListIteratorImp	

queues.h

See Also Header Files

This header file contains the template definitions for these classes, their data members and member functions:

- TIQueueAsDoubleList
- TIQueueAsDoubleListIterator
- TIQueueAsVector
- TIQueueAsVectorIterator
- TMIQueueAsDoubleList
- TMIQueueAsVector
- TMIQueueAsVectorIterator
- TMQueueAsDoubleList
- TMQueueAsDoubleListIterator
- TMQueueAsDoubleListIterator
- TMQueueAsVector
- TMQueueAsVectorIterator
- TQueue
- TQueueAsDoubleList
- TQueueAsDoubleListIterator
- TQueueAsVector
- TQueueAsVectorIterator
- TQueueIterator

sets.h

See Also Header Files

This header file contains the template definitions for these classes, their data members and member functions:

TISetAsVector	TMSetAsVectorIterator
TISetAsVectorIterator	TSet
TMISetAsVector	TSetAsVector
TMISetAsVectorIterator	TSetAsVectorIterator
TMSetAsVector	TSetIterator

stacks.h

See Also Header Files

This header file contains the template definitions for these classes, their data members and member functions:

TIStackAsList	TMStackAsListIterator
TIStackAsListIterator	TMStackAsVector
TIStackAsVector	TMStackAsVectorIterator
TIStackAsVectorIterator	TStack
TMIStackAsList	TStackAsList
TMIStackAsListIterator	TStackAsListIterator
TMIStackAsVector	TStackAsVector
TMIStackAsVectorIterator	TStackAsVectorIterator
TMStackAsList	TStackIterator

vectimp.h

See Also Header Files

This header file contains the template definitions for these classes, their data members and member functions:

TCVectorImp
TCVectorIteratorImp
TICVectorImp
TICVectorIteratorImp
TISVectorImp
TISVectorIteratorImp
TIVectorImp
TIVectorIteratorImp
TMCVectorImp
TMCVectorIteratorImp
TMICVectorImp
TMICVectorIteratorImp
TMISVectorImp
TMISVectorIteratorImp
TMIVectorImp
TMIVectorIteratorImp
TMSVectorImp
TMSVectorIteratorImp
TMVectorImp
TMVectorIteratorImp
TSVectorImp
TSVectorIteratorImp
TVectorImp
TVectorIteratorImp

shddel.h

See Also Header Files

This header file contains the definitions for these classes, their data members and member functions:

TShouldDelete

See Also
Precompiled Headers

TStandardAllocator class

Syntax

```
class TStandardAllocator
```

Description

Provides class-specific operator new and operator delete that simply call the global operator new and operator delete. That is, TStandardAllocator does not provide any specialized behavior. It is used in the non-managed versions of the parametrized containers.

TMArrayAsVector template

Syntax

```
template <class T, class Alloc> class TMArrayAsVector;
```

Header File

arrays.h

Description

TMArrayAsVector implements a managed array of objects of type T, using a vector as the underlying implementation. It requires an == operator for type T. The memory manager Alloc provides class-specific new and delete operators.

Type Definitions

CondFunc

IterFunc

Public Constructor

TMArrayAsVector::TMArrayAsVector

Public Member Functions

Add

AddAt

ArraySize

BoundBase

Destroy

Detach

Find

FirstThat

Flush

ForEach

GetItemsInContainer

Grow

HasMember

InsertEntry

IsEmpty

IsFull

LastThat

LowerBound

Reallocate

RemoveEntry

SetData

UpperBound

ZeroBase

Protected Member Functions

ItemAt

Operators

[]

TMArrayAsVector::CondFunc

TMArrayAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMArrayAsVector::IterFunc

TMArrayAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to the ForEach member function.

TMArrayAsVector::TMArrayAsVector

TMArrayAsVector class

Syntax

```
TMArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Description

Creates an array with an upper bound of upper, a lower bound of lower, and a growth delta of delta.

TMArrayAsVector::Add

TMArrayAsVector class

Syntax

```
int Add( const T& t )
```

Description

Adds a T object at the next available index at the end of an array. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs and delta is nonzero, the array is expanded (by sufficient multiples of delta bytes) to accommodate the addition. If delta is zero, Add fails. Add returns 0 if it couldn't add the object.

TMArrayAsVector::AddAt

TMArrayAsVector class

Syntax

```
int AddAt( const T& t, int loc )
```

Description

Adds a T object at the specified index. If that index is occupied, it moves the object up to make room for the added object. If loc is beyond the upper bound, the array is expanded if delta (see the constructor) is nonzero. If delta is zero, attempting to AddAt beyond the upper bound gives an error.

TMArrayAsVector::ArraySize

TMArrayAsVector class

Syntax

```
unsigned ArraySize() const;
```

Description

Returns the current number of cells allocated.

TMArrayAsVector::BoundBase

See Also TMArrayAsVector class

Syntax

```
int BoundBase( unsigned loc ) const;
```

Description

Boundbase adjust vectors, which are zero-based, to arrays, which aren't zero-based.

See Also
ZeroBase.

TMArrayAsVector::Destroy

TMArrayAsVector class

Form 1

```
int Destroy( int i )
```

Form 2

```
int Destroy( const T& t )
```

Description

Form 1: Removes the object at the given index. The object will be destroyed.

Form 2: Removes the given object and destroys it.

TMArrayAsVector::Detach

See Also TMArrayAsVector class

Form 1

```
int Detach( int loc )
```

Form 2

```
int Detach( const T& t )
```

Description

Form 1: Removes the object at loc.

Form 2: Removes the first object that compares equal to the specified object.

See Also
TShouldDelete::ownsElements

TMArrayAsVector::Find

TMArrayAsVector class

Syntax

```
int Find( const T& t ) const;
```

Description

Finds the specified object and returns the object's index; otherwise returns INT_MAX.

TMArrayAsVector::FirstThat

See Also TMArrayAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the array that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
TMArrayAsVector::LastThat

TMArrayAsVector::Flush

See Also TMArrayAsVector class

Syntax

```
void Flush()
```

Description

Removes all elements from the array without destroying the array.

See Also

TMArrayAsVector::Detach

TMArrayAsVector::ForEach

TMArrayAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMArrayAsVector::GetItemsInContainer

TMArrayAsVector class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the array, as distinguished from ArraySize, which returns the size of the array.

TMArrayAsVector::Grow

TMArrayAsVector class

Syntax

```
void Grow( int loc )
```

Description

Increases the size of the array, in either direction, so that loc is a valid index.

TMArrayAsVector::HasMember

TMArrayAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found in the array; otherwise returns 0.

TMArrayAsVector::InsertEntry

TMArrayAsVector class

Syntax

```
void InsertEntry( int loc )
```

Description

Creates an object and inserts it at loc, moving entries above loc up by one.

TMArrayAsVector::IsEmpty

TMArrayAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the array contains no elements; otherwise returns 0.

TMArrayAsVector::IsFull

TMArrayAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the array is full; otherwise returns 0. The array is full if delta is not equal to 0 and if the number of items in the container equals the value returned by ArraySize.

TMArrayAsVector::LastThat

See Also TMArrayAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the array that satisfies a given condition. You supply a test function pointer, f, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TMArrayAsVector::FirstThat
TMArrayAsVector::ForEach

TMArrayAsVector::LowerBound

TMArrayAsVector class

Syntax

```
int LowerBound() const;
```

Description

Returns the array's lowerbound.

TMArrayAsVector::Reallocate

TMArrayAsVector class

Syntax

```
int Reallocate( unsigned sz, unsigned offset = 0 )
```

Description

If delta (see the constructor) is zero, Reallocate returns 0. Otherwise, Reallocate tries to create a new array of size sz (adjusted upwards to the nearest multiple of delta). The existing array is copied to the expanded array and then deleted. In an array of pointers, the entries are zeroed for each unused element. In an array of objects, the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the array has to be extended downward.

TMArrayAsVector::RemoveEntry

TMArrayAsVector class

Syntax

```
void RemoveEntry( int loc )
```

Description

Removes element at the loc index into the array, and reduces the array by one element. Elements from index (loc + 1) upward are copied to positions loc, (loc + 1), and so on. The original element at loc is lost.

TMArrayAsVector::SetData

TMArrayAsVector class

Syntax

```
void SetData( int loc, const T& t )
```

Description

The given t replaces the existing element at the index loc.

TMArrayAsVector::UpperBound

TMArrayAsVector class

Syntax

```
int UpperBound() const;
```

Description

Returns the array's current upperbound.

TMArrayAsVector::ZeroBase

TMArrayAsVector class

Syntax

```
unsigned ZeroBase( int loc ) const;
```

Description

Returns the location relative to lowerbound (loc - lowerbound).

TMArrayAsVector::ItemAt

TMArrayAsVector class

Syntax

```
T ItemAt( int i ) const;
```

Description

Returns a copy of the object stored at location i.

TMArrayAsVector::operator [ ]

TMArrayAsVector class

Form 1

```
T& operator []( int loc )
```

Form 2

```
T& operator []( int loc ) const;
```

Description

Form 1: Returns a reference to the element at the location specified by loc. The non-const version resizes the array if it's necessary to make loc a valid index.

Form 2: The const throws an exception in the debugging version on an attempt to index out of bounds.

TMArrayAsVectorIterator template

Syntax

```
template <class T, class Alloc> class TMArrayAsVectorIterator;
```

Header File

arrays.h

Description

Implements an iterator object to traverse TMArrayAsVector objects.

Public Constructor

TMArrayAsVectorIterator::TMArrayAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TMArrayAsVectorIterator::TMArrayAsVectorIterator

TMArrayAsVectorIterator class

Syntax

```
TMArrayAsVectorIterator( const TMArrayAsVector<T,Alloc> & a ) :
```

Description

Creates an iterator object to traverse TMArrayAsVector objects.

TMArrayAsVectorIterator::Current

TMArrayAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TMArrayAsVectorIterator::Restart

TMArrayAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Restarts iteration from the beginning, or over the specified range.

TMArrayAsVectorIterator::operator ++

TMArrayAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMArrayAsVectorIterator::operator int

TMArrayAsVectorIterator class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TArrayAsVector template

Syntax

```
template <class T> class TArrayAsVector;
```

Header File

arrays.h

Description

TArrayAsVector implements an array of objects of type T, using a vector as the underlying implementation. TStandardAllocator is used to manage memory.

Type Definitions

CondFunc

IterFunc

Public Constructor

TArrayAsVector::TArrayAsVector

Public Member Functions

Add

AddAt

ArraySize

BoundBase

Destroy

Detach

Find

FirstThat

Flush

ForEach

GetItemsInContainer

Grow

HasMember

InsertEntry

IsEmpty

IsFull

LastThat

LowerBound

Reallocate

RemoveEntry

SetData

UpperBound

ZeroBase

Protected Member Functions

ItemAt

Operators

[]

TArrayAsVector::CondFunc

TArrayAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TArrayAsVector::IterFunc

TArrayAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to the ForEach member function.

TArrayAsVector::TArrayAsVector

TArrayAsVector class

Syntax

```
TArrayAsVector( int upper, int lower = 0, int delta = 0 ) ;
```

Description

Creates an array with an upper bound of upper, a lower bound of lower, and a growth delta of delta.

TArrayAsVector::Add

TArrayAsVector class

Syntax

```
int Add( const T& t )
```

Description

Adds a T object at the next available index at the end of an array. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs and delta is nonzero, the array is expanded (by sufficient multiples of delta bytes) to accommodate the addition. If delta is zero, Add fails. Add returns 0 if it couldn't add the object.

TArrayAsVector::AddAt

TArrayAsVector class

Syntax

```
int AddAt( const T& t, int loc )
```

Description

Adds a T object at the specified index. If that index is occupied, it moves the object up to make room for the added object. If loc is beyond the upper bound, the array is expanded if delta (see the constructor) is nonzero. If delta is zero, attempting to AddAt beyond the upper bound gives an error.

TArrayAsVector::ArraySize

TArrayAsVector class

Syntax

```
unsigned ArraySize() const;
```

Description

Returns the current number of cells allocated.

TArrayAsVector::BoundBase

See Also TArrayAsVector class

Syntax

```
int BoundBase( unsigned loc ) const;
```

Description

Boundbase adjust vectors, which are zero-based, to arrays, which aren't zero-based.

TArrayAsVector::Destroy

TArrayAsVector class

Form 1

```
int Destroy( int i )
```

Form 2

```
int Destroy( const T& t )
```

Description

Form 1: Removes the object at the given index. The object will be destroyed.

Form 2: Removes the given object and destroys it.

TArrayAsVector::Detach

See Also TArrayAsVector class

Form 1

```
int Detach( int loc )
```

Form 2

```
int Detach( const T& t )
```

Description

Form 1: Removes the object at loc.

Form 2: Removes the first object that compares equal to the specified object.

See Also
TShouldDelete::ownsElements

TArrayAsVector::Find

TArrayAsVector class

Syntax

```
int Find( const T& t ) const;
```

Description

Finds the specified object and returns the object's index; otherwise returns INT_MAX.

TArrayAsVector::FirstThat

See Also TArrayAsVector class

Syntax

```
T *FirstThat( cond CondFunc, void *args ) const;
```

Description

Returns a pointer to the first object in the array that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TArrayAsVector::LastThat

TArrayAsVector::Flush

See Also TArrayAsVector class

Syntax

```
void Flush()
```

Description

Removes all elements from the array without destroying the array.

See Also
TArrayAsVector::Detach

TArrayAsVector::ForEach

TArrayAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TArrayAsVector::GetItemsInContainer

TArrayAsVector class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the array, as distinguished from ArraySize, which returns the size of the array.

TArrayAsVector::Grow

TArrayAsVector class

Syntax

```
void Grow( int loc )
```

Description

Increases the size of the array, in either direction, so that loc is a valid index.

TArrayAsVector::HasMember

TArrayAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found in the array; otherwise returns 0.

TArrayAsVector::InsertEntry

TArrayAsVector class

Syntax

```
void InsertEntry( int loc )
```

Description

Creates an object and inserts it at loc, moving entries above loc up by one.

TArrayAsVector::IsEmpty

TArrayAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the array contains no elements; otherwise returns 0.

TArrayAsVector::IsFull

TArrayAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the array is full; otherwise returns 0. The array is full if delta is not equal to 0 and if the number of items in the container equals the value returned by ArraySize.

TArrayAsVector::LastThat

See Also TArrayAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the array that satisfies a given condition. You supply a test function pointer, f, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TArrayAsVector::FirstThat

TArrayAsVector::ForEach

TArrayAsVector::LowerBound

TArrayAsVector class

Syntax

```
int LowerBound() const;
```

Description

Returns the array's lowerbound.

TArrayAsVector::Reallocate

TArrayAsVector class

Syntax

```
int Reallocate( unsigned sz, unsigned offset = 0 )
```

Description

If delta (see the constructor) is zero, Reallocate returns 0. Otherwise, Reallocate tries to create a new array of size sz (adjusted upwards to the nearest multiple of delta). The existing array is copied to the expanded array and then deleted. In an array of pointers, the entries are zeroed for each unused element. In an array of objects, the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the array has to be extended downward.

TArrayAsVector::RemoveEntry

TArrayAsVector class

Syntax

```
void RemoveEntry( int loc )
```

Description

Removes element at the loc index into the array, and reduces the array by one element. Elements from index (loc + 1) upward are copied to positions loc, (loc + 1), and so on. The original element at loc is lost.

TArrayAsVector::SetData

TArrayAsVector class

Syntax

```
void SetData( int loc, const T& t )
```

Description

The given t replaces the existing element at the index loc.

TArrayAsVector::UpperBound

TArrayAsVector class

Syntax

```
int UpperBound() const;
```

Description

Returns the array's current upperbound.

See Also
ZeroBase.

TArrayAsVector::ZeroBase

TArrayAsVector class

Syntax

```
unsigned ZeroBase( int loc ) const;
```

Description

Returns the location relative to lowerbound (loc - lowerbound).

TArrayAsVector::ItemAt

TArrayAsVector class

Syntax

```
T ItemAt( int i ) const;
```

Description

Returns a copy of the object stored at location i.

TArrayAsVector::operator [ ]

TArrayAsVector class

Form 1

```
T& operator []( int loc )
```

Form 2

```
T& operator []( int loc ) const;
```

Description

Form 1: Returns a reference to the element at the location specified by loc. The non-const version resizes the array if it's necessary to make loc a valid index.

Form 2: The const throws an exception in the debugging version on an attempt to index out of bounds.

TArrayAsVectorIterator template

Syntax

```
template <class T> class TArrayAsVectorIterator;
```

Header File

arrays.h

Description

Implements an iterator object to traverse TArrayAsVector objects.

Public Constructor

TArrayAsVectorIterator::TArrayAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TArrayAsVectorIterator::TArrayAsVectorIterator

TArrayAsVectorIterator class

Syntax

```
TArrayAsVectorIterator( const TArrayAsVector<T> & a )
```

Description

Creates an iterator object to traverse TArrayAsVector objects.

TArrayAsVectorIterator::Current

TArrayAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TArrayAsVectorIterator::Restart

TArrayAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Restarts iteration from the beginning, or over the specified range.

TArrayAsVectorIterator::operator ++

TArrayAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TArrayAsVectorIterator::operator int

TArrayAsVectorIterator class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMIArrayAsVector template

Syntax

```
template <class T, class Alloc> class TMIArrayAsVector;
```

Header File

arrays.h

Description

Implements a managed, indirect array of objects of type T, using a vector as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Constructors

TMIArrayAsVector::TMIArrayAsVector

Public Member Functions

Add

AddAt

ArraySize

Destroy

Detach

Find

FirstThat

Flush

ForEach

GetItemsInContainer

HasMember

IsEmpty

IsFull

LastThat

LowerBound

UpperBound

Protected Member Functions

BoundBase

Grow

InsertEntry

ItemAt

Reallocate

RemoveEntry

SetData

SqueezeEntry

ZeroBase

Operators

[]

TMIArrayAsVector::CondFunc

TMIArrayAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMIArrayAsVector::IterFunc

TMIArrayAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMIArrayAsVector::TMIArrayAsVector

TMIArrayAsVector class

Syntax

```
TMIArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Description

Creates an indirect array with an upper bound of upper, a lower bound of lower, and a growth delta of delta.

TMIArrayAsVector::Add

TMIArrayAsVector class

Syntax

```
int Add( T *t )
```

Description

Adds a pointer to a T object at the next available index at the end of an array. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs and delta is nonzero, the array is expanded (by sufficient multiples of delta bytes) to accommodate the addition. If delta is zero, Add fails. Add returns 0 if the object couldn't be added.

TMIArrayAsVector::AddAt

TMIArrayAsVector class

Syntax

```
int AddAt( T *t, int loc )
```

Description

Adds a pointer to a T object at the specified index. If that index is occupied, it moves the object up to make room for the added object. If loc is beyond the upper bound, the array is expanded if delta (see the constructor) is nonzero. If delta is zero, attempting to AddAt beyond the upper bound gives an error. Otherwise it returns 1.

TMIArrayAsVector::ArraySize

TMIArrayAsVector class

Syntax

```
unsigned ArraySize() const;
```

Description

Returns the current number of cells allocated.

TMIArrayAsVector::Destroy

TMIArrayAsVector class

Form 1

```
int Destroy( int i )
```

Form 2

```
int Destroy( T *t )
```

Description

Form 1: Removes the object at the given index. The object will be deleted.

Form 2: Removes the object pointed to by t and deletes it.

TMIArrayAsVector::Detach

See Also TMIArrayAsVector class

Form 1

```
int Detach( T *t, DeleteType dt = NoDelete )
```

Form 2

```
int Detach( int loc, DeleteType dt = NoDelete )
```

Description

Form 1: Removes the object pointer at loc. The value of dt and the current ownership setting determine whether the object itself will be deleted. DeleteType is defined in the base class TShouldDelete as enum { NoDelete, DefDelete, Delete }. The default value of dt, NoDelete, means that the object will not be deleted regardless of ownership. With dt set to Delete, the object will be deleted regardless of ownership. If dt is set to DefDelete, the object will be deleted only if the array owns its elements.

Form 2: Removes the specified pointer. The value of dt and the current ownership setting determine whether the object itself will be deleted. DeleteType is defined in the base class TShouldDelete as enum { NoDelete, DefDelete, Delete }. The default value of dt, NoDelete, means that the object will not be deleted regardless of ownership. With dt set to Delete, the object will be deleted regardless of ownership. If dt is set to DefDelete, the object will be deleted only if the array owns its elements.

See Also
TShouldDelete::ownsElements

TMIArrayAsVector::Find

TMIArrayAsVector class

Syntax

```
int Find( const T *t ) const;
```

Description

Finds the first specified object pointer and returns the index. Returns INT_MAX not found.

TMIArrayAsVector::FirstThat

See Also TMIArrayAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first element in the array that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the container meets the condition.

See Also
TMIArrayAsVector::LastThat

TMIArrayAsVector::Flush

See Also   TMIArrayAsVector class

Syntax

```
void Flush( DeleteType dt = DefDelete )
```

Description

Removes all elements from the array without destroying the array. The value of dt determines whether the elements themselves are destroyed. By default, the ownership status of the array determines their fate, as explained in the Detach member function. You can also set dt to Delete and NoDelete.

See Also
TMIArrayAsVector::Detach

TMIArrayAsVector::ForEach

TMIArrayAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the container. The args argument lets you pass arbitrary data to this function.

TMIArrayAsVector::GetItemsInContainer

TMIArrayAsVector class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the array.

TMIArrayAsVector::HasMember

TMIArrayAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found in the array; otherwise returns 0.

TMIArrayAsVector::IsEmpty

TMIArrayAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the array contains no elements; otherwise returns 0.

TMIArrayAsVector::IsFull

TMIArrayAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the array is full; otherwise returns 0.

TMIArrayAsVector::LastThat

See Also TMIArrayAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last element in the array that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the container meets the condition.

See Also
TMIArrayAsVector::FirstThat
TMIArrayAsVector::ForEach

TMIArrayAsVector::LowerBound

TMIArrayAsVector class

Syntax

```
int LowerBound() const;
```

Description

Returns the array's lowerbound.

TMIArrayAsVector::UpperBound

TMIArrayAsVector class

Syntax

```
int UpperBound() const;
```

Description

Returns the array's current upperbound.

TMIArrayAsVector::BoundBase

See Also TMIArrayAsVector class

Syntax

```
int BoundBase( unsigned loc ) const;
```

Description

Boundbase adjust vectors, which are zero-based, to arrays, which aren't zero-based.

See Also
ZeroBase.

TMIArrayAsVector::Grow

TMIArrayAsVector class

Syntax

```
void Grow( int loc )
```

Description

Increases the size of the array, in either direction, so that loc is a valid index.

TMIArrayAsVector::InsertEntry

TMIArrayAsVector class

Syntax

```
void InsertEntry( int loc )
```

Description

Creates an object and inserts it at loc.

TMIArrayAsVector::ItemAt

TMIArrayAsVector class

Syntax

```
T ItemAt( int i ) const;
```

Description

Returns a copy of the object stored at location i.

TMIArrayAsVector::Reallocate

TMIArrayAsVector class

Syntax

```
int Reallocate( unsigned sz, unsigned offset = 0 )
```

Description

If delta (see the constructor) is zero, Reallocate returns 0. Otherwise, Reallocate tries to create a new array of size sz (adjusted upward to the nearest multiple of delta). The existing array is copied to the expanded array and then deleted. In an array of pointers the entries are zeroed. In an array of objects the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the array has to be extended downward.

TMIArrayAsVector::RemoveEntry

TMIArrayAsVector class

Syntax

```
void RemoveEntry( int loc )
```

Description

Removes element at loc, and reduces the array by one element. Elements from index (loc + 1) upward are copied to positions loc, (loc + 1), and so on. The original element at loc is lost.

TMIArrayAsVector::SetData

TMIArrayAsVector class

Syntax

```
void SetData( int loc, const T& t )
```

Description

The given t replaces the existing element at the index loc.

TMIArrayAsVector::SqueezeEntry

TMIArrayAsVector class

Syntax

```
void SqueezeEntry( unsigned loc )
```

Description

Removes element at loc, and reduces the array by one element. Elements from index (loc + 1) upward are copied to positions loc, (loc + 1), and so on. The original element at loc is lost.

TMIArrayAsVector::ZeroBase

TMIArrayAsVector class

Syntax

```
unsigned ZeroBase( int loc ) const;
```

Description

Returns the location relative to lowerbound (loc - lowerbound).

TMIArrayAsVector::operator [ ]

TMIArrayAsVector class

Form 1

```
T * & operator []( int loc )
```

Form 2

```
T * & operator []( int loc ) const;
```

Description

Form 1: Returns a reference to the element at the location specified by loc. The non-const version resizes the array if it's necessary to make loc a valid index.

Form 2: The const throws an exception in the debugging version on an attempt to index out of bounds.

TMIArrayAsVectorIterator template

Syntax

```
template <class T, class Alloc> class TMIArrayAsVectorIterator;
```

Header File

arrays.h

Description

Implements an iterator object to traverse TMIArrayAsVector objects. Based on TMVectorIteratorImp.

Public Constructor

TMIArrayAsVectorIterator::TMIArrayAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

TMIArrayAsVectorIterator::TMIArrayAsVectorIterator

TMIArrayAsVectorIterator class

Syntax

```
TMIArrayAsVectorIterator( const TMIArrayAsVector<T,Alloc> &a )
```

Description

Creates an iterator object to traverse TMArrayAsVector objects.

TMIArrayAsVectorIterator::Current

TMIArrayAsVectorIterator class

Syntax

```
T *Current();
```

Description

Returns a pointer to the current object.

TMIArrayAsVectorIterator::Restart

TMIArrayAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration from the beginning.

Form 2: Restarts iteration over the specified range.

TMIArrayAsVectorIterator::operator ++

TMIArrayAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TIArrayAsVector template

Syntax

```
template <class T> class TIArrayAsVector;
```

Header File

arrays.h

Description

Implements an indirect array of objects of type T, using a vector as the underlying implementation. TStandardAllocator is used to manage memory.

Public Constructor

TIArrayAsVector

Type Definitions

CondFunc

IterFunc

Public Constructors

TIArrayAsVector::TIArrayAsVector

Public Member Functions

Add

AddAt

ArraySize

BoundBase

Destroy

Detach

Find

FirstThat

Flush

ForEach

GetItemsInContainer

Grow

HasMember

InsertEntry

IsEmpty

IsFull

LastThat

LowerBound

Reallocate

RemoveEntry

SetData

UpperBound

ZeroBase

Protected Member Functions

ItemAt

Operators

[]

TIArrayAsVector::CondFunc

TIArrayAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TIArrayAsVector::IterFunc

TIArrayAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TIArrayAsVector::TIArrayAsVector

TIArrayAsVector class

Syntax

```
TIArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Description

Creates an array with an upper bound of upper, a lower bound of lower, and a growth delta of delta.

TIArrayAsVector::Add

TIArrayAsVector class

Syntax

```
int Add( T *t )
```

Description

Adds a pointer to a T object at the next available index at the end of an array. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs and delta is nonzero, the array is expanded (by sufficient multiples of delta bytes) to accommodate the addition. If delta is zero, Add fails. Add returns 0 if the object couldn't be added.

TIArrayAsVector::AddAt

TIArrayAsVector class

Syntax

```
int AddAt( T *t, int loc )
```

Description

Adds a pointer to a T object at the specified index. If that index is occupied, it moves the object up to make room for the added object. If loc is beyond the upper bound, the array is expanded if delta (see the constructor) is nonzero. If delta is zero, attempting to AddAt beyond the upper bound gives an error.

TIArrayAsVector::ArraySize

TIArrayAsVector class

Syntax

```
unsigned ArraySize() const;
```

Description

Returns the current number of cells allocated.

TIArrayAsVector::BoundBase

See Also TIArrayAsVector class

Syntax

```
int BoundBase( unsigned loc ) const;
```

Description

Boundbase adjust vectors, which are zero-based, to arrays, which aren't zero-based.

See Also
ZeroBase.

TIArrayAsVector::Destroy

TIArrayAsVector class

Form 1

```
int Destroy( int i )
```

Form 2

```
int Destroy( T *t )
```

Description

Form 1: Removes the object at the given index. The object will be deleted.

Form 2: Removes the object pointed to by t and deletes it.

TIArrayAsVector::Detach

See Also TIArrayAsVector class

Form 1

```
int Detach( T *t )
```

Form 2

```
int Detach( int loc )
```

Description

Form 1: Removes the object pointer at loc.

Form 2: Removes the specified pointer.

See Also
TShouldDelete::ownsElements

TIArrayAsVector::Find

TIArrayAsVector class

Syntax

```
int Find( const T *t ) const;
```

Description

Finds the first specified object pointer and returns the index. Returns INT_MAX not found.

TIArrayAsVector::FirstThat

See Also   TIArrayAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first element in the array that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the container meets the condition. Note that FirstThat creates its own internal iterator, so you can treat it as a "search" function.

See Also
TIArrayAsVector::LastThat

TIArrayAsVector::Flush

See Also TIArrayAsVector class

Syntax

```
void Flush()
```

Description

Removes all elements from the array without destroying the array.

See Also
TIArrayAsVector::Detach

TIArrayAsVector::ForEach

TIArrayAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the container. The args argument lets you pass arbitrary data to this function.

TIArrayAsVector::GetItemsInContainer

TIArrayAsVector class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the array.

TIArrayAsVector::Grow

TIArrayAsVector class

Syntax

```
void Grow( int loc )
```

Description

Increases the size of the array, in either direction, so that loc is a valid index.

TIArrayAsVector::HasMember

TIArrayAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found in the array; otherwise returns 0.

TIArrayAsVector::InsertEntry

TIArrayAsVector class

Syntax

```
void InsertEntry( int loc )
```

Description

Creates an object and inserts it at loc.

TIArrayAsVector::IsEmpty

TIArrayAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the array contains no elements; otherwise returns 0.

TIArrayAsVector::IsFull

TIArrayAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the array is full; otherwise returns 0.

TIArrayAsVector::LastThat

See Also TIArrayAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last element in the array that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the container meets the condition.

See Also

TIArrayAsVector::FirstThat

TIArrayAsVector::ForEach

TIArrayAsVector::LowerBound

TIArrayAsVector class

Syntax

```
int LowerBound() const;
```

Description

Returns the array's lowerbound.

TIArrayAsVector::Reallocate

TIArrayAsVector class

Syntax

```
int Reallocate( unsigned sz, unsigned offset = 0 )
```

Description

If delta (see the constructor) is zero, Reallocate returns 0. Otherwise, Reallocate tries to create a new array of size sz (adjusted upward to the nearest multiple of delta). The existing array is copied to the expanded array and then deleted. In an array of pointers the entries are zeroed. In an array of objects the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the array has to be extended downward.

TIArrayAsVector::RemoveEntry

TIArrayAsVector class

Syntax

```
void RemoveEntry( int loc )
```

Description

Removes element at loc, and reduces the array by one element. Elements from index (loc + 1) upward are copied to positions loc, (loc + 1), and so on. The original element at loc is lost.

TIArrayAsVector::SetData

TIArrayAsVector class

Syntax

```
void SetData( int loc, const T& t )
```

Description

The given t replaces the existing element at the index loc.

TIArrayAsVector::SqueezeEntry

TIArrayAsVector class

Syntax

```
void SqueezeEntry( unsigned loc )
```

Description

Removes element at loc, and reduces the array by one element. Elements from index (loc + 1) upward are copied to positions loc, (loc + 1), and so on. The original element at loc is lost.

TIArrayAsVector::UpperBound

TIArrayAsVector class

Syntax

```
int UpperBound() const;
```

Description

Returns the array's current upperbound.

TIArrayAsVector::ZeroBase

TIArrayAsVector class

Syntax

```
unsigned ZeroBase( int loc ) const;
```

Description

Returns the location relative to lowerbound (loc - lowerbound).

TIArrayAsVector::ItemAt

TIArrayAsVector class

Syntax

```
T ItemAt( int i ) const;
```

Description

Returns a copy of the object stored at location i.

TIArrayAsVector::operator [ ]

TIArrayAsVector class

Form 1

```
T * & operator []( int loc )
```

Form 2

```
T * & operator []( int loc ) const;
```

Description

Form 1: Returns a reference to the element at the location specified by loc. The non-const version resizes the array if it's necessary to make loc a valid index.

Form 2: The const throws an exception in the debugging version on an attempt to index out of bounds.

TIArrayAsVectorIterator template

Syntax

```
template <class T> class TIArrayAsVectorIterator;
```

Header File

arrays.h

Description

Implements an iterator object to traverse TIArrayAsVector objects. Uses TStandardAllocator for memory management.

Public Constructors

TIArrayAsVectorIterator::TIArrayAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

TIArrayAsVectorIterator::TIArrayAsVectorIterator

TIArrayAsVectorIterator class

Syntax

```
TIArrayAsVectorIterator( const TIArrayAsVector<T> &a ) :
  TMIArrayAsVectorIterator<T,TStandardAllocator>(a)
```

Description

Creates an iterator object to traverse TIArrayAsVector objects.

TIArrayAsVectorIterator::Current

TIArrayAsVectorIterator class

Syntax

```
T *Current();
```

Description

Returns a pointer to the current object.

TIArrayAsVectorIterator::Restart

TIArrayAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration from the beginning.

Form 2: Restarts iteration over the specified range.

TIArrayAsVectorIterator::operator ++

TIArrayAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMSArrayAsVector template

Syntax

```
template <class T, class Alloc> class TMSArrayAsVector;
```

Header File

arrays.h

Description

Implements a sorted array of objects of type T, using a vector as the underlying implementation. With the exception of the AddAt member function, TMSArrayAsVector inherits its member functions and operators from TMArrayAsVector.

Type Definitions

CondFunc

IterFunc

Public Constructor

TMSArrayAsVector::TMSArrayAsVector

Public Member Functions

Add

AddAt

ArraySize

BoundBase

Destroy

Detach

Find

FirstThat

Flush

ForEach

GetItemsInContainer

Grow

HasMember

InsertEntry

IsEmpty

IsFull

LastThat

LowerBound

Reallocate

RemoveEntry

SetData

UpperBound

ZeroBase

Protected Member Functions

ItemAt

Operators

[]

TMSArrayAsVector::CondFunc

TMSArrayAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMSArrayAsVector::IterFunc

TMSArrayAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to the ForEach member function.

TMSArrayAsVector::TMSArrayAsVector

TMSArrayAsVector class

Syntax

```
TMSArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Description

Creates an array with an upper bound of upper, a lower bound of lower, and a growth delta of delta. It requires a < operator for type T.

TMSArrayAsVector::Add

TMSArrayAsVector class

Syntax

```
int Add( const T& t )
```

Description

Adds a T object at the next available index at the end of an array. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs and delta is nonzero, the array is expanded (by sufficient multiples of delta bytes) to accommodate the addition. If delta is zero, Add fails. Add returns 0 if it couldn't add the object.

TMSArrayAsVector::AddAt

TMSArrayAsVector class

Syntax

```
int AddAt( const T& t, int loc )
```

Description

Adds a T object at the specified index. If that index is occupied, it moves the object up to make room for the added object. If loc is beyond the upper bound, the array is expanded if delta (see the constructor) is nonzero. If delta is zero, attempting to AddAt beyond the upper bound gives an error.

TMSArrayAsVector::ArraySize

TMSArrayAsVector class

Syntax

```
unsigned ArraySize() const;
```

Description

Returns the current number of cells allocated.

TMSArrayAsVector::BoundBase

See Also TMSArrayAsVector class

Syntax

```
int BoundBase( unsigned loc ) const;
```

Description

Boundbase adjust vectors, which are zero-based, to arrays, which aren't zero-based.

See Also
ZeroBase.

TMSArrayAsVector::Destroy

TMSArrayAsVector class

Form 1

```
int Destroy( int i )
```

Form 2

```
int Destroy( const T& t )
```

Description

Form 1: Removes the object at the given index. The object will be destroyed.

Form 2: Removes the given object and destroys it.

TMSArrayAsVector::Detach

See Also TMSArrayAsVector class

Form 1

```
int Detach( int loc )
```

Form 2

```
int Detach( const T& t )
```

Description

Form 1: Removes the object at loc.

Form 2: Removes the first object that compares equal to the specified object.

See Also
TShouldDelete::ownsElements

TMSArrayAsVector::Find

TMSArrayAsVector class

Syntax

```
int Find( const T& t ) const;
```

Description

Finds the specified object and returns the object's index; otherwise returns INT_MAX.

TMSArrayAsVector::FirstThat

See Also TMSArrayAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the array that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TMSArrayAsVector::LastThat

TMSArrayAsVector::Flush

See Also TMSArrayAsVector class

Syntax

```
void Flush()
```

Description

Removes all elements from the array without destroying the array.

See Also
TMSArrayAsVector::Detach

TMSArrayAsVector::ForEach

TMSArrayAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMSArrayAsVector::GetItemsInContainer

TMSArrayAsVector class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the array, as distinguished from ArraySize, which returns the size of the array.

TMSArrayAsVector::Grow

TMSArrayAsVector class

Syntax

```
void Grow( int loc )
```

Description

Increases the size of the array, in either direction, so that loc is a valid index.

TMSArrayAsVector::HasMember

TMSArrayAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found in the array; otherwise returns 0.

TMSArrayAsVector::InsertEntry

TMSArrayAsVector class

Syntax

```
void InsertEntry( int loc )
```

Description

Creates an object and inserts it at loc, moving entries above loc up by one.

TMSArrayAsVector::IsEmpty

TMSArrayAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the array contains no elements; otherwise returns 0.

TMSArrayAsVector::IsFull

TMSArrayAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the array is full; otherwise returns 0. The array is full if delta is not equal to 0 and if the number of items in the container equals the value returned by ArraySize.

TMSArrayAsVector::LastThat

See Also   TMSArrayAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the array that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
TMSArrayAsVector::FirstThat
TMSArrayAsVector::ForEach

TMSArrayAsVector::LowerBound

TMSArrayAsVector class

Syntax

```
int LowerBound() const;
```

Description

Returns the array's lowerbound.

TMSArrayAsVector::Reallocate

TMSArrayAsVector class

Syntax

```
int Reallocate( unsigned sz, unsigned offset = 0 )
```

Description

If delta (see the constructor) is zero, Reallocate returns 0. Otherwise, Reallocate tries to create a new array of size sz (adjusted upwards to the nearest multiple of delta). The existing array is copied to the expanded array and then deleted. In an array of pointers, the entries are zeroed for each unused element. In an array of objects, the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the array has to be extended downward.

TMSArrayAsVector::RemoveEntry

TMSArrayAsVector class

Syntax

```
void RemoveEntry( int loc )
```

Description

Removes element at the loc index into the array, and reduces the array by one element. Elements from index (loc + 1) upward are copied to positions loc, (loc + 1), and so on. The original element at loc is lost.

TMSArrayAsVector::SetData

TMSArrayAsVector class

Syntax

```
void SetData( int loc, const T& t )
```

Description

The given t replaces the existing element at the index loc.

TMSArrayAsVector::UpperBound

TMSArrayAsVector class

Syntax

```
int UpperBound() const;
```

Description

Returns the array's current upperbound.

TMSArrayAsVector::ZeroBase

TMSArrayAsVector class

Syntax

```
unsigned ZeroBase( int loc ) const;
```

Description

Returns the location relative to lowerbound (loc - lowerbound).

TMSArrayAsVector::ItemAt

TMSArrayAsVector class

Syntax

```
T ItemAt( int i ) const;
```

Description

Returns a copy of the object stored at location i.

TMSArrayAsVector::operator [ ]

TMSArrayAsVector class

Form 1

```
T& operator []( int loc )
```

Form 2

```
T& operator []( int loc ) const;
```

Description

Form 1: Returns a reference to the element at the location specified by loc. The non-const version resizes the array if it's necessary to make loc a valid index.

Form 2: The const throws an exception in the debugging version on an attempt to index out of bounds.

TMSArrayAsVectorIterator template

Syntax

```
template <class T, class Alloc> class TMSArrayAsVectorIterator;
```

Header File

arrays.h

Description

Implements an iterator object to traverse TMSArrayAsVector objects.

Public Constructor

TMSArrayAsVectorIterator::TMSArrayAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TMSArrayAsVectorIterator::TMSArrayAsVectorIterator

TMSArrayAsVectorIterator class

Syntax

```
TMSArrayAsVectorIterator( const TMSArrayAsVector<T> & a ) :
```

Description

Creates an iterator object to traverse TSArrayAsVector objects.

TMSArrayAsVectorIterator::Current

TMSArrayAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TMSArrayAsVectorIterator::Restart

TMSArrayAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Restarts iteration from the beginning, or over the specified range.

TMSArrayAsVectorIterator::operator ++

TMSArrayAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMSArrayAsVectorIterator::operator int

TMSArrayAsVectorIterator class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TSArray template

Header File

arrays.h

Description

A simplified name for TSArrayAsVector.

Implements a sorted array of objects of type T, using a vector as the underlying implementation. With the exception of the AddAt member function, TSArray inherits its member functions and operators from TMArrayAsVector.

Type Definitions

CondFunc

IterFunc

Public Constructor

TSArray::TSArray

Public Member Functions

Add

AddAt

ArraySize

BoundBase

Destroy

Detach

Find

FirstThat

Flush

ForEach

GetItemsInContainer

Grow

HasMember

InsertEntry

IsEmpty

IsFull

LastThat

LowerBound

Reallocate

RemoveEntry

SetData

UpperBound

ZeroBase

Protected Member Functions

ItemAt

Operators

[]

TSArray::CondFunc

TSArray class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TSArray::IterFunc

TSArray class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to the ForEach member function.

TSArray::TSArray

TSArray class

Syntax

```
TSArray( int upper, int lower = 0, int delta = 0 )
```

Description

Creates an array with an upper bound of upper, a lower bound of lower, and a growth delta of delta. It requires a < operator for type T.

TSArray::Add

TSArray class

Syntax

```
int Add( const T& t )
```

Description

Adds a T object at the next available index at the end of an array. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs and delta is nonzero, the array is expanded (by sufficient multiples of delta bytes) to accommodate the addition. If delta is zero, Add fails. Add returns 0 if it couldn't add the object.

TSArray::AddAt

TSArray class

Syntax

```
int AddAt( const T& t, int loc )
```

Description

Adds a T object at the specified index. If that index is occupied, it moves the object up to make room for the added object. If loc is beyond the upper bound, the array is expanded if delta (see the constructor) is nonzero. If delta is zero, attempting to AddAt beyond the upper bound gives an error.

TSArray::ArraySize

TSArray class

Syntax

```
unsigned ArraySize() const;
```

Description

Returns the current number of cells allocated.

TSArray::BoundBase

See Also   TSArray class

Syntax

```
int BoundBase( unsigned loc ) const;
```

Description

Boundbase adjust vectors, which are zero-based, to arrays, which aren't zero-based.

See Also

ZeroBase

TSArray::Destroy

TSArray class

Form 1

```
int Destroy( int i )
```

Form 2

```
int Destroy( const T& t )
```

Description

Form 1: Removes the object at the given index. The object will be destroyed.

Form 2: Removes the given object and destroys it.

TSArray::Detach

See Also TSArray class

Form 1

```
int Detach(int loc )
```

Form 2

```
int Detach( const T& t )
```

Description

Form 1: Removes the object at loc.

Form 2: Removes the first object that compares equal to the specified object.

See Also
TShouldDelete::ownsElements

TSArray::Find

TSArray class

Syntax

```
int Find( const T& t ) const;
```

Description

Finds the specified object and returns the object's index; otherwise returns INT_MAX.

TSArray::FirstThat

See Also TSArray class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the array that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TSArray::LastThat

TSArray::Flush

See Also TSArray class

Syntax

```
void Flush()
```

Description

Removes all elements from the array without destroying the array.

See Also
TSArray::Detach

TSArray::ForEach

TSArray class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TSArray::GetItemsInContainer

TSArray class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the array, as distinguished from ArraySize, which returns the size of the array.

TSArray::Grow

TSArray class

Syntax

```
void Grow( int loc )
```

Description

Increases the size of the array, in either direction, so that loc is a valid index.

TSArray::HasMember

TSArray class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found in the array; otherwise returns 0.

TSArray::InsertEntry

TSArray class

Syntax

```
void InsertEntry( int loc )
```

Description

Creates an object and inserts it at loc, moving entries above loc up by one.

TSArray::IsEmpty

TSArray class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the array contains no elements; otherwise returns 0.

TSArray::IsFull

TSArray class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the array is full; otherwise returns 0. The array is full if delta is not equal to 0 and if the number of items in the container equals the value returned by ArraySize.

TSArray::LastThat

See Also TSArray class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the array that satisfies a given condition. You supply a test function pointer, f, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TSArray::FirstThat

TSArray::ForEach

TSArray::LowerBound

TSArray class

Syntax

```
int LowerBound() const;
```

Description

Returns the array's lowerbound.

TSArray::Reallocate

TSArray class

Syntax

```
int Reallocate( unsigned sz, unsigned offset = 0 )
```

Description

If delta (see the constructor) is zero, Reallocate returns 0. Otherwise, Reallocate tries to create a new array of size sz (adjusted upwards to the nearest multiple of delta). The existing array is copied to the expanded array and then deleted. In an array of pointers, the entries are zeroed for each unused element. In an array of objects, the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the array has to be extended downward.

TSArray::RemoveEntry

TSArray class

Syntax

```
void RemoveEntry( int loc )
```

Description

Removes element at the loc index into the array, and reduces the array by one element. Elements from index (loc + 1) upward are copied to positions loc, (loc + 1), and so on. The original element at loc is lost.

TSArray::SetData

TSArray class

Syntax

```
void SetData( int loc, const T& t )
```

Description

The given t replaces the existing element at the index loc.

TSArray::UpperBound

TSArray class

Syntax

```
int UpperBound() const;
```

Description

Returns the array's current upperbound.

TSArray::ZeroBase

TSArray class

Syntax

```
unsigned ZeroBase( int loc ) const;
```

Description

Returns the location relative to lowerbound (loc - lowerbound).

TSArray::ItemAt

TSArray class

Syntax

```
T ItemAt( int i ) const;
```

Description

Returns a copy of the object stored at location i.

TSArray::operator [ ]

TSArray class

Form 1

```
T& operator []( int loc )
```

Form 2

```
T& operator []( int loc ) const;
```

Description

Form 1: Returns a reference to the element at the location specified by loc. The non-const version resizes the array if it's necessary to make loc a valid index.

Form 2: The const throws an exception in the debugging version on an attempt to index out of bounds.

TSArrayAsVector template

Syntax

```
template <class T> class TSArrayAsVector;
```

Header File

arrays.h

Description

Implements a sorted array of objects of type T, using a vector as the underlying implementation. With the exception of the AddAt member function, TSArrayAsVector inherits its member functions and operators from TMArrayAsVector.

Type Definitions

CondFunc

IterFunc

Public Constructor

TSArrayAsVector::TSArrayAsVector

Public Member Functions

Add

AddAt

ArraySize

BoundBase

Destroy

Detach

Find

FirstThat

Flush

ForEach

GetItemsInContainer

Grow

HasMember

InsertEntry

IsEmpty

IsFull

LastThat

LowerBound

Reallocate

RemoveEntry

SetData

UpperBound

ZeroBase

Protected Member Functions

ItemAt

Operators

[]

TSArrayAsVector::CondFunc

TSArrayAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TSArrayAsVector::IterFunc

TSArrayAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to the ForEach member function.

TSArrayAsVector::TSArrayAsVector

TSArrayAsVector class

Syntax

```
TSArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Description

Creates an array with an upper bound of upper, a lower bound of lower, and a growth delta of delta. It requires a < operator for type T.

TSArrayAsVector::Add

TSArrayAsVector class

Syntax

```
int Add( const T& t )
```

Description

Adds a T object at the next available index at the end of an array. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs and delta is nonzero, the array is expanded (by sufficient multiples of delta bytes) to accommodate the addition. If delta is zero, Add fails. Add returns 0 if it couldn't add the object.

TSArrayAsVector::AddAt

TSArrayAsVector class

Syntax

```
int AddAt( const T& t, int loc )
```

Description

Adds a T object at the specified index. If that index is occupied, it moves the object up to make room for the added object. If loc is beyond the upper bound, the array is expanded if delta (see the constructor) is nonzero. If delta is zero, attempting to AddAt beyond the upper bound gives an error.

TSArrayAsVector::ArraySize

TSArrayAsVector class

Syntax

```
unsigned ArraySize() const;
```

Description

Returns the current number of cells allocated.

TSArrayAsVector::BoundBase

See Also TSArrayAsVector class

Syntax

```
int BoundBase( unsigned loc ) const;
```

Description

Boundbase adjust vectors, which are zero-based, to arrays, which aren't zero-based.

TSArrayAsVector::Destroy

TSArrayAsVector class

Form 1

```
int Destroy( int i )
```

Form 2

```
int Destroy( const T& t )
```

Description

Form 1: Removes the object at the given index. The object will be destroyed.

Form 2: Removes the given object and destroys it.

TSArrayAsVector::Detach

See Also TSArrayAsVector class

Form 1

```
int Detach(int loc )
```

Form 2

```
int Detach( const T& t )
```

Description

Form 1: Removes the object at loc.

Form 2: Removes the first object that compares equal to the specified object.

See Also
TShouldDelete::ownsElements

TSArrayAsVector::Find

TSArrayAsVector class

Syntax

```
int Find( const T& t ) const;
```

Description

Finds the specified object and returns the object's index; otherwise returns INT_MAX.

TSArrayAsVector::FirstThat

See Also   TSArrayAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the array that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
TSArrayAsVector::LastThat

TSArrayAsVector::Flush

See Also TSArrayAsVector class

Syntax

```
void Flush()
```

Description

Removes all elements from the array without destroying the array.

See Also
TSArrayAsVector::Detach

TSArrayAsVector::ForEach

TSArrayAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TSArrayAsVector::GetItemsInContainer

TSArrayAsVector class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the array, as distinguished from ArraySize, which returns the size of the array.

TSArrayAsVector::Grow

TSArrayAsVector class

Syntax

```
void Grow( int loc )
```

Description

Increases the size of the array, in either direction, so that loc is a valid index.

TSArrayAsVector::HasMember

TSArrayAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found in the array; otherwise returns 0.

TSArrayAsVector::InsertEntry

TSArrayAsVector class

Syntax

```
void InsertEntry( int loc )
```

Description

Creates an object and inserts it at loc, moving entries above loc up by one.

TSArrayAsVector::IsEmpty

TSArrayAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the array contains no elements; otherwise returns 0.

TSArrayAsVector::IsFull

TSArrayAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the array is full; otherwise returns 0. The array is full if delta is not equal to 0 and if the number of items in the container equals the value returned by ArraySize.

TSArrayAsVector::LastThat

See Also TSArrayAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the array that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TSArrayAsVector::FirstThat

TSArrayAsVector::ForEach

TSArrayAsVector::LowerBound

TSArrayAsVector class

Syntax

```
int LowerBound() const;
```

Description

Returns the array's lowerbound.

TSArrayAsVector::Reallocate

TSArrayAsVector class

Syntax

```
int Reallocate( unsigned sz, unsigned offset = 0 )
```

Description

If delta (see the constructor) is zero, Reallocate returns 0. Otherwise, Reallocate tries to create a new array of size sz (adjusted upwards to the nearest multiple of delta). The existing array is copied to the expanded array and then deleted. In an array of pointers, the entries are zeroed for each unused element. In an array of objects, the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the array has to be extended downward.

TSArrayAsVector::RemoveEntry

TSArrayAsVector class

Syntax

```
void RemoveEntry( int loc )
```

Description

Removes element at the loc index into the array, and reduces the array by one element. Elements from index (loc + 1) upward are copied to positions loc, (loc + 1), and so on. The original element at loc is lost.

TSArrayAsVector::SetData

TSArrayAsVector class

Syntax

```
void SetData( int loc, const T& t )
```

Description

The given t replaces the existing element at the index loc.

TSArrayAsVector::UpperBound

TSArrayAsVector class

Syntax

```
int UpperBound() const;
```

Description

Returns the array's current upperbound.

See Also
ZeroBase.

TSArrayAsVector::ZeroBase

TSArrayAsVector class

Syntax

```
unsigned ZeroBase( int loc ) const;
```

Description

Returns the location relative to lowerbound (loc - lowerbound).

TSArrayAsVector::ItemAt

TSArrayAsVector class

Syntax

```
T ItemAt( int i ) const;
```

Description

Returns a copy of the object stored at location i.

TSArrayAsVector::operator [ ]

TSArrayAsVector class

Form 1

```
T& operator []( int loc )
```

Form 2

```
T& operator []( int loc ) const;
```

Description

Form 1: Returns a reference to the element at the location specified by loc. The non-const version resizes the array if it's necessary to make loc a valid index.

Form 2: The const throws an exception in the debugging version on an attempt to index out of bounds.

TSArrayIterator template

Header File
arrays.h

Description
A simpified name for TSArrayAsVectorIterator. Implements an iterator object to traverse TSArrayAsVector objects.

Public Constructor
TSArrayIterator::TSArrayIterator

Public Member Functions
Current

Restart

Operators
++

int

TSArrayIterator::TSArrayIterator

TSArrayIterator class

Syntax

```
TSArrayIterator( const TSArrayAsVector<T> & a ) :
```

Description

Creates an iterator object to traverse TSArrayAsVector objects.

TSArrayIterator::Current

TSArrayIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TSArrayIterator::Restart

TSArrayIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Restarts iteration from the beginning, or over the specified range.

TSArrayIterator::operator ++

TSArrayIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TSArrayIterator::operator int

TSArrayIterator class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TSArrayAsVectorIterator template

Syntax

```
template <class T> class TSArrayAsVectorIterator;
```

Header File

arrays.h

Description

Implements an iterator object to traverse TSArrayAsVector objects.

Public Constructor

TSArrayAsVectorIterator::TSArrayAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TSArrayAsVectorIterator::TSArrayAsVectorIterator

TSArrayAsVectorIterator class

Syntax

```
TSArrayAsVectorIterator( const TSArrayAsVector<T> & a ) :
```

Description

Creates an iterator object to traverse TSArrayAsVector objects.

TSArrayAsVectorIterator::Current

TSArrayAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TSArrayAsVectorIterator::Restart

TSArrayAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Restarts iteration from the beginning, or over the specified range.

TSArrayAsVectorIterator::operator ++

TSArrayAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TSArrayAsVectorIterator::operator int

TSArrayAsVectorIterator class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TISArrayAsVector template

Syntax

```
template <class T> class TISArrayAsVector;
```

Header File

arrays.h

Description

Implements an indirect sorted array of objects of type T, using a vector as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Constructors

TISArrayAsVector::TISArrayAsVector

Public Member Functions

Add

AddAt

ArraySize

Destroy

Detach

Find

FirstThat

Flush

ForEach

GetItemsInContainer

HasMember

IsEmpty

IsFull

LastThat

LowerBound

UpperBound

Protected Member Functions

BoundBase

Grow

InsertEntry

ItemAt

Reallocate

RemoveEntry

SetData

SqueezeEntry

ZeroBase

Operators

[]

TISArrayAsVector::CondFunc

TISArrayAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TISArrayAsVector::IterFunc

TISArrayAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TISArrayAsVector::TISArrayAsVector

TISArrayAsVector class

Syntax

```
TISArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Description

Creates an indirect array with an upper bound of upper, a lower bound of lower, and a growth delta of delta.

TISArrayAsVector::Add

TISArrayAsVector class

Syntax

```
int Add( T *t )
```

Description

Adds a pointer to a T object at the next available index at the end of an array. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs and delta is nonzero, the array is expanded (by sufficient multiples of delta bytes) to accommodate the addition. If delta is zero, Add fails. Add returns 0 if the object couldn't be added.

TISArrayAsVector::AddAt

TISArrayAsVector class

Syntax

```
int AddAt( T *t, int loc )
```

Description

Adds a pointer to a T object at the specified index. If that index is occupied, it moves the object up to make room for the added object. If loc is beyond the upper bound, the array is expanded if delta (see the constructor) is nonzero. If delta is zero, attempting to AddAt beyond the upper bound gives an error.

TISArrayAsVector::ArraySize

TISArrayAsVector class

Syntax

```
unsigned ArraySize() const;
```

Description

Returns the current number of cells allocated.

TISArrayAsVector::Destroy

TISArrayAsVector class

Form 1

```
int Destroy( int i )
```

Form 2

```
int Destroy( T *t )
```

Description

Form 1: Removes the object at the given index. The object will be deleted.

Form 2: Removes the object pointed to by t and deletes it.

TISArrayAsVector::Detach

See Also TISArrayAsVector class

Form 1

```
int Detach( T *t, DeleteType dt = NoDelete )
```

Form 2

```
int Detach( int loc, DeleteType dt = NoDelete )
```

Description

Form 1: Removes the object pointer at loc. The value of dt and the current ownership setting determine whether the object itself will be deleted. DeleteType is defined in the base class TShouldDelete as enum { NoDelete, DefDelete, Delete }. The default value of dt, NoDelete, means that the object will not be deleted regardless of ownership. With dt set to Delete, the object will be deleted regardless of ownership. If dt is set to DefDelete, the object will be deleted only if the array owns its elements.

Form 2: Removes the specified pointer. The value of dt and the current ownership setting determine whether the object itself will be deleted. DeleteType is defined in the base class TShouldDelete as enum { NoDelete, DefDelete, Delete }. The default value of dt, NoDelete, means that the object will not be deleted regardless of ownership. With dt set to Delete, the object will be deleted regardless of ownership. If dt is set to DefDelete, the object will be deleted only if the array owns its elements.

See Also
TShouldDelete::ownsElements

TISArrayAsVector::Find

TISArrayAsVector class

Syntax

```
int Find( const T *t ) const;
```

Description

Finds the first specified object pointer and returns the index. Returns INT_MAX not found.

TISArrayAsVector::FirstThat

See Also TISArrayAsVector class

Syntax

```
T *FirstThat(CondFunc, void *args ) const;
```

Description

Returns a pointer to the first element in the array that satisfies a given condition. You supply a test-function pointer f that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the container meets the condition. Note that FirstThat creates its own internal iterator, so you can treat it as a "search" function.

See Also
TISArrayAsVector::LastThat

TISArrayAsVector::Flush

See Also TISArrayAsVector class

Syntax

```
void Flush( DeleteType dt = DefDelete )
```

Description

Removes all elements from the array without destroying the array. The value of dt determines whether the elements themselves are destroyed. By default, the ownership status of the array determines their fate, as explained in the Detach member function. You can also set dt to Delete and NoDelete.

See Also
TISArrayAsVector::Detach

TISArrayAsVector::ForEach

TISArrayAsVector class

Syntax

```
void ForEach(IterFunc, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the container. The args argument lets you pass arbitrary data to this function.

TISArrayAsVector::GetItemsInContainer

TISArrayAsVector class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the array.

TISArrayAsVector::HasMember

TISArrayAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found in the array; otherwise returns 0.

TISArrayAsVector::IsEmpty

TISArrayAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the array contains no elements; otherwise returns 0.

TISArrayAsVector::IsFull

TISArrayAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the array is full; otherwise returns 0.

TISArrayAsVector::LastThat

See Also TISArrayAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last element in the array that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the container meets the condition. Note that LastThat creates its own internal iterator, so you can treat it as a "search" function.

See Also

TISArrayAsVector::FirstThat

TISArrayAsVector::ForEach

TISArrayAsVector::LowerBound

TISArrayAsVector class

Syntax

```
int LowerBound() const;
```

Description

Returns the array's lowerbound.

TISArrayAsVector::UpperBound

TISArrayAsVector class

Syntax

```
int UpperBound() const;
```

Description

Returns the array's current upperbound.

TISArrayAsVector::BoundBase

See Also TISArrayAsVector class

Syntax

```
int BoundBase( unsigned loc ) const;
```

Description

Boundbase adjust vectors, which are zero-based, to arrays, which aren't zero-based.

See Also
ZeroBase.

TISArrayAsVector::Grow

TISArrayAsVector class

Syntax

```
void Grow( int loc )
```

Description

Increases the size of the array, in either direction, so that loc is a valid index.

TISArrayAsVector::InsertEntry

TISArrayAsVector class

Syntax

```
void InsertEntry( int loc )
```

Description

Creates an object and inserts it at loc.

TISArrayAsVector::ItemAt

TISArrayAsVector class

Syntax

```
T ItemAt( int i ) const;
```

Description

Returns a copy of the object stored at location i.

TISArrayAsVector::Reallocate

TISArrayAsVector class

Syntax

```
int Reallocate( unsigned sz, unsigned offset = 0 )
```

Description

If delta (see the constructor) is zero, Reallocate returns 0. Otherwise, Reallocate tries to create a new array of size sz (adjusted upward to the nearest multiple of delta). The existing array is copied to the expanded array and then deleted. In an array of pointers the entries are zeroed. In an array of objects the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the array has to be extended downward.

TISArrayAsVector::RemoveEntry

TISArrayAsVector class

Syntax

```
void RemoveEntry( int loc )
```

Description

Removes element at loc, and reduces the array by one element. Elements from index (loc + 1) upward are copied to positions loc, (loc + 1), and so on. The original element at loc is lost.

TISArrayAsVector::SetData

TISArrayAsVector class

Syntax

```
void SetData( int loc, const T& t )
```

Description

The given t replaces the existing element at the index loc.

TISArrayAsVector::SqueezeEntry

TISArrayAsVector class

Syntax

```
void SqueezeEntry( unsigned loc )
```

Description

Removes element at loc, and reduces the array by one element. Elements from index (loc + 1) upward are copied to positions loc, (loc + 1), and so on. The original element at loc is lost.

TISArrayAsVector::ZeroBase

TISArrayAsVector class

Syntax

```
unsigned ZeroBase( int loc ) const;
```

Description

Returns the location relative to lowerbound (loc - lowerbound).

TISArrayAsVector::operator [ ]

TISArrayAsVector class

Form 1

```
T * & operator []( int loc )
```

Form 2

```
T * & operator []( int loc ) const;
```

Description

Form 1: Returns a reference to the element at the location specified by loc. The non-const version resizes the array if it's necessary to make loc a valid index.

Form 2: The const throws an exception in the debugging version on an attempt to index out of bounds.

TISArrayAsVectorIterator template

Syntax

```
template <class T> class TISArrayAsVectorIterator;
```

Syntax

arrays.h

Description

Implements an iterator object to traverse TISArrayAsVector objects.

Public Constructor

TISArrayAsVectorIterator::TISArrayAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TISArrayAsVectorIterator::TISArrayAsVectorIterator

TISArrayAsVectorIterator class

Syntax

```
TISArrayAsVectorIterator( const TISArrayAsVector<T> &a )
```

Description

Creates an iterator object to traverse TISArrayAsVector objects.

TISArrayAsVectorIterator::Current

TISArrayAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TISArrayAsVectorIterator::Restart

TISArrayAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Restarts iteration from the beginning, or over the specified range.

TISArrayAsVectorIterator::operator ++

TISArrayAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TISArrayAsVectorIterator::operator int

TISArrayAsVectorIterator class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMISArrayAsVector template

Syntax

```
template <class T, class Alloc> class TMISArrayAsVector;
```

Header File

arrays.h

Description

Implements a managed, indirect sorted array of objects of type T, using a vector as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Constructors

TMISArrayAsVector::TMISArrayAsVector

Public Member Functions

Add

AddAt

ArraySize

BoundBase

Destroy

Detach

Find

FirstThat

Flush

ForEach

GetItemsInContainer

Grow

HasMember

InsertEntry

IsEmpty

IsFull

LastThat

LowerBound

Reallocate

RemoveEntry

SetData

UpperBound

ZeroBase

Protected Member Functions

ItemAt

Operators

[]

TMISArrayAsVector::CondFunc

TMISArrayAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMISArrayAsVector::IterFunc

TMISArrayAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMISArrayAsVector::TMISArrayAsVector

TMISArrayAsVector class

Syntax

```
TMISArrayAsVector( int upper, int lower = 0, int delta = 0 )
```

Description

Creates an indirect array with an upper bound of upper, a lower bound of lower, and a growth delta of delta.

TMISArrayAsVector::Add

TMISArrayAsVector class

Syntax

```
int Add( T *t )
```

Description

Adds a pointer to a T object at the next available index at the end of an array. Adding an element beyond the upper bound leads to an overflow condition. If overflow occurs and delta is nonzero, the array is expanded (by sufficient multiples of delta bytes) to accommodate the addition. If delta is zero, Add fails. Add returns 0 if the object couldn't be added.

TMISArrayAsVector::AddAt

TMISArrayAsVector class

Syntax

```
int AddAt( T *t, int loc )
```

Description

Adds a pointer to a T object at the specified index. If that index is occupied, it moves the object up to make room for the added object. If loc is beyond the upper bound, the array is expanded if delta (see the constructor) is nonzero. If delta is zero, attempting to AddAt beyond the upper bound gives an error.

TMISArrayAsVector::ArraySize

TMISArrayAsVector class

Syntax

```
unsigned ArraySize() const;
```

Description

Returns the current number of cells allocated.

TMISArrayAsVector::BoundBase

See Also TMISArrayAsVector class

Syntax

```
int BoundBase( unsigned loc ) const;
```

Description

Boundbase adjust vectors, which are zero-based, to arrays, which aren't zero-based.

See Also

ZeroBase

TMISArrayAsVector::Destroy

TMISArrayAsVector class

Form 1

```
int Destroy( int i )
```

Form 2

```
int Destroy( T *t )
```

Description

Form 1: Removes the object at the given index. The object will be deleted.

Form 2: Removes the object pointed to by t and deletes it.

TMISArrayAsVector::Detach

See Also TMISArrayAsVector class

Form 1

```
int Detach( T *t )
```

Form 2

```
int Detach( int loc )
```

Description

Form 1: Removes the object pointer at loc.

Form 2: Removes the specified pointer.

See Also
TShouldDelete::ownsElements

TMISArrayAsVector::FirstThat

See Also		TMISArrayAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first element in the array that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the container meets the condition. Note that FirstThat creates its own internal iterator, so you can treat it as a "search" function.

See Also
TMISArrayAsVector::LastThat

TMISArrayAsVector::Find

TMISArrayAsVector class

Syntax

```
int Find( const T *t ) const;
```

Description

Finds the first specified object pointer and returns the index. Returns INT_MAX not found.

TMISArrayAsVector::Flush

See Also TMISArrayAsVector class

Syntax

```
void Flush()
```

Description

Removes all elements from the array without destroying the array.

See Also
TMISArrayAsVector::Detach

TMISArrayAsVector::ForEach

TMISArrayAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the container. The args argument lets you pass arbitrary data to this function.

TMISArrayAsVector::GetItemsInContainer

TMISArrayAsVector class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the array.

TMISArrayAsVector::Grow

TMISArrayAsVector class

Syntax

```
void Grow( int loc )
```

Description

Increases the size of the array, in either direction, so that loc is a valid index.

TMISArrayAsVector::HasMember

TMISArrayAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found in the array; otherwise returns 0.

TMISArrayAsVector::InsertEntry

TMISArrayAsVector class

Syntax

```
void InsertEntry( int loc )
```

Description

Creates an object and inserts it at loc.

TMISArrayAsVector::IsEmpty

TMISArrayAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the array contains no elements; otherwise returns 0.

TMISArrayAsVector::IsFull

TMISArrayAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the array is full; otherwise returns 0.

TMISArrayAsVector::LastThat

See Also   TMISArrayAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last element in the array that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the container meets the condition. Note that LastThat creates its own internal iterator, so you can treat it as a "search" function.

See Also

TMISArrayAsVector::FirstThat

TMISArrayAsVector::ForEach

TMISArrayAsVector::LowerBound

TMISArrayAsVector class

Syntax

```
int LowerBound() const;
```

Description

Returns the array's lowerbound.

TMISArrayAsVector::Reallocate

TMISArrayAsVector class

Syntax

```
int Reallocate( unsigned sz, unsigned offset = 0 )
```

Description

If delta (see the constructor) is zero, Reallocate returns 0. Otherwise, Reallocate tries to create a new array of size sz (adjusted upward to the nearest multiple of delta). The existing array is copied to the expanded array and then deleted. In an array of pointers the entries are zeroed. In an array of objects the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the array has to be extended downward.

TMISArrayAsVector::RemoveEntry

TMISArrayAsVector class

Syntax

```
void RemoveEntry( int loc )
```

Description

Removes element at loc, and reduces the array by one element. Elements from index (loc + 1) upward are copied to positions loc, (loc + 1), and so on. The original element at loc is lost.

TMISArrayAsVector::SetData

TMISArrayAsVector class

Syntax

```
void SetData( int loc, const T& t )
```

Description

The given t replaces the existing element at the index loc.

TMISArrayAsVector::SqueezeEntry

TMISArrayAsVector class

Syntax

```
void SqueezeEntry( unsigned loc )
```

Description

Removes element at loc, and reduces the array by one element. Elements from index (loc + 1) upward are copied to positions loc, (loc + 1), and so on. The original element at loc is lost.

TMISArrayAsVector::UpperBound

TMISArrayAsVector class

Syntax

```
int UpperBound() const;
```

Description

Returns the array's current upperbound.

TMISArrayAsVector::ZeroBase

TMISArrayAsVector class

Syntax

```
unsigned ZeroBase( int loc ) const;
```

Description

Returns the location relative to lowerbound (loc - lowerbound).

TMISArrayAsVector::ItemAt

TMISArrayAsVector class

Syntax

```
T ItemAt( int i ) const;
```

Description

Returns a copy of the object stored at location i.

TMISArrayAsVector::operator [ ]

TMISArrayAsVector class

Form 1

```
T * & operator []( int loc )
```

Form 2

```
T * & operator []( int loc ) const;
```

Description

Form 1: Returns a reference to the element at the location specified by loc. The non-const version resizes the array if it's necessary to make loc a valid index.

Form 2: The const throws an exception in the debugging version on an attempt to index out of bounds.

TArray template

Header File
arrays.h

Description
A simplified name for TArrayAsVector.

TArrayIterator template

Header File
arrays.h

Description
A simplified name for TArrayAsVectorIterator.

TMDDAssociation template

Syntax

```
template <class K, class V, class A> class TMDDAssociation;
```

Header File

assoc.h

Description

Implements a managed association, binding a direct key (K) with a direct value (V) . Assumes that K has a HashValue member function, or that a global function with one of the following prototypes exists:

- `unsigned HashValue( K );`
- `unsigned HashValue( K & );`
- `unsigned HashValue( const K & );`

K also must have a valid == operator. Class A represents the user-supplied storage manager.

Public Constructors

TMDDAssociation::TMDDAssociation

Public Member Functions

DeleteElements

HashValue

Key

Value

Operators

==

TMDDAssociation::TMDDAssociation

TMDDAssociation class

Form 1

```
TMDDAssociation()
```

Form 2

```
TMDDAssociation( const K &k, const V &v )
```

Description

Form 1: The default constructor.

Form 2: Constructs an object that associates a copy of key object k with a copy of value object v.

TMDDAssociation::DeleteElements

TMDDAssociation class

Syntax

```
void DeleteElements()
```

Description

The dictionary containing the associations determines whether pointed-to objects should be deleted, and if so, calls DeleteElements for each of the associations it holds.

TMDDAssociation::HashValue

TMDDAssociation class

Syntax

```
unsigned HashValue()
```

Description

Returns the hash value for the key.

TMDDAssociation::Key

TMDDAssociation class

Syntax

```
const K& Key()
```

Description

Returns KeyData.

TMDDAssociation::Value

TMDDAssociation class

Syntax

```
const V& Value() const;
```

Description

Returns ValueData.

TMDDAssociation::operator ==

TMDDAssociation class

Syntax

```
int operator ==
```

Description

Tests equality between keys.

TDDAssociation template

Syntax

```
template <class K, class V> class TDDAssociation;
```

Header File

assoc.h

Description

Standard association (direct key, direct value). Implements an association, binding a direct key (K) with a direct value (V). Assumes that K has a HashValue member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

K also must have a valid == operator.

Public Constructors

TDDAssociation

Public Member Functions

DeleteElements

HashValue

Key

Value

Operators

==

TDDAssociation::TDDAssociation

TDDAssociation class

Form 1

```
TDDAssociation()
```

Form 2

```
TDDAssociation( const K &k, const V &v )
```

Description

Form 1: The default constructor.

Form 2: Constructs an object that associates key object k with value object v.

TDDAssociation::DeleteElements

TDDAssociation class

Syntax

```
void DeleteElements()
```

Description

The dictionary containing the associations determines whether pointed-to objects should be deleted, and if so, calls DeleteElements for each of the associations it holds.

TDDAssociation::HashValue

TDDAssociation class

Syntax

```
unsigned HashValue()
```

Description

Returns the hash value for the key.

TDDAssociation::Key

TDDAssociation class

Syntax

```
const K& Key()
```

Description

Returns KeyData.

TDDAssociation::Value

TDDAssociation class

Syntax

```
const V& Value() const;
```

Description

Returns ValueData.

TDDAssociation::operator ==

TDDAssociation class

Syntax

```
int operator == (const TDDAssociation<K,V,A> & a)
```

Description

Tests equality between keys.

TMDIAssociation template

Syntax

```
template <class K, class V, class A> class TMDIAssociation;
```

Header File

assoc.h

Description

Implements a managed association, binding a direct key (K) with a indirect value (V) . Assumes that K has a HashValue member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

K also must have a valid == operator. Class A represents the user-supplied storage manager.

Public Constructors

TMDIAssociation::TMDIAssociation

Public Member Functions

HashValue

Key

Value

Operators

==

TMDIAssociation::TMDIAssociation

TMDIAssociation class

Form 1

```
TMDIAssociation()
```

Form 2

```
TMDIAssociation( const K& k, V * v )
```

Description

Form 1: The default constructor.

Form 2: Constructs an object that associates key object k with value object v.

TMDIAssociation::HashValue

TMDIAssociation class

Syntax

```
unsigned HashValue()
```

Description

Returns the hash value for the key.

TMDIAssociation::Key

TMDIAssociation class

Syntax

```
const K& Key()
```

Description

Returns the key.

TMDIAssociation::Value

TMDIAssociation class

Syntax

```
const V * Value()
```

Description

Returns a pointer to the data.

TMDIAssociation::operator ==

TMDIAssociation class

Syntax

```
int operator == (const TMDDAssociation<K,V,A> & a)
```

Description

Tests the equality between keys.

TDIAssociation template

Syntax

```
template <class K, class V> class TDIAssociation;
```

Header File

assoc.h

Description

Implements an association, binding a direct key (K) with an indirect value (V). Assumes that K has a HashValue member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

K also must have a valid == operator.

Public Constructors

TDIAssociation::TDIAssociation

Public Member Functions

HashValue

Key

Value

Operators

==

TDIAssociation::TDIAssociation

TDIAssociation class

Form 1

```
TDIAssociation()
```

Form 2

```
TDIAssociation( const K& k, V * v )
```

Description

Form 1: The default constructor.

Form 2: Constructs an object that associates key object k with value object v.

TDIAssociation::HashValue

TDIAssociation class

Syntax

```
unsigned HashValue()
```

Description

Returns the hash value for the key.

TDIAssociation::Key

TDIAssociation class

Syntax

```
const K& Key()
```

Description

Returns the key.

TDIAssociation::Value

TDIAssociation class

Syntax

```
const V * Value()
```

Description

Returns a pointer to the data.

TDIAssociation::operator ==

TDIAssociation class

Syntax

```
int operator == (const TDIAssociation<K,V,A> & a)
```

Description

Tests the equality between keys.

TMIDAssociation template

Syntax

```
template <class K, class V, class A> class TMIDAssociation;
```

Header File

assoc.h

Description

Implements a managed association, binding an indirect key (K) with a direct value (V) . Assumes that K has a HashValue member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

K also must have a valid == operator. Class A represents the user-supplied storage manager.

Protected Data Members

KeyData

ValueData

Public Constructors

TMIDAssociation::TMIDAssociation

Public Member Functions

DeleteElements

HashValue

Key

Value

Operators

==

TMIDAssociation::KeyData

TMIDAssociation class

Syntax

```
K KeyData;
```

Description

The key class passed into the template by the user.

TMIDAssociation::ValueData

TMIDAssociation class

Syntax

```
V ValueData;
```

Description

The value class passed into the template by the user.

TMIDAssociation::TMIDAssociation

TMIDAssociation class

Form 1

```
TMIDAssociation()
```

Form 2

```
TMIDAssociation( K *k, const V& v )
```

Description

Form 1: The default constructor.

Form 2: Constructs an object that associates key object k with value object v.

TMIDAssociation::DeleteElements

TMIDAssociation class

Syntax

```
void DeleteElements()
```

Description

The dictionary containing the associations determines whether pointed-to objects should be deleted, and if so, calls DeleteElements for each of the associations it holds.

TMIDAssociation::HashValue

TMIDAssociation class

Syntax

```
unsigned HashValue()
```

Description

Returns the hash value for the key.

TMIDAssociation::Key

TMIDAssociation class

Syntax

```
const K * Key()
```

Description

Returns a pointer to the key.

TMIDAssociation::Value

TMIDAssociation class

Syntax

```
const V& Value() const;
```

Description

Returns a copy of the data.

TMIDAssociation::operator ==

TMIDAssociation class

Syntax

```
int operator == (const TMIDAssociation<K,V,A> & a)
```

Description

Tests the equality between keys.

TIDAssociation template

Syntax

```
template <class K, class V> class TIDAssociation;
```

Header File

assoc.h

Description

Implements an association, binding an indirect key (K) with a direct value (V) . Assumes that K has a HashValue member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

K also must have a valid == operator.

Public Constructors

TIDAssociation::TIDAssociation

Protected Data Members

KeyData

ValueData

Public Member Functions

DeleteElements

HashValue

Key

Value

Operators

==

TIDAssociation::KeyData

TIDAssociation class

Syntax

```
K KeyData;
```

Description

The key class passed into the template by the user.

TIDAssociation::ValueData

TIDAssociation class

Syntax

```
V ValueData;
```

Description

The value class passed into the template by the user.

TIDAssociation::TIDAssociation

TIDAssociation class

Form 1

```
TIDAssociation()
```

Form 2

```
TIDAssociation( K * k, V v )
```

Description

Form 1: The default constructor.

Form 2: Constructs an object that associates key object *k with value object v.

TIDAssociation::DeleteElements

TIDAssociation class

Syntax

```
void DeleteElements()
```

Description

The dictionary containing the associations determines whether pointed-to objects should be deleted, and if so, calls DeleteElements for each of the associations it holds.

TIDAssociation::HashValue

TIDAssociation class

Syntax

```
unsigned HashValue()
```

Description

Returns the hash value for the key.

TIDAssociation::Key

TIDAssociation class

Syntax

```
const K * Key()
```

Description

Returns a pointer to the key.

TIDAssociation::Value

TIDAssociation class

Syntax

```
const V& Value() const;
```

Description

Returns a copy of the data.

TIDAssociation::operator ==

TIDAssociation class

Syntax

```
int operator == (const TIDAssociation<K,V,A> & a)
```

Description

Tests the equality between keys.

TMIIAssociation template

Syntax

```
template <class K, class V, class A> class TMIIAssociation;
```

Header File

assoc.h

Description

Implements a managed association, binding an indirect key (K) with an indirect value (V) . Assumes that K has a HashValue member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

K also must have a valid == operator. Class A represents the user-supplied storage manager.

Public Constructors

TMIIAssociation::TMIIAssociation

Public Member Functions

DeleteElements

HashValue

Key

Value

Operators

==

TMIIAssociation::TMIIAssociation

TMIIAssociation class

Form 1

```
TMIIAssociation()
```

Form 2

```
TMIIAssociation( K * k, V * v )
```

Description

Form 1: The default constructor.

Form 2: Constructs an object that associates key object *k with value object *v.

TMIIAssociation::DeleteElements

TMIIAssociation class

Syntax

```
void DeleteElements()
```

Description

The dictionary containing the associations determines whether pointed-to objects should be deleted, and if so, calls DeleteElements for each of the associations it holds.

TMIIAssociation::HashValue

TMIIAssociation class

Syntax

```
unsigned HashValue()
```

Description

Returns the hash value for the key.

TMIIAssociation::Key

TMIIAssociation class

Syntax

```
const K * Key()
```

Description

Returns a pointer to the key.

TMIIAssociation::Value

TMIIAssociation class

Syntax

```
const V * Value()
```

Description

Returns a pointer to the data.

TMIIAssociation::operator ==

TMIIAssociation class

Syntax

```
int operator == (const TMIIAssociation<K,V,A> & a)
```

Description

Tests equality between keys.

TIIAssociation template

Syntax

```
template <class K, class V> class TIIAssociation;
```

Header File

assoc.h

Description

Standard association (indirect key, indirect value). Implements an association, binding an indirect key (K) with an indirect value (V) . Assumes that K has a HashValue member function, or that a global function with the following prototype exists:

```
unsigned HashValue( K & );
```

K also must have a valid == operator.

Public Constructors

TIIAssociation::TIIAssociation

Public Member Functions

DeleteElements

HashValue

Key

Value

Operators

==

TIIAssociation::TIIAssociation

TIIAssociation class

Form 1
```
TIIAssociation()
```

Form 2
```
TIIAssociation( K *k, V * v )
```

Description

Form 1: The default constructor.

Form 2: Constructs an object that associates key object *k with value object *v.

TIIAssociation::DeleteElements

TIIAssociation class

Syntax

```
void DeleteElements()
```

Description

The dictionary containing the associations determines whether pointed-to objects should be deleted, and if so, calls DeleteElements for each of the associations it holds.

TIIAssociation::HashValue

TIIAssociation class

Syntax

```
unsigned HashValue()
```

Description

Returns the hash value for the key.

TIIAssociation::Key

TIIAssociation class

Syntax

```
const K * Key()
```

Description

Returns a pointer to the key.

TIIAssociation::Value

TIIAssociation class

Syntax

```
const V * Value()
```

Description

Returns a pointer to the data.

TIIAssociation::operator ==

TIIAssociation class

Syntax

```
int operator == (const TIIAssociation<K,V,A> & a)
```

Description

Tests equality between keys.

TMBagAsVector template

Syntax

```
template <class T, class Alloc> class TMBagAsVector;
```

Header File

bags.h

Description

Implements a managed bag of objects of type T, using a vector as the underlying implementation. Bags, unlike sets, can contain duplicate objects.

Type Definitions

CondFunc

IterFunc

Public Constructors

TMBagAsVector

Public Member Functions

Add

Detach

Find

FindMember

Flush

ForEach

GetItemsInContainer

HasMember

isEmpty

isFull

TMBagAsVector::CondFunc

TMBagAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMBagAsVector::IterFunc

TMBagAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMBagAsVector::TMBagAsVector

TMBagAsVector class

Syntax

```
TMBagAsVector( unsigned sz = DEFAULT_BAG_SIZE )
```

Description

Constructs a managed, empty bag. sz represents the number of items the bag can hold.

TMBagAsVector::Add

TMBagAsVector class

Syntax

```
int Add( const T& t )
```

Description

Adds the given object to the bag.

TMBagAsVector::Detach

See Also TMBagAsVector class

Syntax

```
int Detach( const T& t )
```

Description

Removes the specified object.

See Also
TShouldDelete::ownsElements

TMBagAsVector::Find

TMBagAsVector class

Syntax

```
T *Find( const T& t ) const;
```

Description

Returns a pointer to the given object if found; otherwise returns 0.

TMBagAsVector::FindMember

TMBagAsVector class

Syntax

```
T* FindMember( const T& t ) const;
```

Description

Returns a pointer to the given object if found; otherwise returns 0.

TMBagAsVector::Flush

See Also   TMBagAsVector class

Syntax

```
void Flush()
```

Description

Removes all the elements from the bag without destroying the bag.

See Also
Detach

TMBagAsVector::ForEach

TMBagAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the bag. The args argument lets you pass arbitrary data to this function.

TMBagAsVector::GetItemsInContainer

TMBagAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of objects in the bag.

TMBagAsVector::HasMember

TMBagAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found; otherwise returns 0.

TMBagAsVector::IsEmpty

TMBagAsVector class

Syntax

```
int isEmpty() const;
```

Description

Returns 1 if the bag is empty; otherwise returns 0.

TMBagAsVector::IsFull

TMBagAsVector class

Syntax

```
int isFull() const;
```

Description

Returns 0.

TMBagAsVectorIterator template

Syntax

```
template <class T, class Alloc> class TMBagAsVectorIterator;
```

Header File

bags.h

Description

Implements an iterator object to traverse TMBagAsVector objects.

Public Constructors

TMBagAsVectorIterator::TMBagAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TMBagAsVectorIterator::TMBagAsVectorIterator

TMBagAsVectorIterator class

Syntax

```
TMBagAsVectorIterator( const TMBagAsVector<T,Alloc> & b )
```

Description

Constructs an object that iterates on TMBagAsVector objects.

TMBagAsVectorIterator::Current

TMBagAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TMBagAsVectorIterator::Restart

TMBagAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Restarts iteration from the beginning, or over the specified range.

TMBagAsVectorIterator::operator ++

TMBagAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMBagAsVectorIterator::operator int

TMBagAsVectorIterator class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TBagAsVector template

Syntax

```
template <class T> class TBagAsVector;
```

Header File

bags.h

Description

Implements a bag of objects of type T, using a vector as the underlying implementation. TStandardAllocator is used to manage memory.

Type Definitions

CondFunc

IterFunc

Public Constructors

TBagAsVector::TBagAsVector

Public Member Functions

Add

Detach

Find

FindMember

Flush

ForEach

GetItemsInContainer

HasMember

isEmpty

isFull

TBagAsVector::CondFunc

TBagAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TBagAsVector::IterFunc

TBagAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TBagAsVector::TBagAsVector

TBagAsVector class

Syntax

```
TBagAsVector( unsigned sz = DEFAULT_BAG_SIZE )
```

Description

Constructs an empty bag. sz represents the number of items the bag can hold.

TBagAsVector::Add

TBagAsVector class

Syntax

```
int Add( const T& t )
```

Description

Adds the given object to the bag.

TBagAsVector::Detach

See Also TBagAsVector class

Syntax

```
int Detach( const T& t )
```

Description

Removes the specified object.

See Also
TShouldDelete::ownsElements

TBagAsVector::Find

TBagAsVector class

Form 1

```
T *Find( const T& t ) const;
```

Description

Returns a pointer to the given object if found; otherwise returns 0.

TBagAsVector::FindMember

TBagAsVector class

Syntax

```
T* FindMember( const T& t ) const
```

Description

Returns a pointer to the given object if found; otherwise returns 0.

TBagAsVector::Flush

See Also TBagAsVector class

Syntax

```
void Flush()
```

Description

Removes all the elements from the bag without destroying the bag.

See Also
[Detach]

TBagAsVector::ForEach

TBagAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the bag. The args argument lets you pass arbitrary data to this function.

TBagAsVector::GetItemsInContainer

TBagAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of objects in the bag.

TBagAsVector::HasMember

TBagAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found; otherwise returns 0.

TBagAsVector::IsEmpty

TBagAsVector class

Syntax

```
int isEmpty() const;
```

Description

Returns 1 if the bag is empty; otherwise returns 0.

TBagAsVector::IsFull

TBagAsVector class

Syntax

```
int isFull() const;
```

Description

Returns 0.

TBagAsVectorIterator template

Syntax

```
template <class T> class TBagAsVectorIterator;
```

Header File

bags.h

Description

Implements an iterator object to traverse TBagAsVector objects. TStandardAllocator is used to manage memory.

Public Constructors

TBagAsVectorIterator::TBagAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TBagAsVectorIterator::TBagAsVectorIterator

TBagAsVectorIterator class

Syntax

```
TBagAsVectorIterator( const TBagAsVector<T> & b )
```

Description

Constructs an object that iterates on TBagAsVector objects.

TBagAsVectorIterator::Current

TBagAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TBagAsVectorIterator::Restart

TBagAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Restarts iteration from the beginning, or over the specified range.

TBagAsVectorIterator::operator ++

TBagAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TBagAsVectorIterator::operator int

TBagAsVectorIterator class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMIBagAsVector template

Syntax

```
template <class T, class Alloc> class TMIBagAsVector;
```

Header File

bags.h

Description

Implements a managed bag of pointers to objects of type T, using a vector as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Constructors

TMIBagAsVector::TMIBagAsVector

Public Member Functions

Add

Detach

Find

FirstThat

Flush

ForEach

GetItemsInContainer

HasMember

isEmpty

isFull

LastThat

TMIBagAsVector::CondFunc

TMIBagAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMIBagAsVector::IterFunc

TMIBagAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMIBagAsVector::TMIBagAsVector

TMIBagAsVector class

Syntax

```
TMIBagAsVector( unsigned sz = DEFAULT_BAG_SIZE )
```

Description

Constructs an empty, managed, indirect bag. sz represents the initial number of slots allocated.

TMIBagAsVector::Add

TMIBagAsVector class

Syntax

```
int Add( T *t )
```

Description

Adds the given object pointer to the bag.

TMIBagAsVector::Detach

See Also   TMIBagAsVector class

Syntax

```
int Detach( T *t, DeleteType dt = NoDelete )
```

Description

Removes the specified object pointer. The value of dt and the current ownership setting determine whether the object itself will be deleted. DeleteType is defined in the base class TShouldDelete as enum { NoDelete, DefDelete, Delete }. The default value of dt, NoDelete, means that the object will not be deleted regardless of ownership. With dt set to Delete, the object will be deleted regardless of ownership. If dt is set to DefDelete, the object will only be deleted if the bag owns its elements.

See Also
TShouldDelete::ownsElements

TMIBagAsVector::Find

TMIBagAsVector class

Syntax

```
T *Find( T *t ) const;
```

Description

Returns a pointer to the given object if found; otherwise returns 0.

TMIBagAsVector::FirstThat

TMIBagAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the bag that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

TMIBagAsVector::Flush

See Also TMIBagAsVector class

Syntax

```
void Flush( TShouldDelete::DeleteType dt = TShouldDelete::DefDelete )
```

Description

Removes all the elements from the bag without destroying the bag. The value of dt determines whether the elements themselves are destroyed. By default, the ownership status of the bag determines their fate, as explained in the Detach member function. You can also set dt to Delete and NoDelete.

See Also
[Detach]

TMIBagAsVector::ForEach

TMIBagAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the bag. The args argument lets you pass arbitrary data to this function.

TMIBagAsVector::GetItemsInContainer

TMIBagAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of objects in the bag.

TMIBagAsVector::HasMember

TMIBagAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found; otherwise returns 0.

TMIBagAsVector::IsEmpty

TMIBagAsVector class

Syntax

```
int isEmpty() const;
```

Description

Returns 1 if the bag is empty; otherwise returns 0.

TMIBagAsVector::IsFull

TMIBagAsVector class

Syntax

```
int isFull() const;
```

Description

Returns 0.

TMIBagAsVector::LastThat

TMIBagAsVector class

Syntax

```
T *LastThat(CondFunc cond, void *args ) const
```

Description

Returns a pointer to the last object in the bag that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the bag meets the condition.

TMIBagAsVectorIterator template

Syntax

```
template <class T, class Alloc> class TMIBagAsVectorIterator;
```

Header File

bags.h

Description

Implements an iterator object to traverse TMIBagAsVector objects.

Public Constructors

TMIBagAsVectorIterator::TMIBagAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TMIBagAsVectorIterator::TMIBagAsVectorIterator

TMIBagAsVectorIterator class

Syntax

```
TMIBagAsVectorIterator( const TMIBagAsVector<T,Alloc> & b )
```

Description

Constructs an object that iterates on TMBagAsVector objects.

TMIBagAsVectorIterator::Current

TMIBagAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TMIBagAsVectorIterator::Restart

TMIBagAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Restarts iteration from the beginning, or over the specified range.

TMIBagAsVectorIterator::operator ++

TMIBagAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMIBagAsVectorIterator::operator int

TMIBagAsVectorIterator class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TIBagAsVector template

Syntax

```
template <class T> class TIBagAsVector;
```

Header File

bags.h

Description

Implements a bag of pointers to objects of type T, using a vector as the underlying implementation. TStandardAllocator is used to manage memory.

Public Constructors

TIBagAsVector::TIBagAsVector

Type Definitions

CondFunc

IterFunc

Public Member Functions

Add

Detach

Find

FirstThat

Flush

ForEach

GetItemsInContainer

HasMember

isEmpty

isFull

LastThat

TIBagAsVector::CondFunc

TIBagAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TIBagAsVector::IterFunc

TIBagAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TIBagAsVector::TIBagAsVector

TIBagAsVector class

Syntax

```
TIBagAsVector( unsigned sz = DEFAULT_BAG_SIZE )
```

Description

Constructs an empty, managed, indirect bag. sz represents the initial number of slots allocated.

TIBagAsVector::Add

TIBagAsVector class

Syntax

```
int Add( T *t )
```

Description

Adds the given object pointer to the bag.

TIBagAsVector::Detach

See Also TIBagAsVector class

Syntax

```
int Detach( T *t, DeleteType dt = NoDelete )
```

Description

Removes the specified object pointer. The value of dt and the current ownership setting determine whether the object itself will be deleted. DeleteType is defined in the base class TShouldDelete as enum { NoDelete, DefDelete, Delete }. The default value of dt, NoDelete, means that the object will not be deleted regardless of ownership. With dt set to Delete, the object will be deleted regardless of ownership. If dt is set to DefDelete, the object will only be deleted if the bag owns its elements.

See Also
TShouldDelete::ownsElements

TIBagAsVector::Find

TIBagAsVector class

Syntax

```
T *Find( T *t ) const;
```

Description

Returns a pointer to the given object if found; otherwise returns 0.

TIBagAsVector::FirstThat

TIBagAsVector class

Syntax

```
T *FirstThat(CondFunc, void *args ) const;
```

Description

Returns a pointer to the first object in the bag that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the bag meets the condition.

TIBagAsVector::Flush

See Also TIBagAsVector class

Syntax

```
void Flush( TShouldDelete::DeleteType dt = TShouldDelete::DefDelete )
```

Description

Removes all the elements from the bag without destroying the bag. The value of dt determines whether the elements themselves are destroyed. By default, the ownership status of the bag determines their fate, as explained in the Detach member function. You can also set dt to Delete and NoDelete.

See Also
Detach

TIBagAsVector::ForEach

TIBagAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the bag. The args argument lets you pass arbitrary data to this function.

TIBagAsVector::GetItemsInContainer

TIBagAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of objects in the bag.

TIBagAsVector::HasMember

TIBagAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found; otherwise returns 0.

TIBagAsVector::IsEmpty

TIBagAsVector class

Syntax

```
int isEmpty() const;
```

Description

Returns 1 if the bag is empty; otherwise returns 0.

TIBagAsVector::IsFull

TIBagAsVector class

Syntax

```
int isFull() const;
```

Description

Returns 0.

TIBagAsVector::LastThat

TIBagAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the bag that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

TIBagAsVectorIterator template

Syntax
```
template <class T> class TIBagAsVectorIterator;
```

Header File
bags.h

Description
Implements an iterator object to traverse TIBagAsVector objects. TStandardAllocator is used to manage memory.

Public Constructors
TIBagAsVectorIterator::TIBagAsVectorIterator

Public Member Functions
Current

Restart

Operators
++

int

TIBagAsVectorIterator::TIBagAsVectorIterator

TIBagAsVectorIterator class

Syntax

```
TIBagAsVectorIterator( const TIBagAsVector<T> & s )
```

Description

Constructs an object that iterates on TIBagAsVector objects.

TIBagAsVectorIterator::Current

TIBagAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TIBagAsVectorIterator::Restart

TIBagAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Restarts iteration from the beginning, or over the specified range.

TIBagAsVectorIterator::operator ++

TIBagAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TIBagAsVectorIterator::operator int

TIBagAsVectorIterator class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TBinarySearchTreeImp template

Syntax

```
template <class T> class TBinarySearchTreeImp;
```

Header File

binimp.h

Description

Implements an unbalanced binary tree. Class T must have < and == operators, and must have a default constructor.

Public Member Functions

Add

Detach

Find

Flush

ForEach

GetItemsInContainer

IsEmpty

Protected Member Functions

EqualTo

LessThan

DeleteNode

TBinarySearchTreeImp::Add

TBinarySearchTreeImp class

Syntax

```
int Add( const T& t )
```

Description

Creates a new binary-tree node and inserts a copy of object t into it.

TBinarySearchTreeImp::Detach

TBinarySearchTreeImp class

Syntax

```
int Detach( const T& t )
```

Description

Removes the node containing item t from the tree.

TBinarySearchTreeImp::Find

TBinarySearchTreeImp class

Syntax

```
T * Find( const T& t ) const;
```

Description

Returns a pointer to the node containing item t.

TBinarySearchTreeImp::Flush

TBinarySearchTreeImp class

Syntax

```
void Flush();
```

Description

Removes all items from the tree.

TBinarySearchTreeImp::ForEach

TBinarySearchTreeImp class

Syntax

```
void ForEach( IterFunc iter, void * args, IteratorOrder order = InOrder )
```

Description

Creates an internal iterator that executes the given function iter for each item in the container. The args argument lets you pass arbitrary data to this function.

TBinarySearchTreeImp::GetItemsInContainer

TBinarySearchTreeImp class

Syntax

```
unsigned GetItemsInContainer();
```

Description

Returns the number of items in the tree.

TBinarySearchTreeImp::IsEmpty

TBinarySearchTreeImp class

Syntax

```
int IsEmpty();
```

Description

Returns 1 if the tree is empty; otherwise returns 0.

TBinarySearchTreeImp::EqualTo

TBinarySearchTreeImp class

Syntax

```
virtual int EqualTo( BinNode *n1, BinNode *n2 )
```

Description

Tests the equality between two nodes.

TBinarySearchTreeImp::LessThan

TBinarySearchTreeImp class

Syntax

```
virtual int LessThan( BinNode *n1, BinNode *n2 )
```

Description

Tests if node n1 is less than node n2.

TBinarySearchTreeImp::DeleteNode

TBinarySearchTreeImp class

Syntax

```
virtual void DeleteNode( BinNode *node, int del)
```

Description

Deletes node. The second parameter is ignored.

TBinarySearchTreeIteratorImp template

Syntax

```
template <class T> class TBinarySearchTreeIteratorImp;
```

Header File

binimp.h

Description

Implements an iterator that traverses TBinarySearchTreeImp objects.

Public Constructors

TBinarySearchTreeIteratorImp::TBinarySearchTreeIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

TBinarySearchTreeIteratorImp::TBinarySearchTreeIteratorImp

TBinarySearchTreeIteratorImp class

Syntax

```
TBinarySearchTreeIteratorImp( TBinarySearchTreeImp<T>& tree,
  TBinarySearchTreeBase::IteratorOrder order =
  TBinarySearchTreeBase::InOrder )
```

Description

Constructs an iterator object that traverses a TBinarySearchTreeImp container.

TBinarySearchTreeIteratorImp::Current

TBinarySearchTreeIteratorImp class

Syntax

```
const T& Current() const;
```

Description

Returns the current object.

TBinarySearchTreeIteratorImp::Restart

TBinarySearchTreeIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the tree.

TBinarySearchTreeIteratorImp::operator int

TBinarySearchTreeIteratorImp class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TBinarySearchTreeIteratorImp::operator ++

TBinarySearchTreeIteratorImp class

Syntax

```
const T& operator ++ ( int );
```

Description

Moves to the next object in the tree, and returns the object that was current before the move (post-increment).

const T& operator ++ ()

Moves to the next object, and returns the object that was current after the move (pre-increment).

TIBinarySearchTreeImp template

Syntax

```
template <class T> class TIBinarySearchTreeImp;
```

Header File

binimp.h

Description

Implements an indirect unbalanced binary tree. Class T must have < and == operators, and must have a default constructor.

Public Member Functions

Add

Detach

Find

Flush

ForEach

GetItemsInContainer

IsEmpty

Protected Member functions

EqualTo

LessThan

DeleteNode

TIBinarySearchTreeImp::Add

TIBinarySearchTreeImp class

Syntax

```
int Add( T * t )
```

Description

Creates a new binary-tree node and inserts a pointer to object t into the tree.

TIBinarySearchTreeImp::Detach

TIBinarySearchTreeImp class

Syntax

```
int Detach( T * t, int del = 0 )
```

Description

Removes the node containing item t from the tree. The item is deleted if del is 1.

TIBinarySearchTreeImp::Find

TIBinarySearchTreeImp class

Syntax

```
T * Find( T * t ) const;
```

Description

Returns a pointer to the node containing *t.

TIBinarySearchTreeImp::Flush

TIBinarySearchTreeImp class

Syntax

```
void Flush(int del=0);
```

Description

Removes all items from the tree. They are deleted if del is 1. If del is 0 the items are not deleted.

TIBinarySearchTreeImp::ForEach

TIBinarySearchTreeImp class

Syntax

```
void ForEach( IterFunc iter, void * args, IteratorOrder order = InOrder )
```

Description

Creates an internal iterator that executes the given function iter for each item in the container. The args argument lets you pass arbitrary data to this function.

TIBinarySearchTreeImp::GetItemsInContainer

TIBinarySearchTreeImp class

Syntax

```
unsigned GetItemsInContainer();
```

Description

Returns the number of items in the tree.

TIBinarySearchTreeImp::Parent::IsEmpty

TIBinarySearchTreeImp class

Syntax

```
int IsEmpty();
```

Description

Returns 1 if the tree is empty; otherwise returns 0.

TIBinarySearchTreeImp::EqualTo

TIBinarySearchTreeImp class

Syntax

```
virtual int EqualTo( BinNode *n1, BinNode *n2 )
```

Description

Tests the equality between two nodes.

TIBinarySearchTreeImp::LessThan

TIBinarySearchTreeImp class

Syntax

```
virtual int LessThan( BinNode *n1, BinNode *n2 )
```

Description

Tests if node n1 is less than node n2.

TIBinarySearchTreeImp::DeleteNode

TIBinarySearchTreeImp class

Syntax

```
virtual void DeleteNode( BinNode *node, int del)
```

Description

Deletes node. The second parameter is ignored.

TIBinarySearchTreeIteratorImp template

Syntax

```
template <class T> class TIBinarySearchTreeIteratorImp;
```

Header File

binimp.h

Description

Implements an iterator that traverses TIBinarySearchTreeImp objects.

Public Constructors

TIBinarySearchTreeIteratorImp::TIBinarySearchTreeIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

TIBinarySearchTreeIteratorImp::TIBinarySearchTreeIteratorImp

TIBinarySearchTreeIteratorImp class

Syntax

```
TIBinarySearchTreeIteratorImp( TIBinarySearchTreeImp<T>& tree,
  TBinarySearchTreeBase::IteratorOrder order =
  TBinarySearchTreeBase::InOrder ) :
  TBinarySearchTreeIteratorImp<TVoidPointer>(tree,order)
```

Description

Constructs an iterator object that traverses a TIBinarySearchTreeImp container.

TIBinarySearchTreeIteratorImp::Current

TIBinarySearchTreeIteratorImp class

Syntax

```
T *Current() const;
```

Description

Returns a pointer to the current object.

TIBinarySearchTreeIteratorImp::Restart

TIBinarySearchTreeIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the tree.

TIBinarySearchTreeIteratorImp::operator int

TIBinarySearchTreeIteratorImp class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TIBinarySearchTreeIteratorImp::operator ++

TIBinarySearchTreeIteratorImp class

Form 1

```
T *operator ++ ( int i )
```

Form 2

```
T *operator ++ ()
```

Description

Form 1: Moves to the next object in the tree, and returns a pointer to the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns a pointer to the object that was current after the move (pre-increment).

TMDequeAsVector template

Syntax

```
template <class T, class Alloc> class TMDequeAsVector;
```

Header File

deques.h

Description

Implements a managed dequeue of T objects, using a vector as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Constructor

TMDequeAsVector::TMDequeAsVector

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

GetLeft

GetRight

IsEmpty

IsFull

LastThat

PeekLeft

PeekRight

PutLeft

PutRight

Protected Data Members

Data

Left

Right

Protected Member Functions

Next

Prev

TMDequeAsVector::CondFunc

TMDequeAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMDequeAsVector::IterFunc

TMDequeAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMDequeAsVector::TMDequeAsVector

TMDequeAsVector class

Syntax

```
TMDequeAsVector( unsigned max = DEFAULT_DEQUE_SIZE )
```

Description

Constructs a dequeue of max size.

TMDequeAsVector::FirstThat

See Also TMDequeAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the dequeue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TMDequeAsVector::Flush

See Also TMDequeAsVector class

Syntax

```
void Flush()
```

Description

Flushes the dequeue without destroying it.

See Also
TShouldDelete::ownsElements

TMDequeAsVector::ForEach

TMDequeAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args );
```

Description

Executes function iter for each dequeue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMDequeAsVector::GetItemsInContainer

TMDequeAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the dequeue.

TMDequeAsVector::GetLeft

See Also TMDequeAsVector class

Syntax

```
T GetLeft();
```

Description

Returns the object at the left end and removes it from the dequeue. The debuggable version throws an exception when the dequeue is empty.

See Also
PeekLeft

TMDequeAsVector::GetRight

See Also TMDequeAsVector class

Syntax

```
T GetRight();
```

Description

Same as GetLeft, except that the right end of the dequeue is returned.

See Also
PeekRight

TMDequeAsVector::IsEmpty

TMDequeAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the dequeue has no elements; otherwise returns 0.

TMDequeAsVector::IsFull

TMDequeAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the dequeue is full; otherwise returns 0.

TMDequeAsVector::LastThat

See Also TMDequeAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the dequeue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
FirstThat
ForEach

TMDequeAsVector::PeekLeft

See Also TMDequeAsVector class

Syntax

```
Const T& PeekLeft() const;
```

Description

Returns the object at the left end (head) of the dequeue. The object stays in the dequeue.

See Also
GetLeft

TMDequeAsVector::PeekRight

See Also TMDequeAsVector class

Syntax

```
Const T& PeekRight() const;
```

Description

Returns the object at the right end (tail) of the dequeue. The object stays in the dequeue.

See Also
GetRight

TMDequeAsVector::PutLeft

TMDequeAsVector class

Syntax

```
void PutLeft( const T& );
```

Description

Adds (pushes) the given object at the left end (head) of the dequeue.

TMDequeAsVector::PutRight

TMDequeAsVector class

Syntax

```
void PutRight( const T& );
```

Description

Adds (pushes) the given object at the right end (tail) of the dequeue.

TMDequeAsVector::Data

TMDequeAsVector class

Syntax

```
Vect Data;
```

Description

The vector containing the dequeue's data.

TMDequeAsVector::Left

TMDequeAsVector class

Syntax

```
unsigned Left;
```

Description

Index to the leftmost element of the dequeue.

TMDequeAsVector::Right

TMDequeAsVector class

Syntax

```
unsigned Right;
```

Description

Index to the rightmost element of the dequeue.

TMDequeAsVector::Next

See Also   TMDequeAsVector class

Syntax

```
unsigned Next( unsigned index ) const;
```

Description

Returns index + 1. Wraps around to the head of the dequeue.

See Also
Prev

TMDequeAsVector::Prev

TMDequeAsVector class

Syntax

```
unsigned Prev( unsigned index ) const;
```

Description

Returns index - 1. Wraps around to the tail of the dequeue.

TMDequeAsVectorIterator template

Syntax

```
template <class T, class Alloc> class TMDequeAsVectorIterator;
```

Header File

deques.h

Description

Implements an iterator object for a managed, vector-based dequeue.

Public Constructor

TMDequeAsVectorIterator::TMDequeAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TMDequeAsVectorIterator::TMDequeAsVectorIterator

TMDequeAsVectorIterator class

Syntax

```
TMDequeAsVectorIterator( const TMDequeAsVector<T,Alloc> &d )
```

Description

Constructs an object that iterates on TMDequeAsVector objects.

TMDequeAsVectorIterator::Current

TMDequeAsVectorIterator class

Syntax

```
Const T& Current();
```

Description

Returns the current object.

TMDequeAsVectorIterator::Restart

TMDequeAsVectorIterator class

Syntax

```
void Restart();
```

Description

Restarts iteration.

TMDequeAsVectorIterator::operator ++

TMDequeAsVectorIterator class

Form 1

```
Const T& operator ++ ( int );
```

Form 2

```
Const T& operator ++ ();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMDequeAsVectorIterator::operator int

TMDequeAsVectorIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. Iterator converts to 0 if nothing remains in the iterator.

TDequeAsVector template

Syntax

```
template <class T> class TDequeAsVector;
```

Header File

deques.h

Description

Implements a dequeue of T objects, using a vector as the underlying implementation. TStandardAllocator is used to manage memory.

Public Constructor

TDequeAsVector::TDequeAsVector

Type Definitions

CondFunc

IterFunc

Public Constructor

TDequeAsVector::TDequeAsVector

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

GetLeft

GetRight

IsEmpty

IsFull

LastThat

PeekLeft

PeekRight

PutLeft

PutRight

Protected Data Members

Data

Left

Right

Protected Member Functions

Next

Prev

TDequeAsVector::CondFunc

TDequeAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TDequeAsVector::IterFunc

TDequeAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TDequeAsVector::TDequeAsVector

TDequeAsVector class

Syntax

```
TDequeAsVector( unsigned max = DEFAULT_DEQUE_SIZE )
```

Description

Constructs a dequeue of max size.

TDequeAsVector::FirstThat

See Also TDequeAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the deque that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

LastThat

TDequeAsVector::Flush

See Also TDequeAsVector class

Syntax

```
void Flush()
```

Description

Flushes the dequeue without destroying it.

See Also
TShouldDelete::ownsElements

TDequeAsVector::ForEach

TDequeAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args );
```

Description

Executes function iter for each dequeue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TDequeAsVector::GetItemsInContainer

TDequeAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the dequeue.

TDequeAsVector::GetLeft

See Also TDequeAsVector class

Syntax

```
T GetLeft();
```

Description

Returns the object at the left end and removes it from the dequeue. The debuggable version throws an exception when the dequeue is empty.

See Also
PeekLeft

TDequeAsVector::GetRight

See Also TDequeAsVector class

Syntax

```
T GetRight();
```

Description

Same as GetLeft, except that the right end of the dequeue is returned.

See Also
PeekRight

TDequeAsVector::IsEmpty

TDequeAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the dequeue has no elements; otherwise returns 0.

TDequeAsVector::IsFull

TDequeAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the dequeue is full; otherwise returns 0.

TDequeAsVector::LastThat

See Also   TDequeAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the deque that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

FirstThat

ForEach

TDequeAsVector::PeekLeft

See Also   TDequeAsVector class

Syntax

```
Const T& PeekLeft() const;
```

Description

Returns the object at the left end (head) of the dequeue. The object stays in the dequeue.

See Also
GetLeft

TDequeAsVector::PeekRight

See Also   TDequeAsVector class

Syntax

```
Const T& PeekRight() const;
```

Description

Returns the object at the right end (tail) of the dequeue. The object stays in the dequeue.

See Also
GetRight

TDequeAsVector::PutLeft

TDequeAsVector class

Syntax

```
void PutLeft( const T& );
```

Description

Adds (pushes) the given object at the left end (head) of the dequeue.

TDequeAsVector::PutRight

TDequeAsVector class

Syntax

```
void PutRight( const T& );
```

Description

Adds (pushes) the given object at the right end (tail) of the dequeue.

TDequeAsVector::Data

TDequeAsVector class

Syntax

```
Vect Data;
```

Description

The vector containing the dequeue's data.

TDequeAsVector::Left

TDequeAsVector class

Syntax

```
unsigned Left;
```

Description

Index to the leftmost element of the dequeue.

TDequeAsVector::Right

TDequeAsVector class

Syntax

```
unsigned Right;
```

Description

Index to the rightmost element of the dequeue.

TDequeAsVector::Next

See Also   TDequeAsVector class

Syntax

```
unsigned Next( unsigned index ) const;
```

Description

Returns index + 1. Wraps around to the head of the dequeue.

See Also
Prev

TDequeAsVector::Prev

TDequeAsVector class

Syntax

```
unsigned Prev( unsigned index ) const;
```

Description

Returns index - 1. Wraps around to the tail of the dequeue.

TDequeAsVectorIterator template

Syntax

```
template <class T> class TDequeAsVectorIterator;
```

Header File

deques.h

Description

Implements an iterator object for a vector-based dequeue.

Public Constructor

TDequeAsVectorIterator::TDequeAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TDequeAsVectorIterator::TDequeAsVectorIterator

TDequeAsVectorIterator class

Syntax

```
TDequeAsVectorIterator( const TDequeAsVector<T> &d )
```

Description

Constructs an object that iterates on TMDequeAsVector objects.

TDequeAsVectorIterator::Current

TDequeAsVectorIterator class

Syntax

```
Const T& Current();
```

Description

Returns the current object.

TDequeAsVectorIterator::Restart

TDequeAsVectorIterator class

Syntax

```
void Restart();
```

Description

Restarts iteration.

TDequeAsVectorIterator::operator ++

TDequeAsVectorIterator class

Form 1

```
Const T& operator ++ ( int );
```

Form 2

```
Const T& operator ++ ();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TDequeAsVectorIterator::operator int

TDequeAsVectorIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. Iterator converts to 0 if nothing remains in the iterator.

TMIDequeAsVector template

Syntax

```
template <class T, class Alloc> class TMIDequeAsVector;
```

Header File

deques.h

Description

Implements a managed, indirect dequeue of pointers to objects of type T, using a vector as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Constructor

TMIDequeAsVector::TMIDequeAsVector

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

GetLeft

GetRight

IsEmpty

isFull

LastThat

PeekLeft

PeekRight

PutLeft

PutRight

TMIDequeAsVector::CondFunc

TMIDequeAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMIDequeAsVector::IterFunc

TMIDequeAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMIDequeAsVector::TMIDequeAsVector

TMIDequeAsVector class

Syntax

```
TMIDequeAsVector( unsigned sz = DEFAULT_DEQUE_SIZE )
```

Description

Constructs an indirect dequeue of max size.

TMIDequeAsVector::FirstThat

See Also   TMIDequeAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the dequeue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TMIDequeAsVector::Flush

TMIDequeAsVector class

Syntax

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete );
```

Description

Flushes the dequeue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the dt argument.

TMIDequeAsVector::ForEach

TMIDequeAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args );
```

Description

Executes function iter for each dequeue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMIDequeAsVector::GetItemsInContainer

TMIDequeAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the dequeue.

TMIDequeAsVector::GetLeft

See Also TMIDequeAsVector class

Syntax

```
T *GetLeft()
```

Description

Returns a pointer to the object at the left end and removes it from the dequeue. Returns 0 if the dequeue is empty.

See Also
PeekLeft

TMIDequeAsVector::GetRight

See Also   TMIDequeAsVector class

Syntax

```
T *GetRight()
```

Description

Same as GetLeft, except that the right end of the dequeue is returned.

See Also
PeekRight

TMIDequeAsVector::IsEmpty

TMIDequeAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if a dequeue has no elements; otherwise returns 0.

TMIDequeAsVector::IsFull

TMIDequeAsVector class

Syntax

```
int isFull() const;
```

Description

Returns 1 if a dequeue is full; otherwise returns 0.

TMIDequeAsVector::LastThat

See Also   TMIDequeAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the dequeue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

FirstThat

ForEach

TMIDequeAsVector::PeekLeft

See Also   TMIDequeAsVector class

Syntax

```
T *PeekLeft() const;
```

Description

Returns a pointer to the object at the left end (head) of the dequeue. The object stays in the dequeue.

See Also
GetLeft

TMIDequeAsVector::PeekRight

See Also TMIDequeAsVector class

Syntax

```
T *PeekRight() const;
```

Description

Returns the object at the right end (tail) of the dequeue. The object stays in the dequeue.

See Also
GetRight

TMIDequeAsVector::PutLeft

TMIDequeAsVector class

Syntax

```
void PutLeft( T *t )
```

Description

Adds (pushes) the given object pointer at the left end (head) of the dequeue.

TMIDequeAsVector::PutRight

TMIDequeAsVector class

Syntax

```
void PutRight( T *t )
```

Description

Adds (pushes) the given object pointer at the right end (tail) of the dequeue.

TMIDequeAsVectorIterator template

Syntax

```
template <class T, class Alloc> class TMIDequeAsVectorIterator;
```

Header File

deques.h

Description

Implements an iterator for the family of managed, indirect dequeues implemented as vectors.

Public Constructor

TMIDequeAsVectorIterator::TMIDequeAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TMIDequeAsVectorIterator::TMIDequeAsVectorIterator

TMIDequeAsVectorIterator class

Syntax

```
TMIDequeAsVectorIterator( const TMIDequeAsVector<T,Alloc> &d )
```

Description

Creates an object that iterates on TMIDequeAsVector objects.

TMIDequeAsVectorIterator::Current

TMIDequeAsVectorIterator class

Syntax

```
Const T& Current();
```

Description

Returns the current object.

TMIDequeAsVectorIterator::Restart

TMIDequeAsVectorIterator class

Syntax

```
void Restart();
```

Description

Restarts iteration.

TMIDequeAsVectorIterator::operator ++

TMIDequeAsVectorIterator class

Form 1

```
Const T& operator ++ ( int );
```

Form 2

```
Const T& operator ++ ();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMIDequeAsVectorIterator::operator int

TMIDequeAsVectorIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. Iterator converts to 0 if nothing remains in the iterator.

TIDequeAsVector template

Syntax

```
template <class T> class TIDequeAsVector;
```

Header File

deques.h

Description

Implements an indirect dequeue of pointers to objects of type T, using a vector as the underlying implementation.

Public Constructor

TIDequeAsVector::TIDequeAsVector

Type Definitions

CondFunc

IterFunc

Public Constructor

TIDequeAsVector::TIDequeAsVector

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

GetLeft

GetRight

IsEmpty

isFull

LastThat

PeekLeft

PeekRight

PutLeft

PutRight

TIDequeAsVector::CondFunc

TIDequeAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TIDequeAsVector::IterFunc

TIDequeAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TIDequeAsVector::TIDequeAsVector

TIDequeAsVector class

Syntax

```
TIDequeAsVector( unsigned sz = DEFAULT_DEQUE_SIZE ) :
  TMIDequeAsVector<T,TStandardAllocator>(sz)
```

Description

Constructs an indirect dequeue of max size.

TIDequeAsVector::FirstThat

See Also TIDequeAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the dequeue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TIDequeAsVector::Flush

TIDequeAsVector class

Syntax

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete );
```

Description

Flushes the dequeue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the dt argument.

TIDequeAsVector::ForEach

TIDequeAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args );
```

Description

Executes function iter for each dequeue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TIDequeAsVector::GetItemsInContainer

TIDequeAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the dequeue.

TIDequeAsVector::GetLeft

See Also   TIDequeAsVector class

Syntax

```
T *GetLeft()
```

Description

Returns a pointer to the object at the left end and removes it from the dequeue. Returns 0 if the dequeue is empty.

See Also
PeekLeft

TIDequeAsVector::GetRight

See Also TIDequeAsVector class

Syntax

```
T *GetRight()
```

Description

Same as GetLeft, except that the right end of the dequeue is returned.

See Also
PeekRight

TIDequeAsVector::IsEmpty

TIDequeAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if a dequeue has no elements; otherwise returns 0.

TIDequeAsVector::IsFull

TIDequeAsVector class

Syntax

```
int isFull() const;
```

Description

Returns 1 if a dequeue is full; otherwise returns 0.

TIDequeAsVector::LastThat

See Also TIDequeAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the dequeue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

FirstThat

ForEach

TIDequeAsVector::PeekLeft

See Also TIDequeAsVector class

Syntax

```
T *PeekLeft() const;
```

Description

Returns a pointer to the object at the left end (head) of the dequeue. The object stays in the dequeue.

See Also
GetLeft

TIDequeAsVector::PeekRight

See Also TIDequeAsVector class

Syntax

```
T *PeekRight() const;
```

Description

Returns the object at the right end (tail) of the dequeue. The object stays in the dequeue.

See Also
GetRight

TIDequeAsVector::PutLeft

TIDequeAsVector class

Syntax

```
void PutLeft( T *t )
```

Description

Adds (pushes) the given object pointer at the left end (head) of the dequeue.

TIDequeAsVector::PutRight

TIDequeAsVector class

Syntax

```
void PutRight( T *t )
```

Description

Adds (pushes) the given object pointer at the right end (tail) of the dequeue.

TIDequeAsVectorIterator template

Syntax

```
template <class T> class TIDequeAsVectorIterator;
```

Header File

deques.h

Description

Implements an iterator for the family of indirect dequeues implemented as vectors.

Public Constructor

TIDequeAsVectorIterator::TIDequeAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TIDequeAsVectorIterator::TIDequeAsVectorIterator

TIDequeAsVectorIterator class

Syntax

```
TIDequeAsVectorIterator( const TIDequeAsVector<T> &d )
```

Description

Constructs an object that iterates on TIDequeAsVector objects.

TIDequeAsVectorIterator::Current

TIDequeAsVectorIterator class

Syntax

```
Const T& Current();
```

Description

Returns the current object.

TIDequeAsVectorIterator::Restart

TIDequeAsVectorIterator class

Syntax

```
void Restart();
```

Description

Restarts iteration.

TIDequeAsVectorIterator::operator ++

TIDequeAsVectorIterator class

Form 1

```
Const T& operator ++ ( int );
```

Form 2

```
Const T& operator ++ ();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TIDequeAsVectorIterator::operator int

TIDequeAsVectorIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. Iterator converts to 0 if nothing remains in the iterator.

TMDequeAsDoubleList template

Syntax

```
template <class T, class Alloc> class TMDequeDoubleList;
```

Header File

deques.h

Description

Implements a managed dequeue of objects of type T, using a double-linked list as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

GetLeft

GetRight

IsEmpty

IsFull

LastThat

PeekLeft

PeekRight

PutLeft

PutRight

TMDequeAsDoubleList::CondFunc

TMDequeAsDoubleList class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMDequeAsDoubleList::IterFunc

TMDequeAsDoubleList class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMDequeAsDoubleList::FirstThat

See Also TMDequeAsDoubleList class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the dequeue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TMDequeAsDoubleList::Flush

TMDequeAsDoubleList class

Syntax

```
void Flush( int del )
```

Description

Flushes the dequeue without destroying it.

TMDequeAsDoubleList::ForEach

TMDequeAsDoubleList class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each dequeue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMDequeAsDoubleList::GetItemsInContainer

TMDequeAsDoubleList class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the dequeue.

TMDequeAsDoubleList::GetLeft

TMDequeAsDoubleList class

Syntax

```
T GetLeft()
```

Description

Returns the object at the left end and removes it from the dequeue.

TMDequeAsDoubleList::GetRight

See Also TMDequeAsDoubleList class

Syntax

```
T GetRight()
```

Description

Same as GetLeft, except that the right end of the dequeue is returned.

See Also
PeekRight

TMDequeAsDoubleList::IsEmpty

TMDequeAsDoubleList class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if a dequeue has no elements; otherwise returns 0.

TMDequeAsDoubleList::IsFull

TMDequeAsDoubleList class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if a dequeue is full; otherwise returns 0.

TMDequeAsDoubleList::LastThat

See Also   TMDequeAsDoubleList class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the dequeue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

FirstThat

ForEach

TMDequeAsDoubleList::PeekLeft

See Also TMDequeAsDoubleList class

Syntax

```
Const T& PeekLeft() const;
```

Description

Returns a reference to the object at the left end (head) of the dequeue. The object stays in the dequeue.

See Also
GetLeft

TMDequeAsDoubleList::PeekRight

See Also TMDequeAsDoubleList class

Syntax

```
Const T& PeekRight() const;
```

Description

Returns a reference to the object at the right end (tail) of the dequeue. The object stays in the dequeue.

See Also
GetRight

TMDequeAsDoubleList::PutLeft

TMDequeAsDoubleList class

Syntax

```
void PutLeft( const T& t )
```

Description

Adds (pushes) the given object at the left end (head) of the dequeue.

TMDequeAsDoubleList::PutRight

TMDequeAsDoubleList class

Syntax

```
void PutRight( const T& t )
```

Description

Adds (pushes) the given object at the right end (tail) of the dequeue.

TMDequeAsDoubleListIterator template

Syntax

```
template <class T, class Alloc> class TMDequeAsDoubleListIterator;
```

Header File

deques.h

Description

Implements an iterator object for a double-list based deques.

Public Constructor

TMDequeAsDoubleListIterator::TMDequeAsDoubleListIterator

Public Member Functions

Current

Restart

Operators

int

++

--

TMDequeAsDoubleListIterator::TMDequeAsDoubleListIterator

TMDequeAsDoubleListIterator class

Syntax

```
TMDequeAsDoubleListIterator( const TMDequeAsDoubleList<T, Alloc> & s )
```

Description

Constructs an object that iterates on TMDequeAsDoubleList objects.

TMDequeAsDoubleListIterator::Current

TMDequeAsDoubleListIterator class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TMDequeAsDoubleListIterator::Restart

TMDequeAsDoubleListIterator class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TMDequeAsDoubleListIterator::operator int

TMDequeAsDoubleListIterator class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMDequeAsDoubleListIterator::operator ++

TMDequeAsDoubleListIterator class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMDequeAsDoubleListIterator::operator - -

TMDequeAsDoubleListIterator class

Form 1

```
const T& operator -- ( int )
```

Form 2

```
const T& operator -- ()
```

Description

Form 1: Moves to the previous object, and returns the object that was current before the move (post-decrement).

Form 2: Moves to the previous object, and returns the object that was current after the move (pre-decrement).

TDequeAsDoubleList template

Syntax

```
template <class T> class TMDequeAsDoubleList;
```

Header File

deques.h

Description

Implements a dequeue of objects of type T, using a double-linked list as the underlying implementation, and TStandardAllocator as its memory manager.

Type Definitions

CondFunc

IterFunc

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

GetLeft

GetRight

IsEmpty

IsFull

LastThat

PeekLeft

PeekRight

PutLeft

PutRight

TDequeAsDoubleList::CondFunc

TDequeAsDoubleList class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TDequeAsDoubleList::IterFunc

TDequeAsDoubleList class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TDequeAsDoubleList::FirstThat

See Also TDequeAsDoubleList class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the dequeue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TDequeAsDoubleList::Flush

TDequeAsDoubleList class

Syntax

```
void Flush()
```

Description

Flushes the dequeue without destroying it.

TDequeAsDoubleList::ForEach

TDequeAsDoubleList class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each dequeue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TDequeAsDoubleList::GetItemsInContainer

TDequeAsDoubleList class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the dequeue.

TDequeAsDoubleList::GetLeft

TDequeAsDoubleList class

Syntax

```
T GetLeft()
```

Description

Returns the object at the left end and removes it from the dequeue.

TDequeAsDoubleList::GetRight

See Also TDequeAsDoubleList class

Syntax

```
T GetRight()
```

Description

Same as GetLeft, except that the right end of the dequeue is returned.

See Also
PeekRight

TDequeAsDoubleList::IsEmpty

TDequeAsDoubleList class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if a dequeue has no elements; otherwise returns 0.

TDequeAsDoubleList::IsFull

TDequeAsDoubleList class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if a dequeue is full; otherwise returns 0.

TDequeAsDoubleList::LastThat

See Also TDequeAsDoubleList class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the dequeue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

FirstThat

ForEach

TDequeAsDoubleList::PeekLeft

See Also TDequeAsDoubleList class

Syntax

```
Const T& PeekLeft() const;
```

Description

Returns a reference to the object at the left end (head) of the dequeue. The object stays in the dequeue.

See Also
GetLeft

TDequeAsDoubleList::PeekRight

See Also TDequeAsDoubleList class

Syntax

```
Const T& PeekRight() const;
```

Description

Returns a reference to the object at the right end (tail) of the dequeue. The object stays in the dequeue.

See Also
GetRight

TDequeAsDoubleList::PutLeft

TDequeAsDoubleList class

Syntax

```
void PutLeft( const T& t )
```

Description

Adds (pushes) the given object at the left end (head) of the dequeue.

TDequeAsDoubleList::PutRight

TDequeAsDoubleList class

Syntax

```
void PutRight( const T& t )
```

Description

Adds (pushes) the given object at the right end (tail) of the dequeue.

TDequeAsDoubleListIterator template

Syntax

```
template <class T> class TMDequeAsDoubleListIteratir;
```

Header File

deques.h

Description

Implements an iterator object for a double-list based dequeue.

Public Constructor

TMDequeAsDoubleListIterator::TMDequeAsDoubleListIterator

TDequeAsDoubleListIterator::TDequeAsDoubleListIterator

TDequeAsDoubleListIterator class

Syntax

```
TMDequeAsDoubleListIterator( const TMDequeAsDoubleList<T, Alloc> & s )
```

Description

Constructs an object that iterates on TDequeAsDoubleList objects.

TMIDequeAsDoubleList template

Syntax

```
template <class T, class Alloc> class TMIDequeAsDoubleList;
```

Header File

deques.h

Description

Implements a managed dequeue of pointers to objects of type T, using a double-linked list as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

GetLeft

GetRight

IsEmpty

IsFull

LastThat

PeekLeft

PeekRight

PutLeft

PutRight

TMIDequeAsDoubleList::CondFunc

TMIDequeAsDoubleList class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMIDequeAsDoubleList::IterFunc

TMIDequeAsDoubleList class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMIDequeAsDoubleList::FirstThat

See Also TMIDequeAsDoubleList class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the dequeue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TMIDequeAsDoubleList::Flush

TMIDequeAsDoubleList class

Syntax

```
void Flush( TShouldDelete::DeleteType dt = TShouldDelete::DefDelete )
```

Description

Flushes the dequeue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the dt argument.

TMIDequeAsDoubleList::ForEach

TMIDequeAsDoubleList class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each dequeue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMIDequeAsDoubleList::GetItemsInContainer

TMIDequeAsDoubleList class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the dequeue.

TMIDequeAsDoubleList::GetLeft

See Also TMIDequeAsDoubleList class

Syntax

```
T *GetLeft()
```

Description

Returns a pointer to the object at the left end and removes it from the dequeue. Returns 0 if the dequeue is empty.

See Also
PeekLeft

TMIDequeAsDoubleList::GetRight

See Also TMIDequeAsDoubleList class

Syntax

```
T *GetRight()
```

Description

Same as GetLeft, except that a pointer to the object at the right end of the dequeue is returned.

See Also
PeekRight

TMIDequeAsDoubleList::IsEmpty

TMIDequeAsDoubleList class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the dequeue has no elements; otherwise returns 0.

TMIDequeAsDoubleList::IsFull

TMIDequeAsDoubleList class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the dequeue is full; otherwise returns 0.

TMIDequeAsDoubleList::LastThat

See Also   TMIDequeAsDoubleList class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the dequeue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

FirstThat

ForEach

TMIDequeAsDoubleList::PeekLeft

TMIDequeAsDoubleList class

Syntax

```
T *PeekLeft() const;
```

Description

Returns a pointer to the object at the left end (head) of the dequeue. The object stays in the dequeue.

TMIDequeAsDoubleList::PeekRight

TMIDequeAsDoubleList class

Syntax

```
T *PeekRight() const;
```

Description

Returns the object at the right end (tail) of the dequeue. The object stays in the dequeue.

TMIDequeAsDoubleList::PutLeft

TMIDequeAsDoubleList class

Syntax

```
void PutLeft( T *t )
```

Description

Adds (pushes) the given object pointer at the left end (head) of the dequeue.

TMIDequeAsDoubleList::PutRight

TMIDequeAsDoubleList class

Syntax

```
void PutRight( T *t )
```

Description

Adds (pushes) the given object pointer at the right end (tail) of the dequeue.

TMIDequeAsDoubleListIterator template

Syntax

```
template <class T, class Alloc> class TMDequeAsDoubleListIterator;
```

Header File

deques.h

Description

Implements an iterator for the family of managed, indirect dequeues implemented as double lists.

Public Constructor

TMIDequeAsDoubleListIterator::TMIDequeAsDoubleListIterator

Public Member Functions

Current

Restart

Operators

int

++

--

TMIDequeAsDoubleListIterator::TMIDequeAsDoubleListIterator

TMIDequeAsDoubleListIterator class

Syntax

```
TMIDequeAsDoubleListIterator( const TMIDequeAsDoubleList<T,Alloc> s )
```

Description

Constructs an object that iterates on TMIDequeAsDoubleList objects.

TMIDequeAsDoubleListIterator::Current

TMIDequeAsDoubleListIterator class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TMIDequeAsDoubleListIterator::Restart

TMIDequeAsDoubleListIterator class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TMIDequeAsDoubleListIterator::operator int

TMIDequeAsDoubleListIterator class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMIDequeAsDoubleListIterator::operator ++

TMIDequeAsDoubleListIterator class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMIDequeAsDoubleListIterator::operator - -

TMIDequeAsDoubleListIterator class

Form 1

```
const T& operator -- ( int )
```

Form 2

```
const T& operator -- ()
```

Description

Form 1: Moves to the previous object, and returns the object that was current before the move (post-decrement).

Form 2: Moves to the previous object, and returns the object that was current after the move (pre-decrement).

TIDequeAsDoubleList template

Syntax

```
template <class T> class TIDequeAsDoubleList;
```

Header File

deques.h

Description

Implements a dequeue of pointers to objects of type T, using a double-linked list as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

GetLeft

GetRight

IsEmpty

IsFull

LastThat

PeekLeft

PeekRight

PutLeft

PutRight

TIDequeAsDoubleList::CondFunc

TIDequeAsDoubleList class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TIDequeAsDoubleList::IterFunc

TIDequeAsDoubleList class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TIDequeAsDoubleList::FirstThat

See Also TIDequeAsDoubleList class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the dequeue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TIDequeAsDoubleList::Flush

TIDequeAsDoubleList class

Syntax

```
void Flush( TShouldDelete::DeleteType dt = TShouldDelete::DefDelete )
```

Description

Flushes the dequeue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the dt argument.

TIDequeAsDoubleList::ForEach

TIDequeAsDoubleList class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each dequeue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TIDequeAsDoubleList::GetItemsInContainer

TIDequeAsDoubleList class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the dequeue.

TIDequeAsDoubleList::GetLeft

See Also TIDequeAsDoubleList class

Syntax

```
T *GetLeft()
```

Description

Returns a pointer to the object at the left end and removes it from the dequeue. Returns 0 if the dequeue is empty.

See Also
PeekLeft

TIDequeAsDoubleList::GetRight

See Also TIDequeAsDoubleList class

Syntax

```
T *GetRight()
```

Description

Same as GetLeft, except that a pointer to the object at the right end of the dequeue is returned.

See Also
PeekRight

TIDequeAsDoubleList::IsEmpty

TIDequeAsDoubleList class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the dequeue has no elements; otherwise returns 0.

TIDequeAsDoubleList::IsFull

TIDequeAsDoubleList class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the dequeue is full; otherwise returns 0.

TIDequeAsDoubleList::LastThat

See Also TIDequeAsDoubleList class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the dequeue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition. Note that LastThat creates its own internal iterator, so you can treat it as a "search" function.

See Also

FirstThat

ForEach

TIDequeAsDoubleList::PeekLeft

TIDequeAsDoubleList class

Syntax

```
T *PeekLeft() const;
```

Description

Returns a pointer to the object at the left end (head) of the dequeue. The object stays in the dequeue.

TIDequeAsDoubleList::PeekRight

TIDequeAsDoubleList class

Syntax

```
T *PeekRight() const;
```

Description

Returns the object at the right end (tail) of the dequeue. The object stays in the dequeue.

TIDequeAsDoubleList::PutLeft

TIDequeAsDoubleList class

Syntax

```
void PutLeft( T *t )
```

Description

Adds (pushes) the given object pointer at the left end (head) of the dequeue.

TIDequeAsDoubleList::PutRight

TIDequeAsDoubleList class

Syntax

```
void PutRight( T *t )
```

Description

Adds (pushes) the given object pointer at the right end (tail) of the dequeue.

TIDequeAsDoubleListIterator template

Syntax

```
template <class T, class Alloc> class TIDequeAsDoubleListIterator;
```

Header File

deques.h

Description

Implements an iterator for the family of indirect dequeues implemented as double lists.

Public Constructor

TIDequeAsDoubleListIterator::TIDequeAsDoubleListIterator

Public Member Functions

Current

Restart

Operators

int

++

--

TIDequeAsDoubleListIterator::TIDequeAsDoubleListIterator

TIDequeAsDoubleListIterator class

Syntax

```
TIDequeAsDoubleListIterator( const TIDequeAsDoubleList<T> & s )
```

Description

Constructs an object that iterates on TIDequeAsDoubleList objects.

TIDequeAsDoubleListIterator::Current

TIDequeAsDoubleListIterator class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TIDequeAsDoubleListIterator::Restart

TIDequeAsDoubleListIterator class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TIDequeAsDoubleListIterator::operator int

TIDequeAsDoubleListIterator class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TIDequeAsDoubleListIterator::operator ++

TIDequeAsDoubleListIterator class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TIDequeAsDoubleListIterator::operator - -

TIDequeAsDoubleListIterator class

Form 1

```
const T& operator -- ( int )
```

Form 2

```
const T& operator -- ()
```

Description

Form 1: Moves to the previous object, and returns the object that was current before the move (post-decrement).

Form 2: Moves to the previous object, and returns the object that was current after the move (pre-decrement).

TMDictionaryAsHashTable template

Syntax

```
template <class T, class A> class TMDictionaryAsHashTable;
```

Header File

dict.h

Description

Implements a managed dictionary using a hash table as the underlying FDS, and using the user-supplied storage allocator A. It assumes that T is one of the four types of associations, and that T has meaningful copy and == semantics as well as a default constructor.

Protected Data Member

HashTable

Public Constructor

TMDictionaryAsHashTable::TMDictionaryAsHashTable

Public Member Functions

Add
Detach
Find
Flush
ForEach
GetItemsInContainer
IsEmpty

TMDictionaryAsHashTable::HashTable

TMDictionaryAsHashTable class

Syntax

```
TMHashTableImp<T,A> HashTable;
```

Description

Implements the underlying hash table.

TMDictionaryAsHashTable::TMDictionaryAsHashTable

TMDictionaryAsHashTable class

Syntax

```
TMDictionaryAsHashTable( unsigned size = DEFAULT_HASH_TABLE_SIZE )
```

Description

Constructs a dictionary with the specified size.

TMDictionaryAsHashTable::Add

TMDictionaryAsHashTable class

Syntax

```
int Add( const T& t )
```

Description

Adds item t if not already in the dictionary.

TMDictionaryAsHashTable::Detach

TMDictionaryAsHashTable class

Syntax

```
int Detach( const T& t, DeleteType dt = DefDelete )
```

Description

Removes item t from the dictionary. Calls DeleteElements on the association.

TMDictionaryAsHashTable::Find

TMDictionaryAsHashTable class

Syntax

```
T * Find( constT& t )
```

Description

Returns a pointer to item t.

TMDictionaryAsHashTable::Flush

TMDictionaryAsHashTable class

Syntax

```
void Flush( DeleteType dt = DefDelete )
```

Description

Removes all items from the dictionary. Calls DeleteElements on the association.

TMDictionaryAsHashTable::ForEach

TMDictionaryAsHashTable class

Syntax

```
void ForEach( void ( IterFunc iter, void * args )
```

Description

Creates an internal iterator that executes the given function iter for each item in the container. The args argument lets you pass arbitrary data to this function.

TMDictionaryAsHashTable::GetItemsInContainer

TMDictionaryAsHashTable class

Syntax

```
inline unsigned GetItemsInContainer()
```

Description

Returns the number of items in the dictionary.

TMDictionaryAsHashTable::IsEmpty

TMDictionaryAsHashTable class

Syntax

```
inline int IsEmpty()
```

Description

Returns 1 if the dictionary is empty; otherwise returns 0.

TMDictionaryAsHashTableIterator template

Syntax

```
template <class T, class A> class TMDictionaryAsHashTableIterator;
```

Header File

dict.h

Description

Implements an iterator that traverses TMDictionaryAsHashTable objects, using the user-supplied storage allocator A.

Public Constructor

TMDictionaryAsHashTableIterator::TMDictionaryAsHashTableIterator

Public Member Functions

Current

Restart

Operators

int

++

TMDictionaryAsHashTableIterator::TMDictionaryAsHashTableIterator

TMDictionaryAsHashTableIterator class

Syntax

```
TMDictionaryAsHashTableIterator(TMDictionaryAsHashTable<T,A> & t )
```

Description

Constructs an iterator object that traverses a TMDictionaryAsHashTable container.

TMDictionaryAsHashTableIterator::Current

TMDictionaryAsHashTableIterator class

Syntax

```
Const T& Current()
```

Description

Returns the current object.

TMDictionaryAsHashTableIterator::Restart

TMDictionaryAsHashTableIterator class

Syntax

```
void Restart();
```

Description

Restarts iteration from the beginning of the dictionary.

TMDictionaryAsHashTableIterator::operator int

TMDictionaryAsHashTableIterator class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMDictionaryAsHashTableIterator::operator ++

TMDictionaryAsHashTableIterator class

Form 1

```
Const T& operator ++ (int)
```

Form 2

```
Const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TDictionaryAsHashTable template

Syntax

```
template <class T> class TDictionaryAsHashTable;
```

Header File

dict.h

Description

Implements a dictionary objects of type T, using the system storage allocator TStandardAllocator. It assumes that T is one of the four types of associations, and that T has meaningful copy and == semantics as well as a default constructor.

Protected Data Member

HashTable

Public Constructor

TDictionaryAsHashTable::TDictionaryAsHashTable

Public Member Functions

Add

Detach

Find

Flush

ForEach

GetItemsInContainer

IsEmpty

TDictionaryAsHashTable::HashTable

TDictionaryAsHashTable class

Syntax

```
TMHashTableImp<T,A> HashTable;
```

Description

Implements the underlying hash table.

TDictionaryAsHashTable::TDictionaryAsHashTable

TDictionaryAsHashTable class

Syntax

```
TDictionaryAsHashTable( unsigned size = DEFAULT_HASH_TABLE_SIZE )
```

Description

Constructs a dictionary with the specified size.

TDictionaryAsHashTable::Add

TDictionaryAsHashTable class

Syntax

```
int Add( const T& t )
```

Description

Adds item t if not already in the dictionary.

TDictionaryAsHashTable::Detach

TDictionaryAsHashTable class

Syntax

```
int Detach( const T& t, DeleteType dt = DefDelete )
```

Description

Removes item t from the dictionary. Calls DeleteElements on the association.

TDictionaryAsHashTable::Find

TDictionaryAsHashTable class

Syntax

```
T * Find( constT& t )
```

Description

Returns a pointer to item t.

TDictionaryAsHashTable::Flush

TDictionaryAsHashTable class

Syntax

```
void Flush( DeleteType dt = DefDelete )
```

Description

Removes all items from the dictionary. Calls DeleteElements on the association.

TDictionaryAsHashTable::ForEach

TDictionaryAsHashTable class

Syntax

```
void ForEach( IterFunc iter, void * args )
```

Description

Creates an internal iterator that executes the given function iter for each item in the container. The args argument lets you pass arbitrary data to this function.

TDictionaryAsHashTable::GetItemsInContainer

TDictionaryAsHashTable class

Syntax

```
inline unsigned GetItemsInContainer()
```

Description

Returns the number of items in the dictionary.

TDictionaryAsHashTable::IsEmpty

TDictionaryAsHashTable class

Syntax

```
inline int IsEmpty()
```

Description

Returns 1 if the dictionary is empty; otherwise returns 0.

TDictionaryAsHashTableIterator template

Syntax

```
template <class T> class TDictionaryAsHashTableIterator
```

Header File

dict.h

Description

Implements an iterator that traverses TDictionaryAsHashTable objects, using the system storage allocator TStandardAllocator.

Public Constructor

TDictionaryAsHashTableIterator::TDictionaryAsHashTableIterator

TDictionaryAsHashTableIterator::TDictionaryAsHashTableIterator

TDictionaryAsHashTableIterator class

Syntax

```
TDictionaryAsHashTableIterator( TDictionaryAsHashTable<T> & t )
```

Description

Constructs an iterator object that traverses a TDictionaryAsHashTable container.

TMIDictionaryAsHashTable template

Syntax

```
template <class T, class A> class TMIDictionaryAsHashTable;
```

Header File

dict.h

Description

Implements a managed indirect dictionary using a hash table as the underlying FDS, and using the user-supplied storage allocator A. It assumes that T is an association class.

Public Constructor

TMIDictionaryAsHashTable::TMIDictionaryAsHashTable

Public Member Functions

Add

Detach

Find

Flush

ForEach

GetItemsInContainer

IsEmpty

TMIDictionaryAsHashTable::TMIDictionaryAsHashTable

TMIDictionaryAsHashTable class

Syntax

```
TMIDictionaryAsHashTable( unsigned size = DEFAULT_HASH_TABLE_SIZE )
```

Description

Constructs an indirect dictionary with the specified size.

TMIDictionaryAsHashTable::Add

TMIDictionaryAsHashTable class

Syntax

```
int Add( T * t )
```

Description

Adds a pointer to item t if not already in the dictionary.

TMIDictionaryAsHashTable::Detach

TMIDictionaryAsHashTable class

Syntax

```
int Detach( T * t, int del = 0 )
```

Description

Removes the pointer to item t from the dictionary, and deletes if del is 1. If del is 0 the item is not deleted.

TMIDictionaryAsHashTable::Find

TMIDictionaryAsHashTable class

Syntax

```
T * Find( T * t )
```

Description

Returns a pointer to item t.

TMIDictionaryAsHashTable::Flush

TMIDictionaryAsHashTable class

Syntax

```
void Flush( int del = 0 )
```

Description

Removes all items from the dictionary. The item is deleted if del is 1. If del is 0 the item is not deleted.

TMIDictionaryAsHashTable::ForEach

TMIDictionaryAsHashTable class

Syntax

```
void ForEach( IterFunc iter, void * args );
```

Description

Creates an internal iterator that executes the given function iter for each item in the container. The args argument lets you pass arbitrary data to this function.

TMIDictionaryAsHashTable::GetItemsInContainer

TMIDictionaryAsHashTable class

Syntax

```
inline unsigned GetItemsInContainer()
```

Description

Returns the number of items in the dictionary.

TMIDictionaryAsHashTable::IsEmpty

TMIDictionaryAsHashTable class

Syntax

```
inline int IsEmpty()
```

Description

Returns 1 if the dictionary is empty; otherwise returns 0.

TMIDictionaryAsHashTableIterator template

Syntax

```
template <class T, class A> class TMDictionaryAsHashTableIterator;
```

Header File

dict.h

Description

Implements an iterator that traverses TMIDictionaryAsHashTable objects, using the user-supplied storage allocator A.

Public Constructor

TMIDictionaryAsHashTableIterator::TMIDictionaryAsHashTableIterator

Public Member Functions

Current

Restart

Operators

int

++

TMIDictionaryAsHashTableIterator::TMIDictionaryAsHashTableIterator

TMIDictionaryAsHashTableIterator class

Syntax

```
TMIDictionaryAsHashTableIterator( TMIDictionaryAsHashTable<T,A> & t )
```

Description

Constructs an iterator object that traverses a TMIDictionaryAsHashTable container.

TMIDictionaryAsHashTableIterator::Current

TMIDictionaryAsHashTableIterator class

Syntax

```
T *Current()
```

Description

Returns a pointer to the current object.

TMIDictionaryAsHashTableIterator::Restart

TMIDictionaryAsHashTableIterator class

Syntax

```
void Restart();
```

Description

Restarts iteration from the beginning of the dictionary.

TMIDictionaryAsHashTableIterator::operator int

TMIDictionaryAsHashTableIterator class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMIDictionaryAsHashTableIterator::operator ++

TMIDictionaryAsHashTableIterator class

Form 1

```
T *operator ++ (int)
```

Form 2

```
T *operator ++ ()
```

Description

Form 1: Moves to the next object, and returns a pointer to the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns a pointer to the object that was current after the move (pre-increment).

TIDictionaryAsHashTable template

Syntax

```
template <class T> class TIDictionaryAsHashTable;
```

Header File

dict.h

Description

Implements an indirect dictionary using a hash table as the underlying FDS, and using the system storage allocator TStandardAllocator. It assumes that T is one of the four types of associations.

Public Constructor

TIDictionaryAsHashTable::TIDictionaryAsHashTable

Public Member Functions

Add

Detach

Find

Flush

ForEach

GetItemsInContainer

IsEmpty

TIDictionaryAsHashTable::TIDictionaryAsHashTable

TIDictionaryAsHashTable class

Syntax

```
TIDictionaryAsHashTable( unsigned size = DEFAULT_HASH_TABLE_SIZE )
```

Description

Constructs an indirect dictionary with the specified size.

TIDictionaryAsHashTable::Add

TIDictionaryAsHashTable class

Syntax

```
int Add( T * t )
```

Description

Adds a pointer to item t if not already in the dictionary.

TIDictionaryAsHashTable::Detach

TIDictionaryAsHashTable class

Syntax

```
int Detach( T * t, int del = 0 )
```

Description

Removes the pointer to item t from the dictionary, and deletes if del is 1. If del is 0 the item is not deleted.

TIDictionaryAsHashTable::Find

TIDictionaryAsHashTable class

Syntax

```
T * Find( T * t )
```

Description

Returns a pointer to item t.

TIDictionaryAsHashTable::Flush

TIDictionaryAsHashTable class

Syntax

```
void Flush( int del = 0 )
```

Description

Removes all items from the dictionary. The item is deleted if del is 1. If del is 0 the item is not deleted.

TIDictionaryAsHashTable::ForEach

TIDictionaryAsHashTable class

Syntax

```
void ForEach( IterFunc iter, void * args );
```

Description

Creates an internal iterator that executes the given function iter for each item in the container. The args argument lets you pass arbitrary data to this function.

TIDictionaryAsHashTable::GetItemsInContainer

TIDictionaryAsHashTable class

Syntax

```
inline unsigned GetItemsInContainer()
```

Description

Returns the number of items in the dictionary.

TIDictionaryAsHashTable::IsEmpty

TIDictionaryAsHashTable class

Syntax

```
inline int IsEmpty()
```

Description

Returns 1 if the dictionary is empty; otherwise returns 0.

TIDictionaryAsHashTableIterator template

Syntax

```
template <class T> class TIDictionaryAsHashTableIterator;
```

Header File

dict.h

Description

Implements an iterator that traverses TIDictionaryAsHashTable objects, using the user-supplied storage allocator A.

Public Constructor

TIDictionaryAsHashTableIterator::TIDictionaryAsHashTableIterator

Public Member Functions

Current

Restart

Operators

int

++

TIDictionaryAsHashTableIterator::TIDictionaryAsHashTableIterator

TIDictionaryAsHashTableIterator class

Syntax

```
TIDictionaryAsHashTableIterator( TIDictionaryAsHashTable<T> & t )
```

Description

Constructs an iterator object that traverses a TIDictionaryAsHashTable container.

TIDictionaryAsHashTableIterator::Current

TIDictionaryAsHashTableIterator class

Syntax

```
T *Current()
```

Description

Returns a pointer to the current object.

TIDictionaryAsHashTableIterator::Restart

TIDictionaryAsHashTableIterator class

Syntax

```
void Restart();
```

Description

Restarts iteration from the beginning of the dictionary.

TIDictionaryAsHashTableIterator::operator int

TIDictionaryAsHashTableIterator class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TIDictionaryAsHashTableIterator::operator ++

TIDictionaryAsHashTableIterator class

Form 1

```
T *operator ++ (int)
```

Form 2

```
T *operator ++ ()
```

Description

Form 1: Moves to the next object, and returns a pointer to the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns a pointer to the object that was current after the move (pre-increment).

TDictionary template

Header File
dict.h

Description
A simplified name for TDictionaryAsHashTable.

Public Constructor
TDictionary::TDictionary

TDictionary::TDictionary

TDictionary class

Syntax

```
TDictionary( unsigned size = DEFAULT_HASH_TABLE_SIZE )
```

Description

Constructs a dictionary with the specified size.

TDictionaryIterator template

Header File
dict.h

Description
A simplified name for TDictionaryAsHashTableIterator.

Public Constructor
TDictionaryIterator::TDictionaryIterator

TDictionaryIterator::TDictionaryIterator

TDictionaryIterator class

Syntax

```
TDictionaryIterator( const TDictionary<T> & a )
```

Description

Constructs an iterator object that traverses a TDictionary container.

TMDoubleListElement template

Syntax

```
template <class T, class Alloc> class TMDoubleListElement;
```

Header File

dlistimp.h

Description

This class defines the nodes for double-list classes TMDoubleListImp and TMIDoubleListImp.

Public Data Members

data

Next

Prev

Public Constructors

TMDoubleListElement::TMDoubleListElement

Operators

delete

new

TMDoubleListElement::data

TMDoubleListElement class

Syntax

```
T data;
```

Description

Data object contained in the double list.

TMDoubleListElement::Next

TMDoubleListElement class

Syntax

```
TMDoubleListElement<T> *Next;
```

Description

A pointer to the next element in the double list.

TMDoubleListElement::Prev

TMDoubleListElement class

Syntax

```
TMDoubleListElement<T> *Prev;
```

Description

A pointer to the previous element in the double list.

TMDoubleListElement::TMDoubleListElement

TMDoubleListElement class

Form 1

```
TMDoubleListElement();
```

Form 2

```
TMDoubleListElement( T& t, TMDoubleListElement<T> *p )
```

Description

Form 1: Constructs a double-list element.

Form 2: Constructs a double-list element, and inserts after the object pointed to by p.

TMDoubleListElement::operator delete

TMDoubleListElement class

Syntax

```
void operator delete( void * );
```

Description

Deletes an object.

TMDoubleListElement::operator new

TMDoubleListElement class

Syntax

```
void *operator new( size_t sz );
```

Description

Allocates a memory block of sz amount, and returns a pointer to the memory block.

TMDoubleListImp template

Syntax

```
template <class T, class Alloc> class TMDoubleListImp;
```

Header File

dlistimp.h

Description

Implements a managed, double-linked list of objects of type T. Assumes that T has meaningful copy semantics, operator ==, and a default constructor.

Type Definitions

CondFunc

IterFunc

Public Constructor

TMDoubleListImp::TMDoubleListImp

Public Member Functions

Add

AddAtHead

AddAtTail

Detach

DetachAtHead

FirstThat

Flush

ForEach

IsEmpty

LastThat

PeekHead

PeekTail

Protected Data Members

Head

Tail

Protected Member Functions

FindDetach

FindPred

TMDoubleListImp::CondFunc

TMDoubleListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMDoubleListImp::IterFunc

TMDoubleListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMDoubleListImp::TMDoubleListImp

TMDoubleListImp class

Syntax

`TMDoubleListImp()`

Description

Constructs an empty, managed, double-linked list.

TMDoubleListImp::Add

TMDoubleListImp class

Syntax

```
int Add( const T& t );
```

Description

Add the given object at the beginning of the list.

TMDoubleListImp::AddAtHead

TMDoubleListImp class

Syntax

```
int AddAtHead( const T& t );
```

Description

Add the given object at the beginning of the list.

TMDoubleListImp::AddAtTail

TMDoubleListImp class

Syntax

```
int AddAtTail( const T& );
```

Description

Adds the given object at the end (tail) of the list.

TMDoubleListImp::Detach

See Also TMDoubleListImp class

Syntax

```
int Detach( const T& );
```

Description

Removes the first occurrence of the given object encountered by searching from the beginning of the list.

See Also
TShouldDelete

TMDoubleListImp::DetachAtHead

TMDoubleListImp class

Syntax

```
int DetachAtHead();
```

Description

Removes items from the head of a list without searching for a match.

TMDoubleListImp::FirstThat

TMDoubleListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the double-list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

TMDoubleListImp::Flush

TMDoubleListImp class

Syntax

```
void Flush();
```

Description

Removes all elements from the list without destroying the list.

TMDoubleListImp::ForEach

TMDoubleListImp class

Syntax

```
void ForEach(IterFunc iter, void *args );
```

Description

ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMDoubleListImp::IsEmpty

TMDoubleListImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if array contains no elements; otherwise returns 0.

TMDoubleListImp::LastThat

See Also TMDoubleListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the double list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TMDoubleListImp::FirstThat

TMDoubleListImp::ForEach

TMDoubleListImp::PeekHead

TMDoubleListImp class

Syntax

```
Const T& PeekHead() const;
```

Description

Returns a reference to the Head item in the double list, without removing it.

TMDoubleListImp::PeekTail

TMDoubleListImp class

Syntax

```
Const T& PeekTail() const;
```

Description

Returns a reference to the Tail item in the double list, without removing it.

TMDoubleListImp::Head

TMDoubleListImp class

Syntax

```
TMDoubleListElement<T> Head;
```

Description

The head item of the double list.

TMDoubleListImp::Tail

TMDoubleListImp class

Syntax

```
TMDoubleListElement<T> Tail;
```

Description

The tail item of the double list.

TMDoubleListImp::FindDetach

TMDoubleListImp class

Syntax

```
virtual TMDoubleListElement<T> *FindDetach( const T& t )
```

Description

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

TMDoubleListImp::FindPred

TMDoubleListImp class

Syntax

```
virtual TMDoubleListElement<T> *FindPred( const T& );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TMDoubleListIteratorImp template

Syntax

```
template <class T, class Alloc> class TMDoubleListIterator;
```

Header File

dlistimp.h

Description

Implements a double list iterator. This iterator works with any direct double-linked list.

Public Constructors

TMDoubleListIteratorImp::TMDoubleListIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

--

TMDoubleListIteratorImp::TMDoubleListIteratorImp

TMDoubleListIteratorImp class

Form 1

```
TMDoubleListIteratorImp( const TMDoubleListImp<T, Alloc> &l )
```

Form 2

```
TMDoubleListIteratorImp( const TMSDoubleListImp<T, Alloc> &l )
```

Description

Constructs an iterator that traverses TMDoubleListImp objects.

TMDoubleListIteratorImp::Current

TMDoubleListIteratorImp class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TMDoubleListIteratorImp::Restart

TMDoubleListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TMDoubleListIteratorImp::operator int

TMDoubleListIteratorImp class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMDoubleListIteratorImp::operator ++

TMDoubleListIteratorImp class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMDoubleListIteratorImp::operator - -

TMDoubleListIteratorImp class

Form 1

```
const T& operator -- ( int )
```

Form 2

```
const T& operator -- ()
```

Description

Form 1: Moves to the previous object, and returns the object that was current before the move (post-decrement).

Form 2: Moves to the previous object, and returns the object that was current after the move (pre-decrement).

TDoubleListImp template

Syntax

```
template <class T> class TMDoubleListImp;
```

Header File

dlistimp.h

Description

Implements a double-linked list of objects of type T, using TStandardAllocator for memory management. Assumes that T has meaningful copy semantics and a default constructor.

Type Definitions

CondFunc

IterFunc

Public Constructor

TDoubleListImp::TDoubleListImp

Public Member Functions

Add

AddAtHead

AddAtTail

Detach

DetachAtHead

FirstThat

Flush

ForEach

IsEmpty

LastThat

PeekHead

PeekTail

Protected Data Members

Head

Tail

Protected Member Functions

FindDetach

FindPred

TDoubleListImp::CondFunc

TDoubleListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TDoubleListImp::IterFunc

TDoubleListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TDoubleListImp::TDoubleListImp

TDoubleListImp class

Syntax

```
TDoubleListImp()
```

Description

Constructs an empty double-linked list.

TDoubleListImp::Add

TDoubleListImp class

Syntax

```
int Add( const T& t );
```

Description

Add the given object at the beginning of the list.

TDoubleListImp::AddAtHead

TDoubleListImp class

Syntax

```
int AddAtHead( const T& t );
```

Description

Add the given object at the beginning of the list.

TDoubleListImp::AddAtTail

TDoubleListImp class

Syntax

```
int AddAtTail( const T& );
```

Description

Adds the given object at the end (tail) of the list.

TDoubleListImp::Detach

See Also TDoubleListImp class

Syntax

```
int Detach( const T& );
```

Description

Removes the first occurrence of the given object encountered by searching from the beginning of the list.

See Also
TShouldDelete

TDoubleListImp::DetachAtHead

TDoubleListImp class

Syntax

```
int DetachAtHead();
```

Description

Removes items from the head of a list without searching for a match.

TDoubleListImp::FirstThat

TDoubleListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the double-list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

TDoubleListImp::Flush

TDoubleListImp class

Syntax

```
void Flush();
```

Description

Removes all elements from the list without destroying the list.

TDoubleListImp::ForEach

TDoubleListImp class

Syntax

```
void ForEach(IterFunc iter, void *args );
```

Description

ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TDoubleListImp::IsEmpty

TDoubleListImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if array contains no elements; otherwise returns 0.

TDoubleListImp::LastThat

See Also TDoubleListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the double list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TDoubleListImp::FirstThat

TDoubleListImp::ForEach

TDoubleListImp::PeekHead

TDoubleListImp class

Syntax

```
Const T& PeekHead() const;
```

Description

Returns a reference to the Head item in the double list, without removing it.

TDoubleListImp::PeekTail

TDoubleListImp class

Syntax

```
Const T& PeekTail() const;
```

Description

Returns a reference to the Tail item in the double list, without removing it.

TDoubleListImp::Head

TDoubleListImp class

Syntax

```
TMDoubleListElement<T> Head;
```

Description

The head item of the double list.

TDoubleListImp::Tail

TDoubleListImp class

Syntax

```
TMDoubleListElement<T> Tail;
```

Description

The tail item of the double list.

TDoubleListImp::FindDetach

TDoubleListImp class

Syntax

```
virtual TMDoubleListElement<T> *FindDetach( const T& t )
```

Description

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

TDoubleListImp::FindPred

TDoubleListImp class

Syntax

```
virtual TMDoubleListElement<T> *FindPred( const T& );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TDoubleListIteratorImp template

Syntax

```
template <class T> class TDoubleListIteratorImp;
```

Header File

dlistimp.h

Description

Implements a double list iterator. This iterator works with any direct double-linked list.

Public Constructor

TDoubleListIteratorImp::TDoubleListIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

--

TDoubleListIteratorImp::TDoubleListIteratorImp

TDoubleListIteratorImp class

Syntax

```
TDoubleListIteratorImp( const TDoubleListImp<T> &l )
```

Description

Constructs an iterator that traverses TDoubleListImp objects.

TDoubleListIteratorImp::Current

TDoubleListIteratorImp class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TDoubleListIteratorImp::Restart

TDoubleListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TDoubleListIteratorImp::operator int

TDoubleListIteratorImp class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TDoubleListIteratorImp::operator ++

TDoubleListIteratorImp class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TDoubleListIteratorImp::operator - -

TDoubleListIteratorImp class

Form 1

```
const T& operator -- ( int )
```

Form 2

```
const T& operator -- ()
```

Description

Form 1: Moves to the previous object, and returns the object that was current before the move (post-decrement).

Form 2: Moves to the previous object, and returns the object that was current after the move (pre-decrement).

TMSDoubleListImp template

Syntax

```
template <class T, class Alloc> class TMSDoubleListImp;
```

Header File

dlistimp.h

Description

Implements a managed, sorted, double-linked list of objects of type T. It assumes that T has meaningful copy semantics, a == operator, a < operator, and a default constructor.

In addition to the member functions given here, TMSDoubleListImp inherits member functions from TMDoubleListImp.

Protected Member Functions

FindDetach

FindPred

Type Definitions

CondFunc

IterFunc

Public Constructor

TMSDoubleListImp::TMSDoubleListImp

Public Member Functions

Add

AddAtHead

AddAtTail

Detach

DetachAtHead

FirstThat

Flush

ForEach

IsEmpty

LastThat

PeekHead

PeekTail

Protected Data Members

Head

Tail

Protected Member Functions

FindDetach

FindPred

TMSDoubleListImp::CondFunc

TMSDoubleListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMSDoubleListImp::IterFunc

TMSDoubleListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMSDoubleListImp::TMSDoubleListImp

TMSDoubleListImp class

Syntax

```
TMSDoubleListImp()
```

Description

Constructs an empty, managed, double-linked list.

TMSDoubleListImp::Add

TMSDoubleListImp class

Syntax

```
int Add( const T& t );
```

Description

Add the given object at the beginning of the list.

TMSDoubleListImp::AddAtHead

TMSDoubleListImp class

Syntax

```
int AddAtHead( const T& t );
```

Description

Add the given object at the beginning of the list.

TMSDoubleListImp::AddAtTail

TMSDoubleListImp class

Syntax

```
int AddAtTail( const T& );
```

Description

Adds the given object at the end (tail) of the list.

TMSDoubleListImp::Detach

See Also TMSDoubleListImp class

Syntax

```
int Detach( const T&, int = 0 );
```

Description

Removes the first occurrence of the given object encountered by searching from the beginning of the list.

See Also
TShouldDelete

TMSDoubleListImp::DetachAtHead

TMSDoubleListImp class

Syntax

```
int DetachAtHead();
```

Description

Removes items from the head of a list without searching for a match.

TMSDoubleListImp::FirstThat

TMSDoubleListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the double-list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

TMSDoubleListImp::Flush

TMSDoubleListImp class

Syntax

```
void Flush( int = 0 );
```

Description

Removes all elements from the list without destroying the list.

TMSDoubleListImp::ForEach

TMSDoubleListImp class

Syntax

```
void ForEach(IterFunc iter, void *args );
```

Description

ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMSDoubleListImp::IsEmpty

TMSDoubleListImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if array contains no elements; otherwise returns 0.

TMSDoubleListImp::LastThat

See Also TMSDoubleListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the double list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TMSDoubleListImp::FirstThat

TMSDoubleListImp::ForEach

TMSDoubleListImp::PeekHead

TMSDoubleListImp class

Syntax

```
Const T& PeekHead() const;
```

Description

Returns a reference to the Head item in the double list, without removing it.

TMSDoubleListImp::PeekTail

TMSDoubleListImp class

Syntax

```
Const T& PeekTail() const;
```

Description

Returns a reference to the Tail item in the double list, without removing it.

TMSDoubleListImp::Head

TMSDoubleListImp class

Syntax

```
TMDoubleListElement<T> Head;
```

Description

The head item of the double list.

TMSDoubleListImp::Tail

TMSDoubleListImp class

Syntax

```
TMDoubleListElement<T> Tail;
```

Description

The tail item of the double list.

TMSDoubleListImp::FindDetach

TMSDoubleListImp class

Syntax

```
virtual TMDoubleListElement<T> *FindDetach( const T& t )
```

Description

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

TMSDoubleListImp::FindPred

TMSDoubleListImp class

Syntax

```
virtual TMDoubleListElement<T> *FindPred( const T& );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TMSDoubleListIteratorImp template

Syntax

```
template <class T, class Alloc> class TMSDoubleListIteratorImp;
```

Header File

dlistimp.h

Description

Implements a double list iterator. This iterator works with any direct double-linked list.

Public Constructor

TMSDoubleListIteratorImp::TMSDoubleListIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

--

TMSDoubleListIteratorImp::TMSDoubleListIteratorImp

TMSDoubleListIteratorImp class

Syntax

```
TMSDoubleListIteratorImp( const TMSDoubleListImp<T,Alloc> &l )
```

Description

Constructs an iterator that traverses TMSDoubleListImp objects.

TMSDoubleListIteratorImp::Current

TMSDoubleListIteratorImp class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TMSDoubleListIteratorImp::Restart

TMSDoubleListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TMSDoubleListIteratorImp::operator int

TMSDoubleListIteratorImp class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMSDoubleListIteratorImp::operator ++

TMSDoubleListIteratorImp class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMSDoubleListIteratorImp::operator - -

TMSDoubleListIteratorImp class

Form 1

```
const T& operator -- ( int )
```

Form 2

```
const T& operator -- ()
```

Description

Form 1: Moves to the previous object, and returns the object that was current before the move (post-decrement).

Form 2: Moves to the previous object, and returns the object that was current after the move (pre-decrement).

TSDoubleListImp template

Syntax

```
template <class T> class TSDoubleListImp;
```

Header File

dlistimp.h

Description

Implements a sorted, double-linked list of objects of type T. It assumes that T has meaningful copy semantics, a meaningful < operator, and a default constructor.

Type Definitions

CondFunc

IterFunc

Public Constructor

TSDoubleListImp::TSDoubleListImp

Public Member Functions

Add

AddAtHead

AddAtTail

Detach

DetachAtHead

FirstThat

Flush

ForEach

IsEmpty

LastThat

PeekHead

PeekTail

Protected Data Members

Head

Tail

Protected Member Functions

FindDetach

FindPred

TSDoubleListImp::CondFunc

TSDoubleListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TSDoubleListImp::IterFunc

TSDoubleListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TSDoubleListImp::TSDoubleListImp

TSDoubleListImp class

Syntax

```
TSDoubleListImp()
```

Description

Constructs an empty, managed, double-linked list.

TSDoubleListImp::Add

TSDoubleListImp class

Syntax

```
int Add( const T& t );
```

Description

Add the given object at the beginning of the list.

TSDoubleListImp::AddAtHead

TSDoubleListImp class

Syntax

```
int AddAtHead( const T& t );
```

Description

Add the given object at the beginning of the list.

TSDoubleListImp::AddAtTail

TSDoubleListImp class

Syntax

```
int AddAtTail( const T& );
```

Description

Adds the given object at the end (tail) the list.

TSDoubleListImp::Detach

See Also TSDoubleListImp class

Syntax

```
int Detach( const T& );
```

Description

Removes the first occurrence of the given object encountered by searching from the beginning of the list.

See Also
TShouldDelete

TSDoubleListImp::DetachAtHead

TSDoubleListImp class

Syntax

```
int DetachAtHead();
```

Description

Removes items from the head of a list without searching for a match.

TSDoubleListImp::FirstThat

TSDoubleListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the double-list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

TSDoubleListImp::Flush

TSDoubleListImp class

Syntax

```
void Flush();
```

Description

Removes all elements from the list without destroying the list.

TSDoubleListImp::ForEach

TSDoubleListImp class

Syntax

```
void ForEach(IterFunc iter, void *args );
```

Description

ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TSDoubleListImp::IsEmpty

TSDoubleListImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if array contains no elements; otherwise returns 0.

TSDoubleListImp::LastThat

See Also   TSDoubleListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the double list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TSDoubleListImp::FirstThat

TSDoubleListImp::ForEach

TSDoubleListImp::PeekHead

TSDoubleListImp class

Syntax

```
Const T& PeekHead() const;
```

Description

Returns a reference to the Head item in the double list, without removing it.

TSDoubleListImp::PeekTail

TSDoubleListImp class

Syntax

```
Const T& PeekTail() const;
```

Description

Returns a reference to the Tail item in the double list, without removing it.

TSDoubleListImp::Head

TSDoubleListImp class

Syntax

```
TMDoubleListElement<T> Head;
```

Description

The head item of the double list.

TSDoubleListImp::Tail

TSDoubleListImp class

Syntax

```
TMDoubleListElement<T> Tail;
```

Description

The tail item of the double list.

TSDoubleListImp::FindDetach

TSDoubleListImp class

Syntax

```
virtual TMDoubleListElement<T> *FindDetach( const T& t )
```

Description

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

TSDoubleListImp::FindPred

TSDoubleListImp class

Syntax

```
virtual TMDoubleListElement<T> *FindPred( const T& );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TSDoubleListIteratorImp template

Syntax

```
template <class T> class TSDoubleListIteratorImp;
```

Header File

dlistimp.h

Description

Implements a double list iterator. This iterator works with any direct double-linked list.

Public Constructor

TSDoubleListIteratorImp::TSDoubleListIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

--

TSDoubleListIteratorImp::TSDoubleListIteratorImp

TSDoubleListIteratorImp class

Syntax

```
TSDoubleListIteratorImp( const TSDoubleListImp<T> &l )
```

Description

Constructs an iterator that traverses TSDoubleListImp objects.

TSDoubleListIteratorImp::Current

TSDoubleListIteratorImp class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TSDoubleListIteratorImp::Restart

TSDoubleListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TSDoubleListIteratorImp::operator int

TSDoubleListIteratorImp class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TSDoubleListIteratorImp::operator ++

TSDoubleListIteratorImp class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TSDoubleListIteratorImp::operator - -

TSDoubleListIteratorImp class

Form 1

```
const T& operator -- ( int )
```

Form 2

```
const T& operator -- ()
```

Description

Form 1: Moves to the previous object, and returns the object that was current before the move (post-decrement).

Form 2: Moves to the previous object, and returns the object that was current after the move (pre-decrement).

TMIDoubleListImp template

Syntax

```
template <class T, class Alloc> class TMIDoubleListImp;
```

Header File

dlistimp.h

Description

Implements a managed, double-linked list of pointers to objects of type T. The contained objects need a valid == operator. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Type Definitions

CondFunc

IterFunc

Public Member Functions

Add

AddAtHead

AddAtTail

Detach

DetachAtHead

DetachAtTail

FirstThat

Flush

ForEach

GetItemsInContainer

IsEmpty

LastThat

PeekHead

PeekTail

Protected Member Functions

FindPred

TMIDoubleListImp::CondFunc

TMIDoubleListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMIDoubleListImp::IterFunc

TMIDoubleListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMIDoubleListImp::Add

TMIDoubleListImp class

Syntax

```
int Add( T *t )
```

Description

Adds an object pointer to the double list.

TMIDoubleListImp::AddAtHead

TMIDoubleListImp class

Syntax

```
int AddAtHead( T *t );
```

Description

Add the given object at the beginning of the list.

TMIDoubleListImp::AddAtTail

TMIDoubleListImp class

Syntax

```
int AddAtTail( T *t )
```

Description

Adds an object pointer to the tail of the double list.

TMIDoubleListImp::Detach

See Also TMIDoubleListImp class

Syntax

```
int Detach( T *t, int del = 0 )
```

Description

Removes the given object pointer from the list. The second argument specifies whether the object should be deleted.

See Also
TShouldDelete

TMIDoubleListImp::DetachAtHead

TMIDoubleListImp class

Syntax

```
int DetachAtHead( int del = 0 )
```

Description

Deletes the object pointer from the head of the list.

TMIDoubleListImp::DetachAtTail

TMIDoubleListImp class

Syntax

```
int DetachAtTail( int del = 0 )
```

Description

Deletes the object pointer from the tail of the list.

TMIDoubleListImp::FirstThat

See Also   TMIDoubleListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the double list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
TMIDoubleListImp::LastThat

TMIDoubleListImp::Flush

TMIDoubleListImp class

Syntax

```
void Flush( int = 0 );
```

Description

Removes all elements from the list without destroying the list.

TMIDoubleListImp::ForEach

TMIDoubleListImp class

Syntax

```
void ForEach(IterFunc iter, void *args );
```

Description

Executes function iter for each double-list element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMIDoubleListImp::GetItemsInContainer

TMIDoubleListImp class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the array.

TMIDoubleListImp::IsEmpty

TMIDoubleListImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if array contains no elements; otherwise returns 0.

TMIDoubleListImp::LastThat

See Also TMIDoubleListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TMIDoubleListImp::FirstThat

TMIDoubleListImp::ForEach

TMIDoubleListImp::PeekHead

TMIDoubleListImp class

Syntax

```
T *PeekHead() const;
```

Description

Returns the object pointer at the Head of the list, without removing it.

TMIDoubleListImp::PeekTail

TMIDoubleListImp class

Syntax

```
T *PeekTail() const;
```

Description

Returns the object pointer at the Tail of the list, without removing it.

TMIDoubleListImp::FindPred

TMIDoubleListImp class

Syntax

```
virtual TDoubleListElement<void *> *FindPred( void * );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TMIDoubleListIteratorImp template

Syntax

```
template <class T, class Alloc> class TMIDoubleListIteratorImp;
```

Header File

dlistimp.h

Description

Implements a double list iterator. This iterator works with any indirect double list.

Public Constructor

TMIDoubleListIteratorImp::TMIDoubleListIteratorImp

Public Member Functions

Current

Restart

Operators

++

TMIDoubleListIteratorImp::TMIDoubleListIteratorImp

TMIDoubleListIteratorImp class

Syntax

```
TMIDoubleListIteratorImp( const TMIDoubleListImp<T,Alloc> &l )
```

Description

Constructs an object that iterates on TIDoubleListImp objects.

TMIDoubleListIteratorImp::Current

TMIDoubleListIteratorImp class

Syntax

```
T *Current()
```

Description

Returns the current object pointer.

TMIDoubleListIteratorImp::Restart

TMIDoubleListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TMIDoubleListIteratorImp::operator ++

TMIDoubleListIteratorImp class

Form 1

```
T *operator ++ (int)
```

Form 2

```
T *operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TIDoubleListImp template

Syntax

```
template <class T> class TIDoubleListImp;
```

Header File

dlistimp.h

Description

Implements a double-linked list of pointers to objects of type T, using TStandardAllocator for memory management. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Type Definitions

CondFunc

IterFunc

Public Member Functions

Add

AddAtHead

AddAtTail

Detach

DetachAtHead

DetachAtTail

FirstThat

Flush

ForEach

GetItemsInContainer

IsEmpty

LastThat

PeekHead

PeekTail

Protected Member Functions

FindPred

TIDoubleListImp::CondFunc

TIDoubleListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TIDoubleListImp::IterFunc

TIDoubleListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TIDoubleListImp::Add

TIDoubleListImp class

Syntax

```
int Add( T *t )
```

Description

Adds an object pointer to the double list.

TIDoubleListImp::AddAtHead

TIDoubleListImp class

Syntax

```
int AddAtHead( T *t );
```

Description

Add the given object at the beginning of the list.

TIDoubleListImp::AddAtTail

TIDoubleListImp class

Syntax

```
int AddAtTail( T *t )
```

Description

Adds an object pointer to the tail of the double list.

TIDoubleListImp::Detach

See Also TIDoubleListImp class

Syntax

```
int Detach( T *t, int del = 0 )
```

Description

Removes the given object pointer from the list. The second argument specifies whether the object should be deleted.

See Also
TShouldDelete

TIDoubleListImp::DetachAtHead

TIDoubleListImp class

Syntax

```
int DetachAtHead( int del = 0 )
```

Description

Deletes the object pointer from the head of the list.

TIDoubleListImp::DetachAtTail

TIDoubleListImp class

Syntax

```
int DetachAtTail( int del = 0 )
```

Description

Deletes the object pointer from the tail of the list.

TIDoubleListImp::FirstThat

See Also TIDoubleListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the double list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
TIDoubleListImp::LastThat

TIDoubleListImp::Flush

TIDoubleListImp class

Syntax

```
void Flush( int = 0 );
```

Description

Removes all elements from the list without destroying the list.

TIDoubleListImp::ForEach

TIDoubleListImp class

Syntax

```
void ForEach(IterFunc iter, void *args );
```

Description

Executes function iter for each double-list element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TIDoubleListImp::GetItemsInContainer

TIDoubleListImp class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the array.

TIDoubleListImp::IsEmpty

TIDoubleListImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if array contains no elements; otherwise returns 0.

TIDoubleListImp::LastThat

See Also TIDoubleListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TIDoubleListImp::FirstThat

TIDoubleListImp::ForEach

TIDoubleListImp::PeekHead

TIDoubleListImp class

Syntax

```
T *PeekHead() const;
```

Description

Returns the object pointer at the Head of the list, without removing it.

TIDoubleListImp::PeekTail

TIDoubleListImp class

Syntax

```
T *PeekTail() const;
```

Description

Returns the object pointer at the Tail of the list, without removing it.

TIDoubleListImp::FindPred

TIDoubleListImp class

Syntax

```
virtual TDoubleListElement<void *> *FindPred( void * );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TIDoubleListIteratorImp template

Syntax

```
template <class T> class TIDoubleListIteratorImp;
```

Header File

dlistimp.h

Description

Implements a double list iterator. This iterator works with any indirect double list.

Public Constructor

TIDoubleListIteratorImp::TIDoubleListIteratorImp

Public Member Functions

Current

Restart

Operators

++

TIDoubleListIteratorImp::TIDoubleListIteratorImp

TIDoubleListIteratorImp class

Syntax

```
TIDoubleListIteratorImp( const TIDoubleListImp<T> &l )
```

Description

Constructs an object that iterates on TIDoubleListImp objects.

TIDoubleListIteratorImp::Current

TIDoubleListIteratorImp class

Syntax

```
T *Current()
```

Description

Returns the current object pointer.

TIDoubleListIteratorImp::Restart

TIDoubleListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TIDoubleListIteratorImp::operator ++

TIDoubleListIteratorImp class

Form 1

```
T *operator ++ (int)
```

Form 2

```
T *operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMISDoubleListImp template

Syntax

```
template <class T, class Alloc> class TMISDoubleListImp;
```

Header File

dlistimp.h

Description

Implements a managed, sorted, double-linked list of pointers to objects of type T. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Type Definitions

CondFunc

IterFunc

Public Member Functions

Add

AddAtHead

AddAtTail

Detach

DetachAtHead

DetachAtTail

FirstThat

Flush

ForEach

GetItemsInContainer

IsEmpty

LastThat

PeekHead

PeekTail

Protected Member Functions

FindDetach

FindPred

TMISDoubleListImp::CondFunc

TMISDoubleListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMISDoubleListImp::IterFunc

TMISDoubleListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMISDoubleListImp::Add

TMISDoubleListImp class

Syntax

```
int Add( T *t )
```

Description

Adds an object pointer to the double list.

TMISDoubleListImp::AddAtHead

TMISDoubleListImp class

Syntax

```
int AddAtHead( T *t );
```

Description

Add the given object at the beginning of the list.

TMISDoubleListImp::AddAtTail

TMISDoubleListImp class

Syntax

```
int AddAtTail( T *t )
```

Description

Adds an object pointer to the tail of the double list.

TMISDoubleListImp::Detach

See Also TMISDoubleListImp class

Syntax

```
int Detach( T *t, int del = 0 )
```

Description

Removes the given object pointer from the list. The second argument specifies whether the object should be deleted.

See Also
TShouldDelete

TMISDoubleListImp::DetachAtHead

TMISDoubleListImp class

Syntax

```
int DetachAtHead( int del = 0 )
```

Description

Deletes the object pointer from the head of the list.

TMISDoubleListImp::DetachAtTail

TMISDoubleListImp class

Syntax

```
int DetachAtTail( int del = 0 )
```

Description

Deletes the object pointer from the tail of the list.

TMISDoubleListImp::FirstThat

See Also   TMISDoubleListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the double list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
TMISDoubleListImp::LastThat

TMISDoubleListImp::Flush

TMISDoubleListImp class

Syntax

```
void Flush( int = 0 );
```

Description

Removes all elements from the list without destroying the list.

TMISDoubleListImp::ForEach

TMISDoubleListImp class

Syntax

```
void ForEach(IterFunc iter, void *args );
```

Description

Executes function iter for each double-list element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMISDoubleListImp::GetItemsInContainer

TMISDoubleListImp class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the array.

TMISDoubleListImp::IsEmpty

TMISDoubleListImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if array contains no elements; otherwise returns 0.

TMISDoubleListImp::LastThat

See Also TMISDoubleListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TMISDoubleListImp::FirstThat

TMISDoubleListImp::ForEach

TMISDoubleListImp::PeekHead

TMISDoubleListImp class

Syntax

```
T *PeekHead() const;
```

Description

Returns the object pointer at the Head of the list, without removing it.

TMISDoubleListImp::PeekTail

TMISDoubleListImp class

Syntax

```
T *PeekTail() const;
```

Description

Returns the object pointer at the Tail of the list, without removing it.

TMISDoubleListImp::FindDetach

TMISDoubleListImp class

Syntax

```
virtual TMDoubleListElement<void *> *FindDetach( void * );
```

Description

Determines whether an object is in the list, and returns a pointer to its predecessor.

TMISDoubleListImp::FindPred

TMISDoubleListImp class

Syntax

```
virtual TDoubleListElement<void *> *FindPred( void * );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TMISDoubleListIteratorImp template

Syntax

```
template <class T, class Alloc> class TMISDoubleListIteratorImp;
```

Header File

dlistimp.h

Description

Implements a double list iterator. This iterator works with any indirect, sorted double list.

Public Constructor

TMISDoubleListIteratorImp::TMISDoubleListIteratorImp

Public Member Functions

Current

Restart

Operators

++

TMISDoubleListIteratorImp::TMISDoubleListIteratorImp

TMISDoubleListIteratorImp class

Syntax

```
TMISDoubleListIteratorImp( const TMISDoubleListImp<T,Alloc> &l )
```

Description

Constructs an object that iterates on TMISDoubleListImp objects.

TMISDoubleListIteratorImp::Current

TMISDoubleListIteratorImp class

Syntax

```
T *Current()
```

Description

Returns the current object pointer.

TMISDoubleListIteratorImp::Restart

TMISDoubleListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TMISDoubleListIteratorImp::operator ++

TMISDoubleListIteratorImp class

Form 1

```
T *operator ++ (int)
```

Form 2

```
T *operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TISDoubleListImp template

Syntax

```
template <class T> class TISDoubleListImp;
```

Header File

dlistimp.h

Description

Implements a sorted, double-linked list of pointers to objects of type T, using TStandardAllocator for memory management. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Type Definitions

CondFunc

IterFunc

Public Member Functions

Add

AddAtHead

AddAtTail

Detach

DetachAtHead

DetachAtTail

FirstThat

Flush

ForEach

GetItemsInContainer

IsEmpty

LastThat

PeekHead

PeekTail

Protected Member Functions

FindPred

TISDoubleListImp::CondFunc

TISDoubleListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TISDoubleListImp::IterFunc

TISDoubleListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TISDoubleListImp::Add

TISDoubleListImp class

Syntax

```
int Add( T *t )
```

Description

Adds an object pointer to the double list.

TISDoubleListImp::AddAtHead

TISDoubleListImp class

Syntax

```
int AddAtHead( T *t );
```

Description

Add the given object at the beginning of the list.

TISDoubleListImp::AddAtTail

TISDoubleListImp class

Syntax

```
int AddAtTail( T *t )
```

Description

Adds an object pointer to the tail of the double list.

TISDoubleListImp::Detach

See Also TISDoubleListImp class

Syntax

```
int Detach( T *t, int del = 0 )
```

Description

Removes the given object pointer from the list. The second argument specifies whether the object should be deleted.

See Also
TShouldDelete

TISDoubleListImp::DetachAtHead

TISDoubleListImp class

Syntax

```
int DetachAtHead( int del = 0 )
```

Description

Deletes the object pointer from the head of the list.

TISDoubleListImp::DetachAtTail

TISDoubleListImp class

Syntax

```
int DetachAtTail( int del = 0 )
```

Description

Deletes the object pointer from the tail of the list.

TISDoubleListImp::FirstThat

See Also TISDoubleListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the double list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
TISDoubleListImp::LastThat

TISDoubleListImp::Flush

TISDoubleListImp class

Syntax

```
void Flush( int = 0 );
```

Description

Removes all elements from the list without destroying the list.

TISDoubleListImp::ForEach

TISDoubleListImp class

Syntax

```
void ForEach(IterFunc iter, void *args );
```

Description

Executes function iter for each double-list element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TISDoubleListImp::GetItemsInContainer

TISDoubleListImp class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the array.

TISDoubleListImp::IsEmpty

TISDoubleListImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if array contains no elements; otherwise returns 0.

TISDoubleListImp::LastThat

See Also TISDoubleListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

TISDoubleListImp::FirstThat

TISDoubleListImp::ForEach

TISDoubleListImp::PeekHead

TISDoubleListImp class

Syntax

```
T *PeekHead() const;
```

Description

Returns the object pointer at the Head of the list, without removing it.

TISDoubleListImp::PeekTail

TISDoubleListImp class

Syntax

```
T *PeekTail() const;
```

Description

Returns the object pointer at the Tail of the list, without removing it.

TISDoubleListImp::FindPred

TISDoubleListImp class

Syntax

```
virtual TDoubleListElement<void *> *FindPred( void * );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TISDoubleListIteratorImp template

Syntax

```
template <class T> class TISDoubleListImp;
```

Header File

dlistimp.h

Description

Implements a double list iterator. This iterator works with any indirect, sorted double list.

Public Constructor

TISDoubleListIteratorImp::TISDoubleListIteratorImp

Public Member Functions

Current

Restart

Operators

++

TISDoubleListIteratorImp::TISDoubleListIteratorImp

TISDoubleListIteratorImp class

Syntax

```
TISDoubleListIteratorImp( const TISDoubleListImp<T> &l )
```

Description

Constructs an object that iterates on TMISDoubleListImp objects.

TISDoubleListIteratorImp::Current

TISDoubleListIteratorImp class

Syntax

```
T *Current()
```

Description

Returns the current object pointer.

TISDoubleListIteratorImp::Restart

TISDoubleListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TISDoubleListIteratorImp::operator ++

TISDoubleListIteratorImp class

Form 1

```
T *operator ++ (int)
```

Form 2

```
T *operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMHashTableImp template

Syntax

```
template <class T, class Alloc> class TMHashTableImp;
```

Header File

hashimp.h

Description

Implements a managed hash table of objects of type T, using the user-supplied storage allocator A. It assumes that T has meaningful copy and == semantics, as well as a default constructor.

Public Constructors

TMHashTableImp::TMHashTableImp

Public Member Functions

Add

Detach

Find

Flush

ForEach

GetItemsInContainer

IsEmpty

TMHashTableImp::TMHashTableImp

TMHashTableImp class

Constructor

```
TMHashTableImp( unsigned aPrime = DEFAULT::HASH::TABLE::SIZE )
```

Description

Constructs a hash table.

Description

Calls member function Flush to delete the container.

TMHashTableImp::Add

TMHashTableImp class

Syntax

```
int Add( const T& t );
```

Description

Adds item t to the hash table.

TMHashTableImp::Detach

TMHashTableImp class

Syntax

```
int Detach( const T& t, int del=0 );
```

Description

Removes item t from the hash table. If del is set to 0, t is deleted; if del is set to 1, t is not deleted.

TMHashTableImp::Find

TMHashTableImp class

Syntax

```
T * Find( const T& t ) const;
```

Description

Returns a pointer to item t.

TMHashTableImp::Flush

TMHashTableImp class

Syntax

```
void Flush()
```

Description

Flushes all items in the hash table. The hash table is destroyed if del is nonzero.

TMHashTableImp::ForEach

TMHashTableImp class

Syntax

```
void ForEach(IterFunc iter, void *args);
```

Description

Creates an internal iterator that executes the given function f for each item in the container. The args argument lets you pass arbitrary data to this function.

TMHashTableImp::GetItemsInContainer

TMHashTableImp class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the hash table.

TMHashTableImp::IsEmpty

TMHashTableImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the hash table is empty; otherwise returns 0.

TMHashTableIteratorImp template

Syntax

```
template <class T, class Alloc> class TMHashTableIteratorImp;
```

Header File

hashimp.h

Description

Implements an iterator for traversing TMHashTableImp containers, using the user-supplied storage allocator Alloc.

Public Constructor and Destructor

TMHashTableIteratorImp::TMHashTableIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

TMHashTableIteratorImp::TMHashTableIteratorImp

TMHashTableIteratorImp class

Constructor

```
TMHashTableIteratorImp( const TMHashTableImp<T,A> & h )
```

Description

Constructs an iterator object that traverses a TMHashTableImp container.

TMHashTableIteratorImp::Current

TMHashTableIteratorImp class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TMHashTableIteratorImp::Restart

TMHashTableIteratorImp class

Syntax

```
void Restart();
```

Description

Restarts iteration from the beginning of the hash table.

TMHashTableIteratorImp::operator int

TMHashTableIteratorImp class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMHashTableIteratorImp::operator ++

TMHashTableIteratorImp class

Form 1

```
const T& operator ++ (int)
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

THashTableImp template

Syntax

```
template <class T> class THashTableImp;
```

Header File

hashimp.h

Description

Implements a hash table of objects of type T, using the system storage allocator TStandardAllocator. It assumes that T has meaningful copy and == semantics as well as a default constructor.

Public Constructor

THashTableImp::THashTableImp

Public Member Functions

Add

Detach

Find

Flush

ForEach

GetItemsInContainer

IsEmpty

THashTableImp::THashTableImp

THashTableImp class

Syntax

```
THashTableImp( unsigned aPrime = DEFAULT_HASH_TABLE_SIZE )
```

Description

Constructs a hash table that uses TStandardAllocator for memory management.

THashTableImp::Add

THashTableImp class

Syntax

```
int Add( const T& t );
```

Description

Adds item t to the hash table.

THashTableImp::Detach

THashTableImp class

Syntax

```
int Detach( const T& t, int del=0 );
```

Description

Removes item t from the hash table. If del is set to 0, t is deleted; if del is set to 1, t is not deleted.

THashTableImp::Find

THashTableImp class

Syntax

```
T * Find( const T& t ) const;
```

Description

Returns a pointer to item t.

THashTableImp::Flush

THashTableImp class

Syntax

```
void Flush()
```

Description

Flushes all items in the hash table. The hash table is destroyed if del is nonzero.

THashTableImp::ForEach

THashTableImp class

Syntax

```
void ForEach(IterFunc iter, void *args);
```

Description

Creates an internal iterator that executes the given function f for each item in the container. The args argument lets you pass arbitrary data to this function.

THashTableImp::GetItemsInContainer

THashTableImp class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the hash table.

THashTableImp::IsEmpty

THashTableImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the hash table is empty; otherwise returns 0.

THashTableIteratorImp template

Syntax

```
template <class T> class THashTableIteratorImp;
```

Header File

hashimp.h

Description

Implements an iterator for traversing THashTableImp containers.

Public Constructor

THashTableIteratorImp::THashTableIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

THashTableIteratorImp::THashTableIteratorImp

THashTableIteratorImp class

Syntax

```
THashTableIteratorImp( const THashTableImp<T,A> & h )
```

Description

Constructs an iterator object that traverses a THashTableImp container.

THashTableIteratorImp::Current

THashTableIteratorImp class

Syntax

```
const T& Current()
```

Description

Returns the current object.

THashTableIteratorImp::Restart

THashTableIteratorImp class

Syntax

```
void Restart();
```

Description

Restarts iteration from the beginning of the hash table.

THashTableIteratorImp::operator int

THashTableIteratorImp class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

THashTableIteratorImp::operator ++

THashTableIteratorImp class

Form 1

```
const T& operator ++ (int)
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMIHashTableImp template

Syntax

```
template <class T, class Alloc> class TMIHashTableImp;
```

Header File

hashimp.h

Description

Implements a managed hash table of pointers to objects of type T, using the user-supplied storage allocator Alloc.

Public Constructor

TMIHashTableImp::TMIHashTableImp

Public Member Functions

Add

Detach

Find

Flush

ForEach

GetItemsInContainer

IsEmpty

TMIHashTableImp::TMIHashTableImp

TMIHashTableImp class

Syntax

```
TMIHashTableImp( unsigned aPrime = DEFAULT_HASH_TABLE_SIZE )
```

Description

Constructs an indirect hash table.

TMIHashTableImp::Add

TMIHashTableImp class

Syntax

```
int Add( T * t )
```

Description

Adds a pointer to item t to the hash table.

TMIHashTableImp::Detach

TMIHashTableImp class

Syntax

```
int Detach( T * t, int del = 0 )
```

Description

Removes a pointer to item t from the hash table. t is deleted if del is set 1, and not deleted if del is set to 0.

TMIHashTableImp::Find

TMIHashTableImp class

Syntax

```
T * Find( const T * t ) const;
```

Description

Returns a pointer to item t.

TMIHashTableImp::Flush

TMIHashTableImp class

Syntax

```
void Flush( int del = 0 )
```

Description

Flushes all items in the hash table. The hash table is destroyed if del is nonzero.

TMIHashTableImp::ForEach

TMIHashTableImp class

Syntax

```
void ForEach(IterFunc iter, void *args);
```

Description

Creates an internal iterator that executes the given function f for each item in the container. The args argument lets you pass arbitrary data to this function.

TMIHashTableImp::GetItemsInContainer

TMIHashTableImp class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the hash table.

TMIHashTableImp::IsEmpty

TMIHashTableImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the hash table is empty; otherwise returns 0.

TMIHashTableIteratorImp template

Syntax

```
template <class T, class Alloc> class TMIHashTableIteratorImp;
```

Header File

hashimp.h

Description

Implements an iterator for traversing TMIHashTableImp containers.

Public Constructor

TMIHashTableIteratorImp::TMIHashTableIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

TMIHashTableIteratorImp::TMIHashTableIteratorImp

TMIHashTableIteratorImp class

Syntax

```
TMIHashTableIteratorImp( const TMIHashTableImp<T,A> & h )
```

Description

Constructs an iterator object that traverses a TMIHashTableImp container.

TMIHashTableIteratorImp::Current

TMIHashTableIteratorImp class

Syntax

```
T *Current()
```

Description

Returns a pointer to the current object.

TMIHashTableIteratorImp::Restart

TMIHashTableIteratorImp class

Syntax

```
void Restart();
```

Description

Restarts iteration from the beginning of the hash table.

TMIHashTableIteratorImp::operator int

TMIHashTableIteratorImp class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMIHashTableIteratorImp::operator ++

TMIHashTableIteratorImp class

Form 1

```
T *operator ++ (int)
```

Form 2

```
T *operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object pointer that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object pointer that was current after the move (pre-increment).

TIHashTableImp template

Syntax

```
template <class T> class TIHashTableImp;
```

Header File

hashimp.h

Description

Implements a hash table of pointers to objects of type T, using the system storage allocator TStandardAllocator.

Public Constructor

TIHashTableImp::TIHashTableImp

Public Member Functions

Add

Detach

Find

Flush

ForEach

GetItemsInContainer

IsEmpty

TIHashTableImp::TIHashTableImp

TIHashTableImp class

Syntax

```
TIHashTableImp( unsigned aPrime = DEFAULT_HASH_TABLE_SIZE )
```

Description

Constructs an indirect hash table that uses the system storage allocator.

TIHashTableImp::Add

TIHashTableImp class

Syntax

```
int Add( T * t )
```

Description

Adds a pointer to item t to the hash table.

TIHashTableImp::Detach

TIHashTableImp class

Syntax

```
int Detach( T * t, int del = 0 )
```

Description

Removes a pointer to item t from the hash table. t is deleted if del is set 1, and not deleted if del is set to 0.

TIHashTableImp::Find

TIHashTableImp class

Syntax

```
T * Find( const T * t ) const;
```

Description

Returns a pointer to item t.

TIHashTableImp::Flush

TIHashTableImp class

Syntax

```
void Flush( int del = 0 )
```

Description

Flushes all items in the hash table. The hash table is destroyed if del is nonzero.

TIHashTableImp::ForEach

TIHashTableImp class

Syntax

```
void ForEach(IterFunc iter, void *args);
```

Description

Creates an internal iterator that executes the given function f for each item in the container. The args argument lets you pass arbitrary data to this function.

TIHashTableImp::GetItemsInContainer

TIHashTableImp class

Syntax

```
unsigned GetItemsInContainer() const;
```

Description

Returns the number of items in the hash table.

TIHashTableImp::IsEmpty

TIHashTableImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the hash table is empty; otherwise returns 0.

TIHashTableIteratorImp template

Syntax

```
template <class T> class TIHashTableIteratorImp;
```

Header File

hashimp.h

Description

Implements an iterator object that traverses TIHashTableImp containers, and uses the system memory allocator TStandardAllocator.

Public Constructor

TIHashTableIteratorImp::TIHashTableIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

TIHashTableIteratorImp::TIHashTableIteratorImp

TIHashTableIteratorImp class

Syntax

```
TIHashTableIteratorImp( const TMIHashTableImp<T,A> & h )
```

Description

Constructs an iterator object that traverses a TMIHashTableImp container.

TIHashTableIteratorImp::Current

TIHashTableIteratorImp class

Syntax

```
T *Current()
```

Description

Returns a pointer to the current object.

TIHashTableIteratorImp::Restart

TIHashTableIteratorImp class

Syntax

```
void Restart();
```

Description

Restarts iteration from the beginning of the hash table.

TIHashTableIteratorImp::operator int

TIHashTableIteratorImp class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TIHashTableIteratorImp::operator ++

TIHashTableIteratorImp class

Form 1

```
T *operator ++ (int)
```

Form 2

```
T *operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object pointer that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object pointer that was current after the move (pre-increment).

TMListElement template

Syntax

```
template <class T, class Alloc> class TMListElement;
```

Header File

listimp.h

Description

This class defines the nodes for TMListImp and TMIListImp and related classes.

Public Constructors

TMListElement::TMListElement

Public Data Members

Data

Next

Operators

delete

new

TMListElement::data

TMListElement class

Syntax

```
T Data;
```

Description

Data object contained in the list.

TMListElement::Next

TMListElement class

Syntax

```
TMListElement<T,Alloc> *Next;
```

Description

A pointer to the next element in the list.

TMListElement::TMListElement

TMListElement class

Form 1

```
TMListElement();
```

Form 2

```
TMListElement( T& t, TMListElement<T,Alloc> *p )
```

Description

Form 1: Constructs a list element.

Form 2: Constructs a list element, and places it after the object at location p.

TMListElement::operator delete

TMListElement class

Syntax

```
void operator delete( void * );
```

Description

Deletes an object.

TMListElement::operator new

TMListElement class

Syntax

```
void *operator new( size_t sz );
```

Description

Allocates a memory block of sz amount, and returns a pointer to the memory block.

TMListImp template

Syntax

```
template <class T, class Alloc> class TMListImp;
```

Header File

listimp.h

Description

Implements a managed list of objects of type T. TMListImp assumes that T has meaningful copy semantics, and a default constructor.

Type Definitions

CondFunc

IterFunc

Public Constructor

TMListImp::TMListImp

Public Member Functions

Add

Detach

DetachAtHead

FirstThat

Flush

ForEach

IsEmpty

LastThat

PeekHead

Protected Data Members

Head

Tail

Protected Member Functions

FindDetach

FindPred

TMListImp::CondFunc

TMListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMListImp::IterFunc

TMListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMListImp::TMListImp

TMListImp class

Syntax

```
TMListImp()
```

Description

Constructs an empty list.

TMListImp::Add

TMListImp class

Syntax

```
int Add( const T& t );
```

Description

Adds an object to the list.

TMListImp::Detach

See Also TMListImp class

Syntax

```
int Detach( const T& );
```

Description

Removes the given object from the list. Returns 0 for failure, 1 for success in removing the object.

See Also
TShouldDelete

TMListImp::DetachAtHead

TMListImp class

Syntax

```
int DetachAtHead();
```

Description

Removes items from the head of a list without searching for a match.

TMListImp::FirstThat

See Also TMListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TMListImp::Flush

TMListImp class

Syntax

```
void Flush();
```

Description

Flushes the list without destroying it.

TMListImp::ForEach

TMListImp class

Syntax

```
void ForEach( IterFunc iter, void *args );
```

Description

Executes function iter for list element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMListImp::IsEmpty

TMListImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the list has no elements; otherwise returns 0.

TMListImp::LastThat

See Also   TMListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

FirstThat

ForEach

TMListImp::PeekHead

TMListImp class

Syntax

```
Const T& PeekHead() const;
```

Description

Returns a reference to the Head item in the list, without removing it.

TMListImp::Head

TMListImp class

Syntax

```
TMListElement<T,Alloc> Head;
```

Description

The element before the first element in the list.

TMListImp::Tail

TMListImp class

Syntax

```
TMListElement<T,Alloc> Tail;
```

Description

The element after the last element in the list.

TMListImp::FindDetach

TMListImp class

Syntax

```
virtual TMListElement<T,Alloc> *FindDetach( const T& t )
```

Description

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

TMListImp::FindPred

TMListImp class

Syntax

```
virtual TMListElement<T,Alloc> *FindPred( const T& );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TMListIteratorImp template

Syntax

```
template <class T, class Alloc> class TMListIteratorImp;
```

Header File

listimp.h

Description

Implements a list iterator that works on direct, managed list. For indirect list iteration see TMIListIteratorImp.

Public Constructor

TMListIteratorImp::TMListIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

TMListIteratorImp::TMListIteratorImp

TMListIteratorImp class

Syntax

```
TMListIteratorImp(const TMListImp<T,Alloc> &l)
```

Description

Constructs an iterator that traverses TMListImp objects.

TMListIteratorImp::Current

TMListIteratorImp class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TMListIteratorImp::Restart

TMListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TMListIteratorImp::operator int

TMListIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMListIteratorImp::operator ++

TMListIteratorImp class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TListImp template

Syntax

```
template <class T> class TListImp;
```

Header File

listimp.h

Description

Implements a list of objects of type T. TListImp assumes that T has meaningful copy semantics, and a default constructor.

Type Definitions

CondFunc

IterFunc

Public Constructor

TListImp::TListImp

Public Member Functions

Add

Detach

DetachAtHead

FirstThat

Flush

ForEach

IsEmpty

LastThat

PeekHead

Protected Data Members

Head

Tail

Protected Member Functions

FindDetach

FindPred

TListImp::CondFunc

TListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TListImp::IterFunc

TListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TListImp::TListImp

TListImp class

Syntax

```
TListImp()
```

Description

Constructs an empty list.

TListImp::Add

TListImp class

Syntax

```
int Add( const T& t );
```

Description

Adds an object to the list.

TListImp::Detach

See Also TListImp class

Syntax

```
int Detach( const T& );
```

Description

Removes the given object from the list. Returns 0 for failure, 1 for success in removing the object.

See Also
TShouldDelete

TListImp::DetachAtHead

TListImp class

Syntax

```
int DetachAtHead();
```

Description

Removes items from the head of a list without searching for a match.

TListImp::FirstThat

See Also TListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TListImp::Flush

TListImp class

Syntax

```
void Flush();
```

Description

Flushes the list without destroying it.

TListImp::ForEach

TListImp class

Syntax

```
void ForEach( IterFunc iter, void *args );
```

Description

Executes function iter for list element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TListImp::IsEmpty

TListImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the list has no elements; otherwise returns 0.

TListImp::LastThat

See Also   TListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

FirstThat

ForEach

TListImp::PeekHead

TListImp class

Syntax

```
Const T& PeekHead() const;
```

Description

Returns a reference to the Head item in the list, without removing it.

TListImp::Head

TListImp class

Syntax

```
TMListElement<T,Alloc> Head;
```

Description

The element before the first element in the list.

TListImp::Tail

TListImp class

Syntax

```
TMListElement<T,Alloc> Tail;
```

Description

The element after the last element in the list.

TListImp::FindDetach

TListImp class

Syntax

```
virtual TMListElement<T,Alloc> *FindDetach( const T& t )
```

Description

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

TListImp::FindPred

TListImp class

Syntax

```
virtual TMListElement<T,Alloc> *FindPred( const T& );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TListIteratorImp template

Syntax

```
template <class T> class TListIteratorImp;
```

Header File

listimp.h

Description

Implements a list iterator that works on direct, managed list.

Public Constructor

TListIteratorImp::TListIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

TListIteratorImp::TListIteratorImp

TListIteratorImp class

Syntax

```
TListIteratorImp( const TMListImp<T,TStandardAllocator> &l )
```

Description

Constructs an iterator that traverses TListImp objects.

TListIteratorImp::Current

TListIteratorImp class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TListIteratorImp::Restart

TListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TListIteratorImp::operator int

TListIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TListIteratorImp::operator ++

TListIteratorImp class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMSListImp template

Syntax

```
template <class T, class Alloc> class TMSListImp;
```

Header File

listimp.h

Description

Implements a managed, sorted list of objects of type T. TMSListImp assumes that T has meaningful copy semantics, a meaningful < operator, and a default constructor.

Type Definitions

CondFunc

IterFunc

Public Constructor

TMSListImp::TMSListImp

Public Member Functions

Add

Detach

DetachAtHead

FirstThat

Flush

ForEach

IsEmpty

LastThat

PeekHead

Protected Data Members

Head

Tail

Protected Member Functions

FindDetach

FindPred

TMSListImp::CondFunc

TMSListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMSListImp::IterFunc

TMSListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMSListImp::TMSListImp

TMSListImp class

Syntax

```
TMSListImp()
```

Description

Constructs an empty list.

TMSListImp::Add

TMSListImp class

Syntax

```
int Add( const T& t );
```

Description

Adds an object to the list.

TMSListImp::Detach

See Also TMSListImp class

Syntax

```
int Detach( const T& );
```

Description

Removes the given object from the list. Returns 0 for failure, 1 for success in removing the object.

See Also
TShouldDelete

TMSListImp::DetachAtHead

TMSListImp class

Syntax

```
int DetachAtHead();
```

Description

Removes items from the head of a list without searching for a match.

TMSListImp::FirstThat

See Also TMSListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TMSListImp::Flush

TMSListImp class

Syntax

```
void Flush();
```

Description

Flushes the list without destroying it.

TMSListImp::ForEach

TMSListImp class

Syntax

```
void ForEach( IterFunc iter, void *args );
```

Description

Executes function iter for list element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMSListImp::IsEmpty

TMSListImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the list has no elements; otherwise returns 0.

TMSListImp::LastThat

See Also TMSListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

FirstThat

ForEach

TMSListImp::PeekHead

TMSListImp class

Syntax

```
Const T& PeekHead() const;
```

Description

Returns a reference to the Head item in the list, without removing it.

TMSListImp::Head

TMSListImp class

Syntax

```
TMListElement<T,Alloc> Head;
```

Description

The element before the first element in the list.

TMSListImp::Tail

TMSListImp class

Syntax

```
TMListElement<T,Alloc> Tail;
```

Description

The element after the last element in the list.

TMSListImp::FindDetach

TMSListImp class

Syntax

```
virtual TMListElement<T,Alloc> *FindDetach( const T& t )
```

Description

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

TMSListImp::FindPred

TMSListImp class

Syntax

```
virtual TMListElement<T,Alloc> *FindPred( const T& );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TMSListIteratorImp template

Syntax

```
template <class T, class Alloc> class TMSListIteratorImp;
```

Header File

listimp.h

Description

Implements a list iterator that works on direct, managed, sorted list.

Public Constructor

TMSListIteratorImp::TMSListIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

TMSListIteratorImp::TMSListIteratorImp

TMSListIteratorImp class

Syntax

```
TMSListIteratorImp( const TMSListImp<T,Alloc> &l )
```

Description

Constructs an iterator that traverses TMSListImp objects.

TMSListIteratorImp::Current

TMSListIteratorImp class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TMSListIteratorImp::Restart

TMSListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TMSListIteratorImp::operator int

TMSListIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMSListIteratorImp::operator ++

TMSListIteratorImp class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TSListImp template

Syntax

```
template <class T> class TSListImp;
```

Header File

listimp.h

Description

Implements a sorted list of objects of type T, using TStandardAllocator for memory management. TSListImp assumes that T has meaningful copy semantics, a meaningful < operator, and a default constructor.

Type Definitions

CondFunc

IterFunc

Public Constructor

TSListImp::TSListImp

Public Member Functions

Add

Detach

DetachAtHead

FirstThat

Flush

ForEach

IsEmpty

LastThat

PeekHead

Protected Data Members

Head

Tail

Protected Member Functions

FindDetach

FindPred

TSListImp::CondFunc

TSListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TSListImp::IterFunc

TSListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TSListImp::TSListImp

TSListImp class

Syntax

```
TSListImp()
```

Description

Constructs an empty list.

TSListImp::Add

TSListImp class

Syntax

```
int Add( const T& t );
```

Description

Adds an object to the list.

TSListImp::Detach

See Also TSListImp class

Syntax

```
int Detach( const T& );
```

Description

Removes the given object from the list. Returns 0 for failure, 1 for success in removing the object.

See Also
TShouldDelete

TSListImp::DetachAtHead

TSListImp class

Syntax

```
int DetachAtHead();
```

Description

Removes items from the head of a list without searching for a match.

TSListImp::FirstThat

See Also   TSListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

LastThat

TSListImp::Flush

TSListImp class

Syntax

```
void Flush();
```

Description

Flushes the list without destroying it.

TSListImp::ForEach

TSListImp class

Syntax

```
void ForEach( IterFunc iter, void *args );
```

Description

Executes function iter for list element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TSListImp::IsEmpty

TSListImp class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the list has no elements; otherwise returns 0.

TSListImp::LastThat

See Also TSListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

FirstThat

ForEach

TSListImp::PeekHead

TSListImp class

Syntax

```
Const T& PeekHead() const;
```

Description

Returns a reference to the Head item in the list, without removing it.

TSListImp::Head

TSListImp class

Syntax

```
TMListElement<T,Alloc> Head;
```

Description

The element before the first element in the list.

TSListImp::Tail

TSListImp class

Syntax

```
TMListElement<T,Alloc> Tail;
```

Description

The element after the last element in the list.

TSListImp::FindDetach

TSListImp class

Syntax

```
virtual TMListElement<T,Alloc> *FindDetach( const T& t )
```

Description

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

TSListImp::FindPred

TSListImp class

Syntax

```
virtual TMListElement<T,Alloc> *FindPred( const T& );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TSListIteratorImp template

Syntax

```
template <class T> class TSListIteratorImp;
```

Header File

listimp.h

Description

Implements a list iterator that works on direct, sorted list.

Public Constructor

TSListIteratorImp::TSListIteratorImp

Public Member Functions

Current

Restart

Operators

int

++

TSListIteratorImp::TSListIteratorImp

TSListIteratorImp class

Syntax

```
TSListIteratorImp(const TSListImp<T,Alloc> &l)
```

Description

Constructs an iterator that traverses TSListImp objects.

TSListIteratorImp::Current

TSListIteratorImp class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TSListIteratorImp::Restart

TSListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TSListIteratorImp::operator int

TSListIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TSListIteratorImp::operator ++

TSListIteratorImp class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMIListImp template

Syntax

```
template <class T, class Alloc> class TMIListImp;
```

Header File

listimp.h

Description

Implements a managed list of pointers to objects of type T. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Type Definitions

CondFunc

IterFunc

Public Member Functions

Add

Detach

FirstThat

ForEach

LastThat

PeekHead

Protected Member Functions

FindPred

TMIListImp::CondFunc

TMIListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMIListImp::IterFunc

TMIListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMIListImp::Add

TMIListImp class

Syntax

```
int Add( T *t );
```

Description

Adds an object pointer to the list.

TMIListImp::Detach

See Also TMIListImp class

Syntax

```
int Detach( T *t, int del = 0 )
```

Description

Removes the given object pointer from the list. The second argument specifies whether the object should be deleted.

See Also
TShouldDelete

TMIListImp::FirstThat

See Also TMIListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TMIListImp::ForEach

TMIListImp class

Syntax

```
void ForEach( IterFunc iter, void * )
```

Description

Executes function iter for each list element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMIListImp::LastThat

See Also TMIListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

FirstThat

ForEach

TMIListImp::PeekHead

TMIListImp class

Syntax

```
T *PeekHead() const;
```

Description

Returns the object pointer at the Head of the list, without removing it.

TMIListImp::FindPred

TMIListImp class

Syntax

```
virtual TMListElement<VoidPointer,Alloc> *FindPred( VoidPointer );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TMIListIteratorImp template

Syntax

```
template <class T, class Alloc> class TMIListIteratorImp;
```

Header File

listimp.h

Description

Implements a list iterator that works with any managed indirect list. For direct lists, see TMListIteratorImp.

Public Constructor

TMIListIteratorImp::TMIListIteratorImp

Public Member Functions

Current

Restart

Operators

++

TMIListIteratorImp::TMIListIteratorImp

TMIListIteratorImp class

Syntax

```
TMIListIteratorImp( const TMIListImp<VoidPointer,Alloc> &l )
```

Description

Constructs an object that iterates on TMIListImp objects.

TMIListIteratorImp::Current

TMIListIteratorImp class

Syntax

```
T *Current()
```

Description

Returns the current object pointer.

TMIListIteratorImp::Restart

TMIListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TMIListIteratorImp::operator ++

TMIListIteratorImp class

Form 1

```
T *operator ++ (int)
```

Form 2

```
T *operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TIListImp template

Syntax

```
template <class T> class TIListImp;
```

Header File

listimp.h

Description

Implements a list of pointers to objects of type T. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Type Definitions

CondFunc

IterFunc

Public Member Functions

Add

Detach

FirstThat

ForEach

LastThat

PeekHead

Protected Member Functions

FindPred

TIListImp::CondFunc

TIListImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TIListImp::IterFunc

TIListImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TIListImp::Add

TIListImp class

Syntax

```
int Add( T *t );
```

Description

Adds an object pointer to the list.

TIListImp::Detach

See Also TIListImp class

Syntax

```
int Detach( T *t, int del = 0 )
```

Description

Removes the given object pointer from the list. The second argument specifies whether the object should be deleted.

See Also
TShouldDelete

TIListImp::FirstThat

See Also TIListImp class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the list that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TIListImp::ForEach

TIListImp class

Syntax

```
void ForEach( IterFunc iter, void * )
```

Description

Executes function iter for each list element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TIListImp::LastThat

See Also   TIListImp class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the list that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also

FirstThat

ForEach

TIListImp::PeekHead

TIListImp class

Syntax

```
T *PeekHead() const;
```

Description

Returns the object pointer at the Head of the list, without removing it.

TIListImp::FindPred

TIListImp class

Syntax

```
virtual TMListElement<VoidPointer,Alloc> *FindPred( VoidPointer );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TIListIteratorImp template

Syntax

```
template <class T> class TIListIteratorImp;
```

Header File

listimp.h

Description

Implements a list iterator that works with any indirect list.

Public Constructor

TIListIteratorImp::TIListIteratorImp

Public Member Functions

Current

Restart

Operators

++

TIListIteratorImp::TIListIteratorImp

TIListIteratorImp class

Syntax

```
TIListIteratorImp( const TIListImp<T> &l )
```

Description

Constructs an object that iterates on TIListImp objects.

TIListIteratorImp::Current

TIListIteratorImp class

Syntax

```
T *Current()
```

Description

Returns the current object pointer.

TIListIteratorImp::Restart

TIListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TIListIteratorImp::operator ++

TIListIteratorImp class

Form 1

```
T *operator ++ (int)
```

Form 2

```
T *operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMISListImp template

Syntax

```
template <class T, class Alloc> class TMISListImp;
```

Header File

listimp.h

Description

Implements a managed sorted list of pointers to objects of type T. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Public Member Functions

FindDetach

FindPred

TMISListImp::FindDetach

TMISListImp class

Syntax

```
virtual TMListElement<TVoidPointer,Alloc> *FindDetach(TVoidPointer);
```

Description

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

TMISListImp::FindPred

TMISListImp class

Syntax

```
virtual TMListElement<TVoidPointer,Alloc> *FindPred( TVoidPointer );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TMISListIteratorImp template

Syntax

```
template <class T, class Alloc> class TMISListIteratorImp;
```

Header File

listimp.h

Description

Implements a list iterator that works with any managed indirect list. For direct lists, see TMListIteratorImp.

Public Constructor

TMISListIteratorImp::TMISListIteratorImp

TMISListIteratorImp::TMISListIteratorImp

TMISListIteratorImp class

Syntax

```
TMISListIteratorImp( const TMISListImp<T,Alloc> &l ) :
```

Description

Constructs an object that iterates on TMISListImp objects.

TISListImp template

Syntax

```
template <class T> class TISListImp;
```

Header File

listimp.h

Description

Implements a sorted list of pointers to objects of type T, using TStandardAllocator for memory management. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Public Member Functions

FindDetach

FindPred

TISListImp::FindDetach

TISListImp class

Syntax

```
virtual TMListElement<TVoidPointer,Alloc> *FindDetach(TVoidPointer);
```

Description

Determines whether an object is in the list, and returns a pointer to its predecessor. Returns 0 if not found.

TISListImp::FindPred

TISListImp class

Syntax

```
virtual TMListElement<TVoidPointer,Alloc> *FindPred( TVoidPointer );
```

Description

Finds the element that would be followed by the parameter. The function does not check whether the parameter is actually there. This can be used for inserting (insert after returned element pointer).

TISListIteratorImp template

Syntax

```
template <class T> class TISListIteratorImp;
```

Header File

listimp.h

Description

Implements a list iterator that works with any indirect list.

Public Constructor

TISListIteratorImp::TISListIteratorImp

Public Member Functions

Current

Restart

Operators

++

TISListIteratorImp::TISListIteratorImp

TISListIteratorImp class

Syntax

```
TISListIteratorImp( const TISListImp<T> &l )
```

Description

Constructs an object that iterates on TISListImp objects.

TISListIteratorImp::Current

TISListIteratorImp class

Syntax

```
T *Current()
```

Description

Returns the current object pointer.

TISListIteratorImp::Restart

TISListIteratorImp class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TISListIteratorImp::operator ++

TISListIteratorImp class

Form 1

```
T *operator ++ (int)
```

Form 2

```
T *operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMQueueAsVector template

Header File
queues.h

Description
Implements a managed queue of objects of type T, using a vector as the underlying implementation. TMQueueAsVector assumes T has meaningful copy semantics, a < operator, and a default constructor. The memory manager Alloc provides class-specific new and delete operators.

Public Constructors
TMQueueAsVector::TMQueueAsVector

Public Member Functions
FirstThat

Flush

ForEach

Get

GetItemsInContainer

IsEmpty

IsFull

LastThat

Put

TMQueueAsVector::TMQueueAsVector

TMQueueAsVector class

Syntax

```
TMQueueAsVector( unsigned sz = DEFAULT_QUEUE_SIZE )
```

Description

Constructs a managed, vector-implemented queue, of sz size.

TMQueueAsVector::FirstThat

See Also TMQueueAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the queue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TMQueueAsVector::Flush

See Also TMQueueAsVector class

Syntax

```
void Flush()
```

Description

Flushes the queue without destroying it. The fate of any objects removed depends on the current ownership status.

See Also
TShouldDelete::ownsElements

TMQueueAsVector::ForEach

TMQueueAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args );
```

Description

Executes function iter for each queue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMQueueAsVector::Get

TMQueueAsVector class

Syntax

```
T Get()
```

Description

Removes the object from the head of the queue. If the queue is empty, it returns 0. Otherwise the removed object is returned.

TMQueueAsVector::GetItemsInContainer

TMQueueAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the queue.

TMQueueAsVector::IsEmpty

TMQueueAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the queue has no elements; otherwise returns 0.

TMQueueAsVector::IsFull

TMQueueAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the queue is full; otherwise returns 0.

TMQueueAsVector::LastThat

See Also   TMQueueAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the queue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the queue meets the condition.

See Also

FirstThat

ForEach

TMQueueAsVector::Put

TMQueueAsVector class

Syntax

```
void Put( T t )
```

Description

Adds an object to (the tail of) a queue.

TMQueueAsVectorIterator template

Header File

queues.h

Description

Implements an iterator object for managed, vector-based queues.

Public Constructors

TMQueueAsVectorIterator::TMQueueAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TMQueueAsVectorIterator::TMQueueAsVectorIterator

TMQueueAsVectorIterator class

Syntax

```
TMQueueAsVectorIterator( const TMQueueAsVector<T,Alloc> &q )
```

Description

Constructs an object that iterates on TMQueueAsVector objects.

TMQueueAsVectorIterator::Current

TMQueueAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TMQueueAsVectorIterator::Restart

TMQueueAsVectorIterator class

Syntax

```
void Restart();
```

Description

Restarts iteration.

TMQueueAsVectorIterator::operator ++

TMQueueAsVectorIterator class

Form 1

```
const T& operator ++ ( int );
```

Form 2

```
const T& operator ++ ();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMQueueAsVectorIterator::operator int

TMQueueAsVectorIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. Iterator converts to 0 if nothing remains in the iterator.

TQueueAsVector template

Header File

queues.h

Description

Implements an iterator object for vector-based queues.

Public Constructors

TQueueAsVector::TQueueAsVector

Public Member Functions

FirstThat

Flush

ForEach

Get

GetItemsInContainer

IsEmpty

IsFull

LastThat

Put

TQueueAsVector::TQueueAsVector

TQueueAsVector class

Syntax

```
TQueueAsVector( unsigned sz = DEFAULT_QUEUE_SIZE )
```

Description

Constructs a vector-implemented queue, of sz size.

TQueueAsVector::FirstThat

See Also TQueueAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the queue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TQueueAsVector::Flush

See Also TQueueAsVector class

Syntax

```
void Flush()
```

Description

Flushes the queue without destroying it. The fate of any objects removed depends on the current ownership status.

See Also
TShouldDelete::ownsElements

TQueueAsVector::ForEach

TQueueAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args );
```

Description

Executes function iter for each queue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TQueueAsVector::Get

TQueueAsVector class

Syntax

```
T Get()
```

Description

Removes the object from the end (tail) of the queue. If the queue is empty, it returns 0. Otherwise the removed object is returned.

TQueueAsVector::GetItemsInContainer

TQueueAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the queue.

TQueueAsVector::IsEmpty

TQueueAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the queue has no elements; otherwise returns 0.

TQueueAsVector::IsFull

TQueueAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the queue is full; otherwise returns 0.

TQueueAsVector::LastThat

See Also TQueueAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the queue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the queue meets the condition.

See Also

FirstThat

ForEach

TQueueAsVector::Put

TQueueAsVector class

Syntax

```
void Put( T t )
```

Description

Adds an object to (the tail of) a queue.

TQueueAsVectorIterator template

Header File
queues.h

Description
Implements an iterator object for vector-based queues.

Public Constructors
TQueueAsVectorIterator::TQueueAsVectorIterator

Public Member Functions
Current

Restart

Operators
++

int

TQueueAsVectorIterator::TQueueAsVectorIterator

TQueueAsVectorIterator class

Syntax

```
TQueueAsVectorIterator( const TQueueAsVector<T> &q )
```

Description

Constructs an object that iterates on TQueueAsVector objects.

TQueueAsVectorIterator::Current

TQueueAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TQueueAsVectorIterator::Restart

TQueueAsVectorIterator class

Syntax

```
void Restart();
```

Description

Restarts iteration.

TQueueAsVectorIterator::operator ++

TQueueAsVectorIterator class

Form 1

```
const T& operator ++ ( int );
```

Form 2

```
const T& operator ++ ();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TQueueAsVectorIterator::operator int

TQueueAsVectorIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. Iterator converts to 0 if nothing remains in the iterator.

TMIQueueAsVector template

Header File
queues.h

Description
Implements a managed queue of pointers to objects of type T, using a vector as the underlying implementation.

Public Constructors
TMIQueueAsVector::TMIQueueAsVector

Public Member Functions
FirstThat
Flush
ForEach
Get
GetItemsInContainer
IsEmpty
isFull
LastThat
Put

TMIQueueAsVector::TMIQueueAsVector

TMIQueueAsVector class

Syntax

```
TMIQueueAsVector( unsigned sz = DEFAULT_QUEUE_SIZE )
```

Description

Constructs a managed, indirect queue, of sz size.

TMIQueueAsVector::FirstThat

See Also   TMIQueueAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the queue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TMIQueueAsVector::Flush

TMIQueueAsVector class

Syntax

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete );
```

Description

Flushes the queue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the dt argument.

TMIQueueAsVector::ForEach

TMIQueueAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args );
```

Description

Executes function iter for each queue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMIQueueAsVector::Get

TMIQueueAsVector class

Syntax

```
T *Get()
```

Description

Removes and returns the object pointer from the queue. If the queue is empty, it returns 0.

TMIQueueAsVector::GetItemsInContainer

TMIQueueAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the queue.

TMIQueueAsVector::IsEmpty

TMIQueueAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if a queue has no elements; otherwise returns 0.

TMIQueueAsVector::IsFull

TMIQueueAsVector class

Syntax

```
int isFull() const;
```

Description

Returns 1 if a queue is full; otherwise returns 0.

TMIQueueAsVector::LastThat

See Also TMIQueueAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the queue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the queue meets the condition.

See Also

FirstThat

ForEach

TMIQueueAsVector::Put

TMIQueueAsVector class

Syntax

```
void Put( T *t )
```

Description

Adds an object pointer to (the tail of) a queue.

TMIQueueAsVectorIterator template

Header File
queues.h

Description
Implements an iterator object for managed, indirect, vector-based queues.

Public Constructors
TMIQueueAsVectorIterator::TMIQueueAsVectorIterator

TMIQueueAsVectorIterator::TMIQueueAsVectorIterator

TMIQueueAsVectorIterator class

Syntax

```
TMIQueueAsVectorIterator( const TMIDequeAsVector<T,Alloc> &q )
```

Description

Constructs an object that iterates on TMIQueueAsVector objects.

TIQueueAsVector template

Header File
queues.h

Description
Implements a queue of pointers to objects of type T, using a vector as the underlying implementation.

Public Constructors
TIQueueAsVector::TIQueueAsVector

TIQueueAsVector::TIQueueAsVector

TIQueueAsVector class

Syntax

```
TIQueueAsVector( unsigned sz = DEFAULT_QUEUE_SIZE )
```

Description

Constructs an indirect queue, of sz size.

TIQueueAsVectorIterator template

Header File
queues.h

Description
Implements an iterator object for indirect, vector-based queues.

Public Constructors
TIQueueAsVectorIterator::TIQueueAsVectorIterator

Public Member Functions
Current

Restart

Operators
++

int

TIQueueAsVectorIterator::TIQueueAsVectorIterator

TIQueueAsVectorIterator class

Syntax

```
TIQueueAsVectorIterator( const TIQueueAsVector<T> &q )
```

Description

Constructs an object that iterates on TIQueueAsVector objects.

TIQueueAsVectorIterator::Current

TIQueueAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TIQueueAsVectorIterator::Restart

TIQueueAsVectorIterator class

Syntax

```
void Restart();
```

Description

Restarts iteration.

TIQueueAsVectorIterator::operator ++

TIQueueAsVectorIterator class

Form 1

```
const T& operator ++ ( int );
```

Form 2

```
const T& operator ++ ();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TIQueueAsVectorIterator::operator int

TIQueueAsVectorIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. Iterator converts to 0 if nothing remains in the iterator.

TMQueueAsDoubleList template

Header File
queues.h

Description
Implements a managed queue of objects of type T, using a double-linked list as the underlying implementation.

Public Member Functions
FirstThat
Flush
ForEach
Get
GetItemsInContainer
IsEmpty
IsFull
LastThat
Put

TMQueueAsDoubleList::FirstThat

See Also   TMQueueAsDoubleList class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the queue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TMQueueAsDoubleList::Flush

TMQueueAsDoubleList class

Syntax

```
void Flush()
```

Description

Flushes the queue without destroying it. The fate of any objects removed depends on the current ownership status.

TMQueueAsDoubleList::ForEach

TMQueueAsDoubleList class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each queue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMQueueAsDoubleList::Get

TMQueueAsDoubleList class

Syntax

```
T Get()
```

Description

Removes the object from the end (tail) of the queue. If the queue is empty, it returns 0. Otherwise the removed object is returned.

TMQueueAsDoubleList::GetItemsInContainer

TMQueueAsDoubleList class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the queue.

TMQueueAsDoubleList::IsEmpty

TMQueueAsDoubleList class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if a queue has no elements; otherwise returns 0.

TMQueueAsDoubleList::IsFull

TMQueueAsDoubleList class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if a queue is full; otherwise returns 0.

TMQueueAsDoubleList::LastThat

See Also TMQueueAsDoubleList class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the queue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition. Note that LastThat creates its own internal iterator, so you can treat it as a "search" function.

See Also

FirstThat

ForEach

TMQueueAsDoubleList::Put

TMQueueAsDoubleList class

Syntax

```
void Put( T t )
```

Description

Adds an object to (the tail of) a queue.

TMQueueAsDoubleListIterator template

Header File

queues.h

Description

Implements an iterator object for list-based queues.

Public Constructors

TMQueueAsDoubleListIterator::TMQueueAsDoubleListIterator

Public Member Functions

Current

Restart

Operators

int

++

--

TMQueueAsDoubleListIterator::TMQueueAsDoubleListIterator

TMQueueAsDoubleListIterator class

Syntax

```
TMQueueAsDoubleListIterator( const TMQueueAsDoubleList<T,Alloc> & q )
```

Description

Constructs an object that iterates on TMQueueAsDoubleList objects.

TMQueueAsDoubleListIterator::Current

TMQueueAsDoubleListIterator class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TMQueueAsDoubleListIterator::Restart

TMQueueAsDoubleListIterator class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TMQueueAsDoubleListIterator::operator int

TMQueueAsDoubleListIterator class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMQueueAsDoubleListIterator::operator ++

TMQueueAsDoubleListIterator class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMQueueAsDoubleListIterator::operator - -

TMQueueAsDoubleListIterator class

Form 1

```
const T& operator -- ( int )
```

Form 2

```
const T& operator -- ()
```

Description

Form 1: Moves to the previous object, and returns the object that was current before the move (post-decrement).

Form 2: Moves to the previous object, and returns the object that was current after the move (pre-decrement).

TQueueAsDoubleList template

Header File
queues.h

Description
Implements a queue of objects of type T, using a double-linked list as the underlying implementation.

Public Member Functions
FirstThat
Flush
ForEach
Get
GetItemsInContainer
IsEmpty
IsFull
LastThat
Put

TQueueAsDoubleList::FirstThat

See Also   TQueueAsDoubleList class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the queue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TQueueAsDoubleList::Flush

TQueueAsDoubleList class

Syntax

```
void Flush()
```

Description

Flushes the queue without destroying it. The fate of any objects removed depends on the current ownership status.

TQueueAsDoubleList::ForEach

TQueueAsDoubleList class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each queue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TQueueAsDoubleList::Get

TQueueAsDoubleList class

Syntax

```
T Get()
```

Description

Removes the object from the end (tail) of the queue. If the queue is empty, it returns 0. Otherwise the removed object is returned.

TQueueAsDoubleList::GetItemsInContainer

TQueueAsDoubleList class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the queue.

TQueueAsDoubleList::IsEmpty

TQueueAsDoubleList class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if a queue has no elements; otherwise returns 0.

TQueueAsDoubleList::IsFull

TQueueAsDoubleList class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if a queue is full; otherwise returns 0.

TQueueAsDoubleList::LastThat

See Also TQueueAsDoubleList class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the queue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition. Note that LastThat creates its own internal iterator, so you can treat it as a "search" function.

See Also

FirstThat

ForEach

TQueueAsDoubleList::Put

TQueueAsDoubleList class

Syntax

```
void Put( T t )
```

Description

Adds an object to (the tail of) a queue.

TQueueAsDoubleListIterator template

Header File

queues.h

Description

Implements an iterator object for list-based queues.

Public Constructors

TQueueAsDoubleListIterator::TQueueAsDoubleListIterator

Public Member Functions

Current

Restart

Operators

int

++

--

TQueueAsDoubleListIterator::TQueueAsDoubleListIterator

TQueueAsDoubleListIterator class

Syntax

```
TQueueAsDoubleListIterator( const TQueueAsDoubleList<T> &q )
```

Description

Constructs an object that iterates on TQueueAsDoubleList objects.

TQueueAsDoubleListIterator::Current

TQueueAsDoubleListIterator class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TQueueAsDoubleListIterator::Restart

TQueueAsDoubleListIterator class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TQueueAsDoubleListIterator::operator int

TQueueAsDoubleListIterator class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TQueueAsDoubleListIterator::operator ++

TQueueAsDoubleListIterator class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TQueueAsDoubleListIterator::operator - -

TQueueAsDoubleListIterator class

Form 1

```
const T& operator -- ( int )
```

Form 2

```
const T& operator -- ()
```

Description

Form 1: Moves to the previous object, and returns the object that was current before the move (post-decrement).

Form 2: Moves to the previous object, and returns the object that was current after the move (pre-decrement).

TMIQueueAsDoubleList template

Header File

queues.h

Description

Implements a managed indirect queue of pointers to objects of type T, using a double-linked list as the underlying implementation.

Public Member Functions

FirstThat

Flush

ForEach

Get

GetItemsInContainer

IsEmpty

IsFull

LastThat

Put

TMIQueueAsDoubleList::FirstThat

See Also TMIQueueAsDoubleList class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the queue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TMIQueueAsDoubleList::Flush

TMIQueueAsDoubleList class

Syntax

```
void Flush( TShouldDelete::DeleteType dt = TShouldDelete::DefDelete )
```

Description

Flushes the queue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the dt argument.

TMIQueueAsDoubleList::ForEach

TMIQueueAsDoubleList class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each queue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TMIQueueAsDoubleList::Get

TMIQueueAsDoubleList class

Syntax

```
T *Get()
```

Description

Removes and returns the object pointer from the queue. If the queue is empty, it throws the PRECONDITION exception in the debug version. In the non-debug version, Get returns a meaningless object is the queue is empty.

TMIQueueAsDoubleList::GetItemsInContainer

TMIQueueAsDoubleList class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the queue.

TMIQueueAsDoubleList::IsEmpty

TMIQueueAsDoubleList class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the queue has no elements; otherwise returns 0.

TMIQueueAsDoubleList::IsFull

TMIQueueAsDoubleList class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the queue is full; otherwise returns 0.

TMIQueueAsDoubleList::LastThat

See Also TMIQueueAsDoubleList class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the queue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the queue meets the condition.

See Also

FirstThat

ForEach

TMIQueueAsDoubleList::Put

TMIQueueAsDoubleList class

Syntax

```
void Put( T *t )
```

Description

Adds an object pointer to (the tail of) a queue. If the queue is full, it throws the PRECONDITION exception in the debug version. If the queue is full, the behavior of the non-debug version of Put is undefined.

TMIQueueAsDoubleListIterator template

Header File

queues.h

Description

Implements an iterator object for indirect, list-based queues.

Public Constructors

TMIQueueAsDoubleListIterator::TMIQueueAsDoubleListIterator

Public Member Functions

Current

Restart

Operators

int

++

--

TMIQueueAsDoubleListIterator::TMIQueueAsDoubleListIterator

TMIQueueAsDoubleListIterator class

Syntax

```
TMIQueueAsDoubleListIterator( const TMIQueueAsDoubleList<T,Alloc> & q )
```

Description

Constructs an object that iterates on TMIQueueAsDoubleList objects.

TMIQueueAsDoubleListIterator::Current

TMIQueueAsDoubleListIterator class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TMIQueueAsDoubleListIterator::Restart

TMIQueueAsDoubleListIterator class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TMIQueueAsDoubleListIterator::operator int

TMIQueueAsDoubleListIterator class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMIQueueAsDoubleListIterator::operator ++

TMIQueueAsDoubleListIterator class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMIQueueAsDoubleListIterator::operator - -

TMIQueueAsDoubleListIterator class

Form 1

```
const T& operator -- ( int )
```

Form 2

```
const T& operator -- ()
```

Description

Form 1: Moves to the previous object, and returns the object that was current before the move (post-decrement).

Form 2: Moves to the previous object, and returns the object that was current after the move (pre-decrement).

TIQueueAsDoubleList template

Header File
queues.h

Description
Implements an indirect queue of pointers to objects of type T, using a double-linked list as the underlying implementation.

Public Member Functions
FirstThat
Flush
ForEach
Get
GetItemsInContainer
IsEmpty
IsFull
LastThat
Put

TIQueueAsDoubleList::FirstThat

See Also TIQueueAsDoubleList class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the queue that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition.

See Also
LastThat

TIQueueAsDoubleList::Flush

TIQueueAsDoubleList class

Syntax

```
void Flush( TShouldDelete::DeleteType dt = TShouldDelete::DefDelete )
```

Description

Flushes the queue without destroying it. The fate of any objects removed depends on the current ownership status and the value of the dt argument.

TIQueueAsDoubleList::ForEach

TIQueueAsDoubleList class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each queue element. ForEach creates an internal iterator to execute the given function for each element in the array. The args argument lets you pass arbitrary data to this function.

TIQueueAsDoubleList::Get

TIQueueAsDoubleList class

Syntax

```
T *Get()
```

Description

Removes and returns the object pointer from the queue. If the queue is empty, it returns 0. If the queue is empty, it throws the PRECONDITION exception in the debug version. In the non-debug version, Get returns a meaningless object is the queue is empty.

TIQueueAsDoubleList::GetItemsInContainer

TIQueueAsDoubleList class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the queue.

TIQueueAsDoubleList::IsEmpty

TIQueueAsDoubleList class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the queue has no elements; otherwise returns 0.

TIQueueAsDoubleList::IsFull

TIQueueAsDoubleList class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the queue is full; otherwise returns 0.

TIQueueAsDoubleList::LastThat

See Also   TIQueueAsDoubleList class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the queue that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the array meets the condition. Note that LastThat creates its own internal iterator, so you can treat it as a "search" function.

See Also

FirstThat

ForEach

TIQueueAsDoubleList::Put

TIQueueAsDoubleList class

Syntax

```
void Put( T *t )
```

Description

Adds an object pointer to (the tail of) a queue. If the queue is full, it throws the PRECONDITION exception in the debug version. If the queue is full, the behavior of the non-debug version of Put is undefined.

TIQueueAsDoubleListIterator template

Header File

queues.h

Description

Implements an iterator object for indirect, list-based queues.

Public Constructors

TIQueueAsDoubleListIterator::TIQueueAsDoubleListIterator

Public Member Functions

Current

Restart

Operators

int

++

--

TIQueueAsDoubleListIterator::TIQueueAsDoubleListIterator

TIQueueAsDoubleListIterator class

Syntax

```
TIQueueAsDoubleListIterator( const TIQueueAsDoubleList<T> & q )
```

Description

Constructs an object that iterates on TIQueueAsDoubleList objects.

TIQueueAsDoubleListIterator::Current

TIQueueAsDoubleListIterator class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TIQueueAsDoubleListIterator::Restart

TIQueueAsDoubleListIterator class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TIQueueAsDoubleListIterator::operator int

TIQueueAsDoubleListIterator class

Syntax

```
operator int()
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TIQueueAsDoubleListIterator::operator ++

TIQueueAsDoubleListIterator class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TIQueueAsDoubleListIterator::operator - -

TIQueueAsDoubleListIterator class

Form 1

```
const T& operator -- ( int )
```

Form 2

```
const T& operator -- ()
```

Description

Form 1: Moves to the previous object, and returns the object that was current before the move (post-decrement).

Form 2: Moves to the previous object, and returns the object that was current after the move (pre-decrement).

TQueue template

Header File
queues.h

Description
A simplified name for TQueueAsVector.

TQueueIterator template

Header File
queues.h

Description
A simplified name for TQueueAsVectorIterator.

TMSetAsVector template

Syntax

```
template <class T, class Alloc> class TMSetAsVector;
```

Header File

sets.h

Description

Implements a managed set of objects of type T, using a vector as the underlying implementation. A set, unlike a bag, cannot contain duplicate items.

TMSetAsVector inherits member functions from TMBagAsVector.

Type Definitions

CondFunc

IterFunc

Public Constructors

TMSetAsVector::TMSetAsVector

Public Member Functions

Add

Detach

Find

FindMember

Flush

ForEach

GetItemsInContainer

HasMember

isEmpty

isFull

TMSetAsVector::TMSetAsVector

TMSetAsVector class

Syntax

```
TMSetAsVector( unsigned sz = DEFAULT_SET_SIZE )
```

Description

Constructs a managed, empty set. sz represents the number of items the set can hold.

TMSetAsVector::Add

TMSetAsVector class

Syntax

```
int Add( const T& t );
```

Description

Adds an object to the set.

TMSetAsVector::CondFunc

TMSetAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMSetAsVector::IterFunc

TMSetAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMSetAsVector::Detach

See Also TMSetAsVector class

Syntax

```
int Detach( const T& t )
```

Description

Removes the specified object. The value of dt and the current ownership setting determine whether the object itself will be deleted. DeleteType is defined in the base class TShouldDelete as enum { NoDelete, DefDelete, Delete }. The default value of dt, NoDelete, means that the object will not be deleted regardless of ownership. With dt set to Delete, the object will be deleted regardless of ownership. If dt is set to DefDelete, the object will be deleted only if the set owns its elements.

See Also
TShouldDelete::ownsElements

TMSetAsVector::Find

TMSetAsVector class

Syntax

```
virtual T *Find( const T& ) const;
```

Description

Returns a pointer to the given object if found; otherwise returns 0.

TMSetAsVector::FindMember

TMSetAsVector class

Syntax

```
T* FindMember( const T& t ) const;
```

Description

Returns a pointer to the given object if found; otherwise returns 0.

TMSetAsVector::Flush

See Also   TMSetAsVector class

Syntax

```
void Flush()
```

Description

Removes all the elements from the set without destroying the set. The value of dt determines whether the elements themselves are destroyed. By default, the ownership status of the set determines their fate, as explained in the Detach member function. You can also set dt to Delete and NoDelete.

See Also
Detach

TMSetAsVector::ForEach

TMSetAsVector class

Syntax

```
void ForEach(IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the set. The args argument lets you pass arbitrary data to this function.

TMSetAsVector::GetItemsInContainer

TMSetAsVector class

Syntax

```
int GetItemsInContainer() const
```

Description

Returns the number of objects in the set.

TMSetAsVector::HasMember

TMSetAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found; otherwise returns 0.

TMSetAsVector::IsEmpty

TMSetAsVector class

Syntax

```
int isEmpty() const;
```

Description

Returns 1 if the set is empty; otherwise returns 0.

TMSetAsVector::IsFull

TMSetAsVector class

Syntax

```
int isFull() const;
```

Description

Returns 0.

TMSetAsVectorIterator template

Syntax

```
template <class T, class Alloc> class TMSetAsVectorIterator;
```

Header File

sets.h

Description

Implements an iterator object to traverse TMSetAsVector objects.

Public Constructors

TMSetAsVectorIterator::TMSetAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TMSetAsVectorIterator::TMSetAsVectorIterator

Syntax

```
TMSetAsVectorIterator( const TMSetAsVector<T,Alloc> &s ) :
```

Description

Constructs an object that iterates on TMSetAsVector objects.

TMSetAsVectorIterator::Current

TMSetAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TMSetAsVectorIterator::Restart

TMSetAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Restarts iteration from the beginning, or over the specified range.

TMSetAsVectorIterator::operator ++

TMSetAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMSetAsVectorIterator::operator int

TMSetAsVectorIterator class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TSetAsVector template

Syntax

```
template <class T> class TSetAsVector;
```

Header File

sets.h

Description

Implements a set of objects of type T, using a vector as the underlying implementation. TStandardAllocator is used to manage memory.

Type Definitions

CondFunc

IterFunc

Public Constructors

TSetAsVector::TSetAsVector

Public Member Functions

Add

Detach

Find

FindMember

Flush

ForEach

GetItemsInContainer

HasMember

isEmpty

isFull

TSetAsVector::CondFunc

TSetAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TSetAsVector::IterFunc

TSetAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TSetAsVector::TSetAsVector

Syntax

```
TSetAsVector( unsigned sz = DEFAULT_SET_SIZE ) :
```

Description

Constructs an empty set. sz represents the number of items the set can hold.

TSetAsVector::Add

TSetAsVector class

Syntax

```
int Add( const T& t )
```

Description

Adds the given object to the set.

TSetAsVector::Detach

See Also   TSetAsVector class

Syntax

```
int Detach( const T& t )
```

Description

Removes the specified object. The value of dt and the current ownership setting determine whether the object itself will be deleted. DeleteType is defined in the base class TShouldDelete as enum { NoDelete, DefDelete, Delete }. The default value of dt, NoDelete, means that the object will not be deleted regardless of ownership. With dt set to Delete, the object will be deleted regardless of ownership. If dt is set to DefDelete, the object will be deleted only if the set owns its elements.

See Also
TShouldDelete::ownsElements

TSetAsVector::Find

TSetAsVector class

Syntax

```
virtual T *Find( const T& ) const;
```

Description

Returns a pointer to the given object if found; otherwise returns 0.

TSetAsVector::FindMember

TSetAsVector class

Syntax

```
T* FindMember( const T& t ) const;
```

Description

Returns a pointer to the given object if found; otherwise returns 0.

TSetAsVector::Flush

See Also TSetAsVector class

Syntax

```
void Flush()
```

Description

Removes all the elements from the set without destroying the set. The value of dt determines whether the elements themselves are destroyed. By default, the ownership status of the set determines their fate, as explained in the Detach member function. You can also set dt to Delete and NoDelete.

See Also
Detach

TSetAsVector::ForEach

TSetAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the set. The args argument lets you pass arbitrary data to this function.

TSetAsVector::GetItemsInContainer

TSetAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of objects in the set.

TSetAsVector::HasMember

TSetAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found; otherwise returns 0.

TSetAsVector::IsEmpty

TSetAsVector class

Syntax

```
int isEmpty() const;
```

Description

Returns 1 if the set is empty; otherwise returns 0.

TSetAsVector::IsFull

TSetAsVector class

Syntax

```
int isFull() const;
```

Description

Returns 0.

TSetAsVectorIterator template

Syntax

```
template <class T> class TSetAsVectorIterator;
```

Header File

sets.h

Description

Implements an iterator object to traverse TSetAsVector objects.

Public Constructors

TSetAsVectorIterator::TSetAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TSetAsVectorIterator::TSetAsVectorIterator

Syntax

```
TSetAsVectorIterator( const TSetAsVector<T> &s )
```

Description

Constructs an object that iterates on TMSetAsVector objects.

TSetAsVectorIterator::Current

TSetAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TSetAsVectorIterator::Restart

TSetAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Restarts iteration from the beginning, or over the specified range.

TSetAsVectorIterator::operator ++

TSetAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TSetAsVectorIterator::operator int

TSetAsVectorIterator class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMISetAsVector template

Syntax

```
template <class T, class Alloc> class TMISetAsVector;
```

Header File

sets.h

Description

Implements a managed set of pointers to objects of type T, using a vector as the underlying implementation.

TMISetAsVector inherits member functions from TMIBagAsVector.

Type Definitions

CondFunc

IterFunc

Public Constructors

TMISetAsVector::TMISetAsVector

Public Member Functions

Add

Detach

Find

FirstThat

Flush

ForEach

GetItemsInContainer

HasMember

isEmpty

isFull

LastThat

TMISetAsVector::CondFunc

TMISetAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMISetAsVector::IterFunc

TMISetAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMISetAsVector::TMISetAsVector

Syntax

```
TMISetAsVector( unsigned sz = DEFAULT_SET_SIZE ) :
```

Description

Constructs an empty, managed, indirect set. sz represents the initial number of slots allocated.

TMISetAsVector::Add

Syntax

```
int Add( T * );
```

Description

Adds an object pointer to the set.

TMISetAsVector::Detach

See Also TMISetAsVector class

Syntax

```
int Detach( T *t, DeleteType dt = NoDelete )
```

Description

Removes the specified object pointer. The value of dt and the current ownership setting determine whether the object itself will be deleted. DeleteType is defined in the base class TShouldDelete as enum { NoDelete, DefDelete, Delete }. The default value of dt, NoDelete, means that the object will not be deleted regardless of ownership. With dt set to Delete, the object will be deleted regardless of ownership. If dt is set to DefDelete, the object will only be deleted if the set owns its elements.

See Also
TShouldDelete::ownsElements

TMISetAsVector::Find

TMISetAsVector class

Syntax

```
T *Find( T *t ) const;
```

Description

Returns a pointer to the object if found; otherwise returns 0.

TMISetAsVector::FirstThat

TMISetAsVector class

Syntax

```
T *FirstThat(CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the set that satisfies a given condition. You supply a test-function pointer f that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the set meets the condition.

TMISetAsVector::Flush

See Also   TMISetAsVector class

Syntax

```
void Flush( TShouldDelete::DeleteType dt = TShouldDelete::DefDelete )
```

Description

Removes all the elements from the set without destroying the set. The value of dt determines whether the elements themselves are destroyed. By default, the ownership status of the set determines their fate, as explained in the Detach member function. You can also set dt to Delete and NoDelete.

See Also
Detach

TMISetAsVector::ForEach

TMISetAsVector class

Syntax

```
void ForEach(IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the set. The args argument lets you pass arbitrary data to this function.

TMISetAsVector::GetItemsInContainer

TMISetAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of objects in the set.

TMISetAsVector::HasMember

TMISetAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found; otherwise returns 0.

TMISetAsVector::IsEmpty

TMISetAsVector class

Syntax

```
int isEmpty() const;
```

Description

Returns 1 if the set is empty; otherwise returns 0.

TMISetAsVector::IsFull

TMISetAsVector class

Syntax

```
int isFull() const;
```

Description

Returns 0.

TMISetAsVector::LastThat

TMISetAsVector class

Syntax

```
T *LastThat(CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the set that satisfies a given condition. You supply a test function pointer, f, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the set meets the condition.

TMISetAsVectorIterator template

Syntax

```
template <class T, class Alloc> class TMISetAsVectorIterator;
```

Header File

sets.h

Description

Implements an iterator object to traverse TMISetAsVector objects.

Public Constructors

TMISetAsVectorIterator::TMISetAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

TMISetAsVectorIterator::TMISetAsVectorIterator

TMISetAsVectorIterator class

Syntax

```
TMISetAsVectorIterator( const TMISetAsVector<T,Alloc> &s )
```

Description

Constructs an object that iterates on TMISetAsVector objects.

TMISetAsVectorIterator::Current

TMISetAsVectorIterator class

Syntax

```
T *Current();
```

Description

Returns a pointer to the current object.

TMISetAsVectorIterator::Restart

TMISetAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration from the beginning.

Form 2: Restarts iteration over the specified range.

TMISetAsVectorIterator::operator ++

TMISetAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TISetAsVector template

Syntax

```
template <class T> class TISetAsVector;
```

Header File

sets.h

Description

Implements a set of pointers to objects of type T, using a vector as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Constructors

TISetAsVector::TISetAsVector

Public Member Functions

Add

Detach

Find

FirstThat

Flush

ForEach

GetItemsInContainer

HasMember

isEmpty

isFull

LastThat

TISetAsVector::CondFunc

TISetAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TISetAsVector::IterFunc

TISetAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TISetAsVector::TISetAsVector

Syntax

```
TISetAsVector( unsigned sz = DEFAULT_SET_SIZE )
```

Description

Constructs an empty, indirect set. sz represents the initial number of slots allocated.

TISetAsVector::Add

TISetAsVector class

Syntax

```
int Add( T *t )
```

Description

Adds the given object pointer to the set.

TISetAsVector::Detach

See Also TISetAsVector class

Syntax

```
int Detach( T *t, DeleteType dt = NoDelete )
```

Description

Removes the specified object pointer. The value of dt and the current ownership setting determine whether the object itself will be deleted. DeleteType is defined in the base class TShouldDelete as enum { NoDelete, DefDelete, Delete }. The default value of dt, NoDelete, means that the object will not be deleted regardless of ownership. With dt set to Delete, the object will be deleted regardless of ownership. If dt is set to DefDelete, the object will only be deleted if the set owns its elements.

See Also
TShouldDelete::ownsElements

TISetAsVector::Find

TISetAsVector class

Syntax

```
T *Find( T *t ) const;
```

Description

Returns a pointer to the object if found; otherwise returns 0.

TISetAsVector::FirstThat

TISetAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the set that satisfies a given condition. You supply a test-function pointer f that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the set meets the condition.

TISetAsVector::Flush

See Also   TISetAsVector class

Syntax

```
void Flush( TShouldDelete::DeleteType dt = TShouldDelete::DefDelete )
```

Description

Removes all the elements from the set without destroying the set. The value of dt determines whether the elements themselves are destroyed. By default, the ownership status of the set determines their fate, as explained in the Detach member function. You can also set dt to Delete and NoDelete.

See Also
Detach

TISetAsVector::ForEach

TISetAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the set. The args argument lets you pass arbitrary data to this function.

TISetAsVector::GetItemsInContainer

TISetAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of objects in the set.

TISetAsVector::HasMember

TISetAsVector class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found; otherwise returns 0.

TISetAsVector::IsEmpty

TISetAsVector class

Syntax

```
int isEmpty() const;
```

Description

Returns 1 if the set is empty; otherwise returns 0.

TISetAsVector::IsFull

TISetAsVector class

Syntax

```
int isFull() const;
```

Description

Returns 0.

TISetAsVector::LastThat

TISetAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the set that satisfies a given condition. You supply a test function pointer, f, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the set meets the condition. Note that LastThat creates its own internal iterator, so you can treat it as a "search" function.

TISetAsVectorIterator template

Syntax

```
template <class T> class TISetAsVectorIterator;
```

Header File

sets.h

Description

Implements an iterator object to traverse TISetAsVector objects.

Public Constructors

TISetAsVectorIterator::TISetAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

TISetAsVectorIterator::TISetAsVectorIterator

TISetAsVectorIterator class

Syntax

```
TISetAsVectorIterator( const TISetAsVector<T> &s )
```

Description

Constructs an object that iterates on TISetAsVector objects.

TISetAsVectorIterator::Current

TISetAsVectorIterator class

Syntax

```
T *Current();
```

Description

Returns a pointer to the current object.

TISetAsVectorIterator::Restart

TISetAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration from the beginning.

Form 2: Restarts iteration over the specified range.

TISetAsVectorIterator::operator ++

TISetAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TSet template

Header File

sets.h

Description

A simplified name for TSetAsVector.

Type Definitions

CondFunc

IterFunc

Public Constructors

TSet::TSet

Public Member Functions

Add

Detach

Find

FindMember

Flush

ForEach

GetItemsInContainer

HasMember

isEmpty

isFull

TSet::CondFunc

TSet class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TSet::IterFunc

TSet class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TSet::TSet

TSet class

Syntax

```
TSet( unsigned sz = DEFAULT_SET_SIZE ) :
```

Description

Constructs an empty set. sz represents the number of items the set can hold.

TSet::Add

TSet class

Syntax

```
int Add( const T& t )
```

Description

Adds the given object to the set.

TSet::Detach

See Also TSet class

Syntax

```
int Detach( const T& t )
```

Description

Removes the specified object. The value of dt and the current ownership setting determine whether the object itself will be deleted. DeleteType is defined in the base class TShouldDelete as enum { NoDelete, DefDelete, Delete }. The default value of dt, NoDelete, means that the object will not be deleted regardless of ownership. With dt set to Delete, the object will be deleted regardless of ownership. If dt is set to DefDelete, the object will be deleted only if the set owns its elements.

See Also
TShouldDelete::ownsElements

TSet::Find

TSet class

Syntax

```
virtual T *Find( const T& ) const;
```

Description

Returns a pointer to the given object if found; otherwise returns 0.

TSet::FindMember

TSet class

Syntax

```
T* FindMember( const T& t ) const;
```

Description

Returns a pointer to the given object if found; otherwise returns 0.

TSet::Flush

See Also TSet class

Syntax

```
void Flush()
```

Description

Removes all the elements from the set without destroying the set. The value of dt determines whether the elements themselves are destroyed. By default, the ownership status of the set determines their fate, as explained in the Detach member function. You can also set dt to Delete and NoDelete.

See Also
Detach

TSet::ForEach

TSet class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

ForEach creates an internal iterator to execute the given function for each element in the set. The args argument lets you pass arbitrary data to this function.

TSet::GetItemsInContainer

TSet class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of objects in the set.

TSet::HasMember

TSet class

Syntax

```
int HasMember( const T& t ) const;
```

Description

Returns 1 if the given object is found; otherwise returns 0.

TSet::IsEmpty

TSet class

Syntax

```
int isEmpty() const;
```

Description

Returns 1 if the set is empty; otherwise returns 0.

TSet::IsFull

TSet class

Syntax

```
int isFull() const;
```

Description

Returns 0.

TSetIterator template

Header File

sets.h

Description

A simplified name for TSetAsVectorIterator.

Public Constructors

TSetIterator::TSetIterator

Public Member Functions

Current

Restart

Operators

++

int

TSetIterator::TSetIterator

TSetIterator class

Syntax

```
TSetIterator( const TSet<T> &s )
```

Description

Constructs an object that iterates on TMSetAsVector objects.

TSetIterator::Current

TSetIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TSetIterator::Restart

TSetIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Restarts iteration from the beginning, or over the specified range.

TSetIterator::operator ++

TSetIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TSetIterator::operator int

TSetIterator class

Syntax

```
operator int() const;
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TShouldDelete class

Syntax

```
class TShouldDelete;
```

Header File

shddel.h

Description

TShouldDelete maintains the ownership state of an indirect container. The fate of objects that are removed from a container can be made to depend on whether the container owns its elements or not. Similarly, when a container is destroyed, ownership can dictate the fate of contained objects that are still in scope. As a virtual base class, TShouldDelete provides ownership control for all containers classes. The member function OwnsElements can be used either to report or to change the ownership status of a container. The member function DelObj is used to determine if objects in containers should be deleted or not.

Public Constructors

TShouldDelete

Public Data Members

DeleteType

Public Member Functions

OwnsElements

Protected Member Functions

DelObj

TShouldDelete::TShouldDelete

See Also   TShouldDelete class

Syntax

```
TShouldDelete( DeleteType dt = Delete )
```

Description

Creates a TShouldDelete object.

See Also

TShouldDelete::DelObj

TShouldDelete::DeleteType

TShouldDelete class

Syntax

```
enum DeleteType { NoDelete, DefDelete, Delete };
```

Description

Enumerates values to determine whether or not an object should be deleted upon removal from a container.

TShouldDelete::OwnsElements

TShouldDelete class

Form 1

```
int OwnsElements()
```

Form 2

```
void OwnsElements( int del )
```

Description

Form 1: Returns 1 if the container owns its elements; otherwise returns 0.

Form 2: Changes the ownership status as follows: if del is 0, ownership is turned off; otherwise ownership is turned on.

TShouldDelete::DelObj

See Also   TShouldDelete class

Syntax

```
int DelObj( DeleteType dt )
```

Description

Tests the state of ownership and returns 1 if the contained objects should be deleted or 0 if the contained elements should not be deleted. The factors determining this are the current ownership state, and the value of dt, as shown in the following table.

	delObj	
ownsElements	**No**	**Yes**
NoDelete	No	No
DefDelete	No	Yes
Delete	Yes	Yes

delObj returns 1 if (dt is Delete) or (dt is DefDelete and the container currently owns its elements). Thus a dt of NoDelete returns 0 (don't delete) regardless of ownership; a dt of Delete return 1 (do delete) regardless of ownership; and a dt of DefDelete returns 1 (do delete) if the elements are owned, but a 0 (don't delete) if the objects are not owned.

See Also
TShouldDelete::OwnsElements

TMStackAsVector template

Syntax

```
template <class T, class Alloc> class TMStackAsVector;
```

Header File

stacks.h

Description

Implements a managed stack of objects of type T, using a vector as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Constructors

TMStackAsVector::TMStackAsVector

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

IsEmpty

IsFull

LastThat

Pop

Push

Top

TMStackAsVector::CondFunc

TMStackAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMStackAsVector::IterFunc

TMStackAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMStackAsVector::TMStacAsVector

TMStackAsVector class

Syntax

```
TMStackAsVector( unsigned max = DEFAULT_STACK_SIZE )
```

Description

Constructs a managed, vector-implemented stack, with max indicating the maximum stack size.

TMStackAsVector::FirstThat

See Also TMStackAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the stack that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also
LastThat

TMStackAsVector::Flush

See Also   TMStackAsVector class

Syntax

```
void Flush()
```

Description

Flushes the stack without destroying it.

See Also

TShouldDelete::ownsElements

TMStackAsVector::ForEach

TMStackAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each stack element. ForEach creates an internal iterator to execute the given function for each element in the stack. The args argument lets you pass arbitrary data to this function.

TMStackAsVector::GetItemsInContainer

TMStackAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the stack.

TMStackAsVector::IsEmpty

TMStackAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the stack has no elements; otherwise returns 0.

TMStackAsVector::IsFull

TMStackAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the stack is full; otherwise returns 0.

TMStackAsVector::LastThat

See Also TMStackAsVector class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the stack that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also

FirstThat

ForEach

TMStackAsVector::Pop

See Also TMStackAsVector class

Syntax

```
T Pop()
```

Description

Removes the object from the top of the stack and returns the object. The fate of the popped object is determined by ownership.

See Also
TShouldDelete

TMStackAsVector::Push

TMStackAsVector class

Syntax

```
void Push( const T& t )
```

Description

Pushes an object on the top of the stack.

TMStackAsVector::Top

TMStackAsVector class

Syntax

```
Const T& Top() const;
```

Description

Returns but does not remove the object at the top of the stack.

TMStackAsVectorIterator template

Syntax

```
template <class T, class Alloc> class TMStackAsVectorIterator;
```

Header File

stacks.h

Description

Implements an iterator object for managed, vector-based stacks.

Public constructor

TMStackAsVectorIterator::TMStackAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TMStackAsVectorIterator::TMStacAsVectorIterator

TMStackAsVectorIterator class

Syntax

```
TMStackAsVectorIterator( const TMStackAsVector<T,Alloc> & s ) :
```

Description

Constructs an object that iterates on TMStackAsVector objects.

TMStackAsVectorIterator::Current

TMStackAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TMStackAsVectorIterator::Restart

TMStackAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TMStackAsVectorIterator::operator ++

TMStackAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMStackAsVectorIterator::operator int

TMStackAsVectorIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TStackAsVector template

Syntax

```
template <class T> class TStackAsVector;
```

Header File

stacks.h

Description

Implements a stack of objects of type T, using a vector as the underlying implementation, and TStandardAllocator for memory management.

Public Constructors

TStackAsVector::TStackAsVector

TStackAsVector::TStacAsVector

TStackAsVector class

Syntax

```
TStackAsVector( unsigned max = DEFAULT_STACK_SIZE )
```

Description

Constructs a vector-implemented stack, with max indicating the maximum stack size.

TStackAsVectorIterator template

Syntax

```
template <class T> class TStackAsVectorIterator;
```

Header File

stacks.h

Description

Implements an iterator object for managed, vector-based stacks.

Public Constructors

TStackAsVectorIterator::TStackAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TStackAsVectorIterator::TStacAsVectorIterator

TStackAsVectorIterator class

Syntax

```
TStackAsVectorIterator( const TStackAsVector<T> & s ) :
```

Description

Constructs an object that iterates on TStackAsVector objects.

TStackAsVectorIterator::Current

TStackAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TStackAsVectorIterator::Restart

TStackAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TStackAsVectorIterator::operator ++

TStackAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TStackAsVectorIterator::operator int

TStackAsVectorIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMIStackAsVector template

Syntax

```
template <class T, class Alloc> class TMIStackAsVector;
```

Header File

stacks.h

Description

TMIStackAsVector implements a managed stack of pointers to objects of type T, using a vector as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Constructors

TMIStackAsVector::TMIStackAsVector

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

IsEmpty

IsFull

LastThat

Pop

Push

Top

TMIStackAsVector::CondFunc

TMIStackAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMIStackAsVector::IterFunc

TMIStackAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMIStackAsVector::TMIStacAsVector

TMIStackAsVector class

Syntax

```
TMIStackAsVector( unsigned max = DEFAULT_STACK_SIZE )
```

Description

Constructs a managed, indirect, vector-implemented stack, with max indicating the maximum stack size.

TMIStackAsVector::FirstThat

See Also TMIStackAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the stack that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also
LastThat

TMIStackAsVector::Flush

See Also   TMIStackAsVector class

Syntax

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete )
```

Description

Flushes the stack without destroying it. The fate of any objects removed depends on the current ownership status and the value of the dt argument.

See Also

TShouldDelete::ownsElements

TMIStackAsVector::ForEach

TMIStackAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each stack element. ForEach creates an internal iterator to execute the given function for each element in the stack. The args argument lets you pass arbitrary data to this function.

TMIStackAsVector::GetItemsInContainer

TMIStackAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the stack.

TMIStackAsVector::IsEmpty

TMIStackAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the stack has no elements; otherwise returns 0.

TMIStackAsVector::IsFull

TMIStackAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the stack is full; otherwise returns 0.

TMIStackAsVector::LastThat

See Also TMIStackAsVector class

Syntax

```
T *LastThat( CondFunc, void *args ) const;
```

Description

Returns a pointer to the last object in the stack that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also

FirstThat

ForEach

TMIStackAsVector::Pop

See Also TMIStackAsVector class

Syntax

```
T *Pop()
```

Description

Removes the object from the top of the stack and returns a pointer to the object. The fate of the popped object is determined by ownership.

See Also
TShouldDelete

TMIStackAsVector::Push

TMIStackAsVector class

Syntax

```
void Push( T *t )
```

Description

Pushes a pointer to an object on the top of the stack.

TMIStackAsVector::Top

TMIStackAsVector class

Syntax

```
T *Top() const;
```

Description

Returns but does not remove the object pointer at the top of the stack.

TMIStackAsVectorIterator template

Syntax

```
template <class T, class Alloc> class TMIStackAsVectorIterator;
```

Header File

stacks.h

Description

Implements an iterator object for managed, indirect, vector-based stacks.

Public Constructors

TMIStackAsVectorIterator::TMIStackAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TMIStackAsVectorIterator::TMIStacAsVectorIterator

TMIStackAsVectorIterator class

Syntax

```
TMIStackAsVectorIterator( const TMIStackAsVector<T,Alloc> & s )
```

Description

Constructs an object that iterates on TMIStackAsVector objects.

TMIStackAsVectorIterator::Current

TMIStackAsVectorIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TMIStackAsVectorIterator::Restart

TMIStackAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TMIStackAsVectorIterator::operator ++

TMIStackAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMIStackAsVectorIterator::operator int

TMIStackAsVectorIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TIStackAsVector template

Syntax

```
template <class T> class TIStackAsVector;
```

Header File

stacks.h

Description

Implements an indirect stack of pointers to objects of type T, using a vector as the underlying implementation.

Public Constructors

TIStackAsVector::TIStackAsVector

Type Definitions

CondFunc

IterFunc

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

IsEmpty

IsFull

LastThat

Pop

Push

Top

TIStackAsVector::CondFunc

TIStackAsVector class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TIStackAsVector::IterFunc

TIStackAsVector class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TIStackAsVector::TIStacAsVector

TIStackAsVector class

Syntax

```
TIStackAsVector( unsigned max = DEFAULT::STACK::SIZE ) :
  TMIStackAsVector<T,TStandardAllocator>( max )
```

Description

Constructs an indirect, vector-implemented stack, with max indicating the maximum stack size.

TIStackAsVector::FirstThat

See Also TIStackAsVector class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the stack that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also
LastThat

TIStackAsVector::Flush

See Also TIStackAsVector class

Syntax

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete )
```

Description

Flushes the stack without destroying it. The fate of any objects removed depends on the current ownership status and the value of the dt argument.

See Also
TShouldDelete::ownsElements

TIStackAsVector::ForEach

TIStackAsVector class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each stack element. ForEach creates an internal iterator to execute the given function for each element in the stack. The args argument lets you pass arbitrary data to this function.

TIStackAsVector::GetItemsInContainer

TIStackAsVector class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the stack.

TIStackAsVector::IsEmpty

TIStackAsVector class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the stack has no elements; otherwise returns 0.

TIStackAsVector::IsFull

TIStackAsVector class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the stack is full; otherwise returns 0.

TIStackAsVector::LastThat

See Also TIStackAsVector class

Syntax

```
T *LastThat( CondFunc, void *args ) const;
```

Description

Returns a pointer to the last object in the stack that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also

FirstThat

ForEach

TIStackAsVector::Pop

See Also   TIStackAsVector class

Syntax

```
T *Pop()
```

Description

Removes the object from the top of the stack and returns a pointer to the object. The fate of the popped object is determined by ownership.

See Also
TShouldDelete

TIStackAsVector::Push

TIStackAsVector class

Syntax

```
void Push( T *t )
```

Description

Pushes a pointer to an object on the top of the stack.

TIStackAsVector::Top

TIStackAsVector class

Syntax

```
T *Top() const;
```

Description

Returns but does not remove the object pointer at the top of the stack.

TIStackAsVectorIterator template

Syntax

```
template <class T> class TIStackAsVector;
```

Header File

stacks.h

Description

Implements an iterator object for indirect, vector-based stacks.

Public Constructors

TMIStackAsVectorIterator::TMIStackAsVectorIterator

Public Member Functions

Current

Restart

Operators

++

int

TIStackAsVectorIterator::TIStacAsVectorIterator

TIStackAsVectorIterator class

Syntax

```
TMIStackAsVectorIterator( const TMIStackAsVector<T,Alloc> & s )
```

Description

Constructs an object that iterates on TIStackAsVector objects.

TIStackAsVectorIterator::Current

TIStackAsVectorIterator class

Syntax

```
T *Current();
```

Description

Returns a pointer to the current object.

TIStackAsVectorIterator::Restart

TIStackAsVectorIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TIStackAsVectorIterator::operator ++

TIStackAsVectorIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TIStackAsVectorIterator::operator int

TIStackAsVectorIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMStackAsList template

Syntax

```
template <class T, class Alloc> class TMStackAsList;
```

Header File

stacks.h

Description

Implements a managed stack of objects of type T, using a list as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Constructors

TMStackAsList::TMStackAsList

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

IsEmpty

IsFull

LastThat

Pop

Push

Top

TMStackAsList::CondFunc

TMStackAsList class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMStackAsList::IterFunc

TMStackAsList class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMStackAsList::TMStacAsList

TMStackAsList class

Syntax

```
TMStackAsList( unsigned max = DEFAULT_STACK_SIZE )
```

Description

Constructs a managed, vector-implemented stack, with max indicating the maximum stack size.

TMStackAsList::FirstThat

See Also   TMStackAsList class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the stack that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also
LastThat

TMStackAsList::Flush

See Also TMStackAsList class

Syntax

```
void Flush()
```

Description

Flushes the stack without destroying it.

See Also
TShouldDelete::ownsElements

TMStackAsList::ForEach

TMStackAsList class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each stack element. ForEach creates an internal iterator to execute the given function for each element in the stack. The args argument lets you pass arbitrary data to this function.

TMStackAsList::GetItemsInContainer

TMStackAsList class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the stack.

TMStackAsList::IsEmpty

TMStackAsList class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the stack has no elements; otherwise returns 0.

TMStackAsList::IsFull

TMStackAsList class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the stack is full; otherwise returns 0.

TMStackAsList::LastThat

See Also TMStackAsList class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the stack that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also

FirstThat

ForEach

TMStackAsList::Pop

See Also   TMStackAsList class

Syntax

```
T Pop()
```

Description

Removes the object from the top of the stack and returns the object. The fate of the popped object is determined by ownership.

See Also
TShouldDelete

TMStackAsList::Push

TMStackAsList class

Syntax

```
void Push( const T& t )
```

Description

Pushes an object on the top of the stack.

TMStackAsList::Top

TMStackAsList class

Syntax

```
const T& Top() const;
```

Description

Returns but does not remove the object at the top of the stack.

TMStackAsListIterator template

Syntax

```
template <class T, class Alloc> class TMStackAsListIterator;
```

Header File

stacks.h

Description

Implements an iterator object for managed, list-based stacks.

Public Constructors

TMStackAsListIterator::TMStackAsListIterator

Public Member Functions

Current

Restart

Operators

int

++

TMStackAsListIterator::TMStacAsListIterator

TMStackAsListIterator class

Syntax

```
TMStackAsListIterator( const TMStackAsList<T,Alloc> & s ) :
  TMListIteratorImp<T,Alloc>(s.Data)
```

Description

Constructs an object that iterates on TMStackAsList objects.

TMStackAsListIterator::Current

TMStackAsListIterator class

Syntax

```
const T& Current()
```

Description

Returns the current object.

TMStackAsListIterator::Restart

TMStackAsListIterator class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TMStackAsListIterator::operator int

TMStackAsListIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMStackAsListIterator::operator ++

TMStackAsListIterator class

Form 1

```
const T& operator ++ ( int )
```

Form 2

```
const T& operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TStackAsList template

Syntax

```
template <class T> class TStackAsList;
```

Header File

stacks.h

Description

Implements a managed stack of objects of type T, using a list as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Constructors

TStackAsList::TStackAsList

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

IsEmpty

IsFull

LastThat

Pop

Push

Top

TStackAsList::CondFunc

TStackAsList class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TStackAsList::IterFunc

TStackAsList class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TStackAsList::TStacAsList

TStackAsList class

Syntax

```
TStackAsList( unsigned max = DEFAULT_STACK_SIZE )
```

Description

Constructs a managed, vector-implemented stack, with max indicating the maximum stack size.

TStackAsList::FirstThat

See Also TStackAsList class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the stack that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also
LastThat

TStackAsList::Flush

See Also TStackAsList class

Syntax

```
void Flush()
```

Description

Flushes the stack without destroying it.

See Also

TShouldDelete::ownsElements

TStackAsList::ForEach

TStackAsList class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each stack element. ForEach creates an internal iterator to execute the given function for each element in the stack. The args argument lets you pass arbitrary data to this function.

TStackAsList::GetItemsInContainer

TStackAsList class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the stack.

TStackAsList::IsEmpty

TStackAsList class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the stack has no elements; otherwise returns 0.

TStackAsList::IsFull

TStackAsList class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the stack is full; otherwise returns 0.

TStackAsList::LastThat

See Also TStackAsList class

Syntax

```
T *LastThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the last object in the stack that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also

FirstThat

ForEach

TStackAsList::Pop

See Also   TStackAsList class

Syntax

```
T Pop()
```

Description

Removes the object from the top of the stack and returns the object. The fate of the popped object is determined by ownership.

See Also
TShouldDelete

TStackAsList::Push

TStackAsList class

Syntax

```
void Push( const T& t )
```

Description

Pushes an object on the top of the stack.

TStackAsList::Top

TStackAsList class

Syntax

```
const T& Top() const;
```

Description

Returns but does not remove the object at the top of the stack.

TStackAsListIterator template

Syntax

```
template <class T> class TStackAsListIterator;
```

Header File

stacks.h

Description

Implements an iterator object for list-based stacks.

Public Constructor

TStackAsListIterator::TStackAsListIterator

Public Member Functions

Current

Restart

Operators

++

int

TStackAsListIterator::TStacAsListIterator

TStackAsListIterator class

Syntax

```
TStackAsListIterator( const TStackAsList<T> & s ) :
  TMStackAsListIterator<T,TStandardAllocator>(s)
```

Description

Constructs an object that iterates on TIStackAsVector objects.

TStackAsListIterator::Current

TStackAsListIterator class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TStackAsListIterator::Restart

TStackAsListIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TStackAsListIterator::operator ++

TStackAsListIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TStackAsListIterator::operator int

TStackAsListIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMIStackAsList template

Syntax

```
template <class T, class Alloc> class TMIStackAsList;
```

Header File

stacks.h

Description

Implements a managed stack of pointers to objects of type T, using a linked list as the underlying implementation.

Type Definitions

CondFunc

IterFunc

Public Constructors

TMIStackAsList::TMIStackAsList

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

IsEmpty

IsFull

LastThat

Pop

Push

Top

TMIStackAsList::CondFunc

TMIStackAsList class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMIStackAsList::IterFunc

TMIStackAsList class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMIStackAsList::TMIStacAsList

TMIStackAsList class

Syntax

```
TMIStackAsList( unsigned max = DEFAULT_STACK_SIZE )
```

Description

Constructs a managed, indirect, vector-implemented stack, with max indicating the maximum stack size.

TMIStackAsList::FirstThat

See Also TMIStackAsList class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the stack that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also
LastThat

TMIStackAsList::Flush

See Also TMIStackAsList class

Syntax

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete )
```

Description

Flushes the stack without destroying it. The fate of any objects removed depends on the current ownership status and the value of the dt argument.

See Also

TShouldDelete::ownsElements

TMIStackAsList::ForEach

TMIStackAsList class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each stack element. ForEach creates an internal iterator to execute the given function for each element in the stack. The args argument lets you pass arbitrary data to this function.

TMIStackAsList::GetItemsInContainer

TMIStackAsList class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the stack.

TMIStackAsList::IsEmpty

TMIStackAsList class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the stack has no elements; otherwise returns 0.

TMIStackAsList::IsFull

TMIStackAsList class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the stack is full; otherwise returns 0.

TMIStackAsList::LastThat

See Also TMIStackAsList class

Syntax

```
T *LastThat( CondFunc, void *args ) const;
```

Description

Returns a pointer to the last object in the stack that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also

FirstThat

ForEach

TMIStackAsList::Pop

See Also TMIStackAsList class

Syntax

```
T *Pop()
```

Description

Removes the object from the top of the stack and returns a pointer to the object. The fate of the popped object is determined by ownership.

See Also
TShouldDelete

TMIStackAsList::Push

TMIStackAsList class

Syntax

```
void Push( T *t )
```

Description

Pushes a pointer to an object on the top of the stack.

TMIStackAsList::Top

TMIStackAsList class

Syntax

```
T *Top() const;
```

Description

Returns but does not remove the object pointer at the top of the stack.

TMIStackAsListIterator template

Syntax

```
template <class T, class Alloc> class TMIStackAsListIterator;
```

Header File

stacks.h

Description

Implements an iterator object for managed, indirect, list-based stacks.

Public Constructors

TMIStackAsListIterator::TMIStackAsListIterator

Public Member Functions

Current

Restart

Operators

++

TMIStackAsListIterator::TMIStacAsListIterator

TMIStackAsListIterator class

Syntax

```
TMIStackAsListIterator( const TMIStackAsList<T,Alloc> & s )
```

Description

Constructs an object that iterates on TMIStackAsList objects.

TMIStackAsListIterator::Current

TMIStackAsListIterator class

Syntax

```
T *Current()
```

Description

Returns the current object pointer.

TMIStackAsListIterator::Restart

TMIStackAsListIterator class

Syntax

```
void Restart()
```

Description

Restarts iteration from the beginning of the list.

TMIStackAsListIterator::operator ++

TMIStackAsListIterator class

Form 1

```
T *operator ++ (int)
```

Form 2

```
T *operator ++ ()
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TIStackAsList template

Syntax

```
template <class T> class TIStackAsList;
```

Header File

stacks.h

Description

Implements TMIStackAsList with the standard allocator TStandardAllocator.

Type Definitions

CondFunc

IterFunc

Public Constructors

TIStackAsList::TIStackAsList

Public Member Functions

FirstThat

Flush

ForEach

GetItemsInContainer

IsEmpty

IsFull

LastThat

Pop

Push

Top

TIStackAsList::CondFunc

TIStackAsList class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TIStackAsList::IterFunc

TIStackAsList class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TIStackAsList::TIStacAsList

TIStackAsList class

Syntax

```
TIStackAsList( unsigned max = DEFAULT_STACK_SIZE )
```

Description

Constructs a managed, indirect, vector-implemented stack, with max indicating the maximum stack size.

TIStackAsList::FirstThat

See Also   TIStackAsList class

Syntax

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Description

Returns a pointer to the first object in the stack that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also
LastThat

TIStackAsList::Flush

See Also	TIStackAsList class

Syntax

```
void Flush( TShouldDelete::DeleteType = TShouldDelete::DefDelete )
```

Description

Flushes the stack without destroying it. The fate of any objects removed depends on the current ownership status and the value of the dt argument.

See Also
TShouldDelete::ownsElements

TIStackAsList::ForEach

TIStackAsList class

Syntax

```
void ForEach( IterFunc iter, void *args )
```

Description

Executes function iter for each stack element. ForEach creates an internal iterator to execute the given function for each element in the stack. The args argument lets you pass arbitrary data to this function.

TIStackAsList::GetItemsInContainer

TIStackAsList class

Syntax

```
int GetItemsInContainer() const;
```

Description

Returns the number of items in the stack.

TIStackAsList::IsEmpty

TIStackAsList class

Syntax

```
int IsEmpty() const;
```

Description

Returns 1 if the stack has no elements; otherwise returns 0.

TIStackAsList::IsFull

TIStackAsList class

Syntax

```
int IsFull() const;
```

Description

Returns 1 if the stack is full; otherwise returns 0.

TIStackAsList::LastThat

See Also TIStackAsList class

Syntax

```
T *LastThat( CondFunc, void *args ) const;
```

Description

Returns a pointer to the last object in the stack that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the stack meets the condition.

See Also

FirstThat

ForEach

TIStackAsList::Pop

See Also TIStackAsList class

Syntax

```
T *Pop()
```

Description

Removes the object from the top of the stack and returns a pointer to the object. The fate of the popped object is determined by ownership.

See Also
TShouldDelete

TIStackAsList::Push

TIStackAsList class

Syntax

```
void Push( T *t )
```

Description

Pushes a pointer to an object on the top of the stack.

TIStackAsList::Top

TIStackAsList class

Syntax

```
T *Top() const;
```

Description

Returns but does not remove the object pointer at the top of the stack.

TIStackAsListIterator template

Syntax

```
template <class T> class TIStackAsListIterator;
```

Header File

stacks.h

Description

Implements an iterator object for indirect, list-based stacks.

Public Constructors

TIStackAsListIterator::TIStackAsListIterator

Public Member Functions

Current

Restart

Operators

++

int

TIStackAsListIterator::TIStacAsListIterator

TIStackAsListIterator class

Syntax

```
TIStackAsListIterator( const TIStackAsList<T> & s )
```

Description

Constructs an object that iterates on TIStackAsList objects.

TIStackAsListIterator::Current

TIStackAsListIterator class

Syntax

```
T *Current();
```

Description

Returns a pointer to the current object.

TIStackAsListIterator::Restart

TIStackAsListIterator class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TIStackAsListIterator::operator ++

TIStackAsListIterator class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TIStackAsListIterator::operator int

TIStackAsListIterator class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TStack template

Header File
stacks.h

Description
A simplified name for TStackAsVector.

TStackIterator template

Header File
stacks.h

Description
A simplified name for TStackAsVectorIterator.

TMVectorImp template

Syntax

```
template <class T, class Alloc> class TMVectorImp;
```

Header File

vectimp.h

Description

Implements a managed vector of objects of type T. TMVectorImp assumes that T has meaningful copy semantics, and a default constructor.

Type Definitions

CondFunc

IterFunc

Public Constructors

TMVectorImp::TMVectorImp

Public Member Functions

FirstThat

Flush

ForEach

GetDelta

LastThat

Limit

Resize

Top

Operators

[]

=

Protected Data Members

Lim

Protected Member Functions

Zero

TMVectorImp::CondFunc

TMVectorImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMVectorImp::IterFunc

TMVectorImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMVectorImp::TMVectorImp

TMVectorImp class

Form 1

```
TMVectorImp();
```

Form 2

```
TMVectorImp( unsigned sz, unsigned = 0 );
```

Form 3

```
TMVectorImp( const TMVectorImp<T,Alloc> & );
```

Description

Form 1: Constructs a vector with no entries.

Form 2: Constructs a vector of sz objects, initialized by default to 0.

Form 3: Constructs a vector copy.

TMVectorImp::FirstThat

See Also TMVectorImp class

Form 1

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *FirstThat( CondFunc cond, void *args, unsigned start, unsigned stop )
  const;
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

Form 2: This version of FirstThat allows you to specify a range to be searched. Returns a pointer to the first object in the vector that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

See Also
LastThat

TMVectorImp::Flush

See Also TMVectorImp class

Syntax

```
void Flush( unsigned stop = UINT_MAX, unsigned start = 0 );
```

Description

Flushes the vector without destroying it. The fate of any objects removed depends on the current ownership status and the value of the first argument.

See Also
TShouldDelete::ownsElements

TMVectorImp::ForEach

See Also TMVectorImp class

Form 1

```
void ForEach( IterFunc iter, void *args )
```

Form 2

```
void ForEach( IterFunc iter, void *args, unsigned start, unsigned stop );
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition. ForEach creates an internal iterator to execute the given function for each element in the vector. The args argument lets you pass arbitrary data to this function.

Form 2: This version allows you to specify a range.

See Also
LastThat

TMVectorImp::GetDelta

TMVectorImp class

Syntax

```
virtual unsigned GetDelta( ) const;
```

Description

Returns the growth delta for the vector.

TMVectorImp::LastThat

See Also TMVectorImp class

Form 1

```
T *LastThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *LastThat( CondFunc cond, void *args, unsigned start, unsigned stop )
  const;
```

Description

Form 1: Returns a pointer to the last object in the vector that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

Form 2: This version allows you to specify a range.

See Also

FirstThat

ForEach

TMVectorImp::Limit

TMVectorImp class

Syntax

```
unsigned Limit() const;
```

Description

Returns the number of items that the vector can hold.

TMVectorImp::Resize

TMVectorImp class

Syntax

```
void Resize( unsigned sz, unsigned offset = 0 );
```

Description

Creates a new vector of size sz. The existing vector is copied to the expanded vector, then deleted. In a vector of pointers the entries are zeroed. In an vector of objects the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the vector has to be extended downward.

TMVectorImp::Top

TMVectorImp class

Syntax

```
virtual unsigned Top() const;
```

Description

Returns the index of the current top element. For plain vectors Top returns Lim; for counted and sorted vectors Top returns the current insertion point.

TMVectorImp::operator [ ]

TMVectorImp class

Syntax

```
T & operator [] ( unsigned index ) const;
```

Description

Returns a reference to the object at index.

TMVectorImp::operator =

TMVectorImp class

Syntax

```
const TMVectorImp<T,Alloc> & operator = ( const TMVectorImp<T,Alloc> & );
```

Description

Provides the vector assignment operator.

TMVectorImp::Lim

TMVectorImp class

Syntax

```
unsigned Lim;
```

Description

Lim stores the upper limit for indexes into the vector.

TMVectorImp::Zero

TMVectorImp class

Syntax

```
virtual void Zero( unsigned, unsigned )
```

Description

Provides for zeroing vector contents within the specified range.

TMVectorIteratorImp template

Syntax

```
template <class T, class Alloc> class TMVectorIteratorImp;
```

Header File

vectimp.h

Description

Implements a vector iterator that works with any direct, managed vector of objects of type T. For indirect vector iterators, see TMIVectorIteratorImp.

Public Constructors

TMVectorIteratorImp::TMVectorIteratorImp

Public Member Functions

Current

Restart

Operators

++

int

TMVectorIteratorImp::TMVectorIteratorImp

TMVectorIteratorImp class

Form 1

```
TMVectorIteratorImp( const TMVectorImp<T,Alloc> &v )
```

Form 2

```
TMVectorIteratorImp( const TMVectorImp<T,Alloc> &v, unsigned start,
  unsigned stop )
```

Description

Form 1: Creates an iterator object to traverse TMVectorImp objects.

Form 2: Creates an iterator object to traverse TMVectorImp objects. A range can be specified.

TMVectorIteratorImp::Current

TMVectorIteratorImp class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TMVectorIteratorImp::Restart

TMVectorIteratorImp class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TMVectorIteratorImp::operator ++

TMVectorIteratorImp class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMVectorIteratorImp::operator int

TMVectorIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TVectorImp template

Syntax

```
template <class T> class TVectorImp;
```

Header File

vectimp.h

Description

Implements a vector of objects of type T. TVectorImp assumes that T has meaningful copy semantics, and a default constructor.

Type Definitions

CondFunc

IterFunc

Public Constructors

TVectorImp::TVectorImp

Public Member Functions

FirstThat

Flush

ForEach

GetDelta

LastThat

Limit

Resize

Top

Operators

[]

=

Protected Data Members

Lim

Protected Member Functions

Zero

TVectorImp::CondFunc

TVectorImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TVectorImp::IterFunc

TVectorImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TVectorImp::TVectorImp

TVectorImp class

Form 1

```
TVectorImp()
```

Form 2

```
TVectorImp( unsigned sz, unsigned = 0 )
```

Form 3

```
TVectorImp( const TVectorImp<T> &v )
```

Description

Form 1: Constructs a vector with no entries.

Form 2: Constructs a vector of sz objects, initialized by default to 0.

Form 3: Constructs a vector copy.

TVectorImp::FirstThat

See Also TVectorImp class

Form 1

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *FirstThat( CondFunc cond, void *args, unsigned start, unsigned stop )
  const;
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

Form 2: This version of FirstThat allows you to specify a range to be searched. Returns a pointer to the first object in the vector that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

See Also
LastThat

TVectorImp::Flush

See Also TVectorImp class

Syntax

```
void Flush( unsigned stop = UINT_MAX, unsigned start = 0 );
```

Description

Flushes the vector without destroying it. The fate of any objects removed depends on the current ownership status and the value of the first argument.

See Also
TShouldDelete::ownsElements

TVectorImp::ForEach

See Also TVectorImp class

Form 1

```
void ForEach( IterFunc iter, void *args )
```

Form 2

```
void ForEach( IterFunc iter, void *args, unsigned start, unsigned stop );
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition. ForEach creates an internal iterator to execute the given function for each element in the vector. The args argument lets you pass arbitrary data to this function.

Form 2: This version allows you to specify a range.

See Also
LastThat

TVectorImp::GetDelta

TVectorImp class

Syntax

```
virtual unsigned GetDelta( ) const;
```

Description

Returns the growth delta for the vector.

TVectorImp::LastThat

See Also TVectorImp class

Form 1

```
T *LastThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *LastThat( CondFunc cond, void *args, unsigned start, unsigned stop )
  const;
```

Description

Form 1: Returns a pointer to the last object in the vector that satisfies a given condition. You supply a test function pointer, cond, that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

Form 2: This version allows you to specify a range.

See Also

FirstThat

ForEach

TVectorImp::Limit

TVectorImp class

Syntax

```
unsigned Limit() const;
```

Description

Returns the number of items that the vector can hold.

TVectorImp::Resize

TVectorImp class

Syntax

```
void Resize( unsigned sz, unsigned offset = 0 );
```

Description

Creates a new vector of size sz. The existing vector is copied to the expanded vector, then deleted. In a vector of pointers the entries are zeroed. In an vector of objects the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the vector has to be extended downward.

TVectorImp::Top

TVectorImp class

Syntax

```
virtual unsigned Top() const;
```

Description

Returns the index of the current top element. For plain vectors Top returns Lim; for counted and sorted vectors Top returns the current insertion point.

TVectorImp::operator [ ]

TVectorImp class

Syntax

```
T & operator [] ( unsigned index ) const;
```

Description

Returns a reference to the object at index.

TVectorImp::operator =

TVectorImp class

Syntax

```
const TVectorImp<T,Alloc> & operator = ( const TVectorImp<T,Alloc> & );
```

Description

Provides the vector assignment operator.

TVectorImp::Lim

TVectorImp class

Syntax

```
unsigned Lim;
```

Description

Lim stores the upper limit for indexes into the vector.

TVectorImp::Zero

TVectorImp class

Syntax

```
virtual void Zero( unsigned, unsigned )
```

Description

Provides for zeroing vector contents within the specified range.

TVectorIteratorImp template

Syntax

```
template <class T> class TVectorIteratorImp;
```

Header File

vectimp.h

Description

Implements a vector iterator that works with any direct vector of objects of type T.

Public Constructors

TVectorIteratorImp::TVectorIteratorImp

Public Member Functions

Current

Restart

Operators

++

int

TVectorIteratorImp::TVectorIteratorImp

TVectorIteratorImp class

Form 1

```
TVectorIteratorImp( const TVectorImp<T> &v )
```

Form 2

```
TVectorIteratorImp( const TVectorImp<T> &v, unsigned start, unsigned stop )
```

Description

Form 1: Creates an iterator object to traverse TVectorImp objects.

Form 2: Creates an iterator object to traverse TVectorImp objects. A range can be specified.

TVectorIteratorImp::Current

TVectorIteratorImp class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TVectorIteratorImp::Restart

TVectorIteratorImp class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TVectorIteratorImp::operator ++

TVectorIteratorImp class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TVectorIteratorImp::operator int

TVectorIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMCVectorImp template

Syntax

```
template <class T, class Alloc> class TMCVectorImp;
```

Header File

vectimp.h

Description

Implements a managed, counted vector of objects of type T. TMCVectorImp assumes that T has meaningful copy semantics, and a default constructor.

Public Constructors

TMCVectorImp::TMCVectorImp

Public Member Functions

Add

AddAt

Count

Detach

Find

GetDelta

Protected Data Members

Count_

Delta

Top

TMCVectorImp::TMCVectorImp

TMCVectorImp class

Form 1

```
TMCVectorImp();
```

Form 2

```
TMCVectorImp( unsigned sz, unsigned = 0 );
```

Description

Form 1: Constructs a vector with no entries.

Form 2: Constructs a vector of sz objects, initialized by default to 0.

TMCVectorImp::Add

TMCVectorImp class

Syntax

```
int Add( const T& t );
```

Description

Adds an object to the vector and increments Count_.

TMCVectorImp::AddAt

TMCVectorImp class

Syntax

```
int AddAt( const T&, unsigned );
```

Description

Adds an object to the vector at the specified location, and increments Count_.

TMCVectorImp::Count

TMCVectorImp class

Syntax

```
unsigned Count( ) const;
```

Description

Returns Count_.

TMCVectorImp::Detach

TMCVectorImp class

Form 1

```
int Detach( unsigned loc );
```

Form 2

```
int Detach( const T& loc );
```

Description

Remove by specifying the object or its index.

TMCVectorImp::Find

TMCVectorImp class

Syntax

```
virtual unsigned Find( const T& ) const;
```

Description

Finds the specified object and returns the object's index; otherwise returns INT_MAX.

TMCVectorImp::GetDelta

TMCVectorImp class

Syntax

```
virtual unsigned GetDelta( ) const;
```

Description

Returns Delta.

TMCVectorImp::Count_

TMCVectorImp class

Syntax

```
unsigned Count_;
```

Description

Maintains the number of objects in the vector.

TMCVectorImp::Delta

TMCVectorImp class

Syntax

```
unsigned Delta;
```

Description

Specifies the size increment to be used when the vector grows.

TMCVectorImp::Top

TMCVectorImp class

Syntax

```
virtual unsigned Top( ) const;
```

Description

Returns Count_.

TMCVectorIteratorImp template

Syntax

```
template <class T, class Alloc> class TMCVectorIteratorImp;
```

Header File

vectimp.h

Description

Implements a vector iterator that works with any direct, managed, counted vector of objects of type T.

Public Constructors

TMCVectorIteratorImp::TMCVectorIteratorImp

Public Member Functions

Current

Restart

Operators

++

int

TMCVectorIteratorImp::TMCVectorIteratorImp

TMCVectorIteratorImp class

Form 1

```
TMCVectorIteratorImp( const TMCVectorImp<T,Alloc> &v )
```

Form 2

```
TMVectorIteratorImp( const TMCVectorImp<T,Alloc> &v, unsigned start,
  unsigned stop )
```

Description

Form 1: Creates an iterator object to traverse TMCVectorImp objects.

Form 2: Creates an iterator object to traverse TMCVectorImp objects. A range can be specified.

TMCVectorIteratorImp::Current

TMCVectorIteratorImp class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TMCVectorIteratorImp::Restart

TMCVectorIteratorImp class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TMCVectorIteratorImp::operator ++

TMCVectorIteratorImp class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMCVectorIteratorImp::operator int

TMCVectorIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TCVectorImp template

Syntax

```
template <class T> class TCVectorImp;
```

Header File

vectimp.h

Description

Implements a counted vector of objects of type T. TCVectorImp assumes that T has meaningful copy semantics, and a default constructor.

Public Constructors

TCVectorImp::TCVectorImp

Public Member Functions

Add

AddAt

Count

Detach

Find

GetDelta

Protected Data Members

Count_

Delta

Top

TCVectorImp::TCVectorImp

TCVectorImp class

Form 1

```
TCVectorImp();
```

Form 2

```
TMCVectorImp( unsigned sz, unsigned = 0 );
```

Description

Form 1: Constructs a vector with no entries.

Form 2: Constructs a vector of sz objects, initialized by default to 0.

TCVectorImp::Add

TCVectorImp class

Syntax

```
int Add( const T& t );
```

Description

Adds an object to the vector and increments Count_.

TCVectorImp::AddAt

TCVectorImp class

Syntax

```
int AddAt( const T&, unsigned );
```

Description

Adds an object to the vector at the specified location, and increments Count_.

TCVectorImp::Count

TCVectorImp class

Syntax

```
unsigned Count( ) const;
```

Description

Returns Count_.

TCVectorImp::Detach

TCVectorImp class

Form 1

```
int Detach( unsigned loc );
```

Form 2

```
int Detach( const T& loc );
```

Description

Remove by specifying the object or its index.

TCVectorImp::Find

TCVectorImp class

Syntax

```
virtual unsigned Find( const T& ) const;
```

Description

Finds the specified object and returns the object's index; otherwise returns INT_MAX.

TCVectorImp::GetDelta

TCVectorImp class

Syntax

```
virtual unsigned GetDelta( ) const;
```

Description

Returns Delta.

TCVectorImp::Count_

TCVectorImp class

Syntax

```
unsigned Count_;
```

Description

Maintains the number of objects in the vector.

TCVectorImp::Delta

TCVectorImp class

Syntax

```
unsigned Delta;
```

Description

Specifies the size increment to be used when the vector grows.

TCVectorImp::Top

TCVectorImp class

Syntax

```
virtual unsigned Top( ) const;
```

Description

Returns Count_.

TCVectorIteratorImp template

Syntax

```
template <class T> class TCVectorIteratorImp;
```

Header File

vectimp.h

Description

Implements a vector iterator that works with any direct, counted vector of objects of type T.

Public Constructors

TCVectorIteratorImp::TCVectorIteratorImp

Public Member Functions

Current

Restart

Operators

++

int

TCVectorIteratorImp::TCVectorIteratorImp

TCVectorIteratorImp class

Form 1

```
TCVectorIteratorImp( const TCVectorImp<T> &v )
```

Form 2

```
TCVectorIteratorImp( const TCVectorImp<T> &v, unsigned start, unsigned stop
  )
```

Description

Form 1: Creates an iterator object to traverse TCVectorImp objects.

Form 2: Creates an iterator object to traverse TCVectorImp objects. A range can be specified.

TCVectorIteratorImp::Current

TCVectorIteratorImp class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TCVectorIteratorImp::Restart

TCVectorIteratorImp class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TCVectorIteratorImp::operator ++

TCVectorIteratorImp class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TCVectorIteratorImp::operator int

TCVectorIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMSVectorImp template

Syntax

```
template <class T, class Alloc> class TMSVectorImp;
```

Header File

vectimp.h

Description

Implements a managed, sorted vector of objects of type T. TMSVectorImp assumes that T has meaningful copy semantics, a meaningful < operator, and a default constructor.

Public Constructors

TMSVectorImp::TMSVectorImp

Public Member Functions

Add

AddAt

Count

Detach

Find

GetDelta

Protected Data Members

Count_

Delta

Top

TMSVectorImp::TMSVectorImp

TMSVectorImp class

Form 1

```
TMSVectorImp()
```

Form 2

```
TMSVectorImp( unsigned sz, unsigned d = 0 )
```

Description

Form 1: Constructs a vector with no entries.

Form 2: Constructs a vector of sz objects, initialized by default to 0.

TMSVectorImp::Add

TMSVectorImp class

Syntax

```
int Add( const T& t );
```

Description

Adds an object to the vector and increments Count_.

TMSVectorImp::AddAt

TMSVectorImp class

Syntax

```
int AddAt( const T&, unsigned );
```

Description

Adds an object to the vector at the specified location, and increments Count_.

TMSVectorImp::Count

TMSVectorImp class

Syntax

```
unsigned Count( ) const;
```

Description

Returns Count_.

TMSVectorImp::Detach

TMSVectorImp class

Form 1

```
int Detach( unsigned loc );
```

Form 2

```
int Detach( const T& loc );
```

Description

Remove by specifying the object or its index.

TMSVectorImp::Find

TMSVectorImp class

Syntax

```
virtual unsigned Find( const T& ) const;
```

Description

Finds the specified object and returns the object's index; otherwise returns INT_MAX.

TMSVectorImp::GetDelta

TMSVectorImp class

Syntax

```
virtual unsigned GetDelta( ) const;
```

Description

Returns Delta.

TMSVectorImp::Count_

TMSVectorImp class

Syntax

```
unsigned Count_;
```

Description

Maintains the number of objects in the vector.

TMSVectorImp::Delta

TMSVectorImp class

Syntax

```
unsigned Delta;
```

Description

Specifies the size increment to be used when the vector grows.

TMSVectorImp::Top

TMSVectorImp class

Syntax

```
virtual unsigned Top( ) const;
```

Description

Returns Count_.

TMSVectorIteratorImp template

Syntax

```
template <class T, class Alloc> class TMSVectorIteratorImp;
```

Header File

vectimp.h

Description

Implements a vector iterator that works with any direct, managed, sorted vector of objects of type T.

Public Constructors

TMSVectorIteratorImp::TMSVectorIteratorImp

Public Member Functions

Current

Restart

Operators

++

int

TMSVectorIteratorImp::TMSVectorIteratorImp

TMSVectorIteratorImp class

Form 1

```
TMSVectorIteratorImp( const TMSVectorImp<T,Alloc> &v )
```

Form 2

```
TMSVectorIteratorImp( const TMSVectorImp<T,Alloc> &v, unsigned start,
  unsigned stop )
```

Description

Form 1: Creates an iterator object to traverse TMSVectorImp objects.

Form 2: Creates an iterator object to traverse TMSVectorImp objects. A range can be specified.

TMSVectorIteratorImp::Current

TMSVectorIteratorImp class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TMSVectorIteratorImp::Restart

TMSVectorIteratorImp class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TMSVectorIteratorImp::operator ++

TMSVectorIteratorImp class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMSVectorIteratorImp::operator int

TMSVectorIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TSVectorImp template

Syntax

```
template <class T> class TSVectorImp;
```

Header File

vectimp.h

Description

Implements a sorted vector of objects of type T. TMSVectorImp assumes that T has meaningful copy semantics, a meaningful < operator, and a default constructor.

Public Constructors

TSVectorImp::TSVectorImp

Public Member Functions

Add

AddAt

Count

Detach

Find

GetDelta

Protected Data Members

Count_

Delta

Top

TSVectorImp::TSVectorImp

TSVectorImp class

Form 1

```
TSVectorImp()
```

Form 2

```
TSVectorImp( unsigned sz, unsigned d = 0 )
```

Description

Form 1: Constructs a vector with no entries.

Form 2: Constructs a vector of sz objects, initialized by default to 0.

TSVectorImp::Add

TSVectorImp class

Syntax

```
int Add( const T& t );
```

Description

Adds an object to the vector and increments Count_.

TSVectorImp::AddAt

TSVectorImp class

Syntax

```
int AddAt( const T&, unsigned );
```

Description

Adds an object to the vector at the specified location, and increments Count_.

TSVectorImp::Count

TSVectorImp class

Syntax

```
unsigned Count( ) const;
```

Description

Returns Count_.

TSVectorImp::Detach

TSVectorImp class

Form 1

```
int Detach( unsigned loc );
```

Form 2

```
int Detach( const T& loc );
```

Description

Remove by specifying the object or its index.

TSVectorImp::Find

TSVectorImp class

Syntax

```
virtual unsigned Find( const T& ) const;
```

Description

Finds the specified object and returns the object's index; otherwise returns INT_MAX.

TSVectorImp::GetDelta

TSVectorImp class

Syntax

```
virtual unsigned GetDelta( ) const;
```

Description

Returns Delta.

TSVectorImp::Count_

TSVectorImp class

Syntax

```
unsigned Count_;
```

Description

Maintains the number of objects in the vector.

TSVectorImp::Delta

TSVectorImp class

Syntax

```
unsigned Delta;
```

Description

Specifies the size increment to be used when the vector grows.

TSVectorImp::Top

TSVectorImp class

Syntax

```
virtual unsigned Top( ) const;
```

Description

Returns Count_.

TSVectorIteratorImp template

Syntax

```
template <class T> class TSVectorIteratorImp;
```

Header File

vectimp.h

Description

Implements a vector iterator that works with any direct, sorted vector of objects of type T.

Public Constructors

TSVectorIteratorImp::TSVectorIteratorImp

Public Member Functions

Current

Restart

Operators

++

int

TSVectorIteratorImp::TSVectorIteratorImp

TSVectorIteratorImp class

Form 1

```
TSVectorIteratorImp( const TSVectorImp<T> &v )
```

Form 2

```
TSVectorIteratorImp( const TSVectorImp<T> &v, unsigned start, unsigned stop
  )
```

Description

Form 1: Creates an iterator object to traverse TSVectorImp objects.

Form 2: Creates an iterator object to traverse TSVectorImp objects. A range can be specified.

TSVectorIteratorImp::Current

TSVectorIteratorImp class

Syntax

```
const T& Current();
```

Description

Returns the current object.

TSVectorIteratorImp::Restart

TSVectorIteratorImp class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TSVectorIteratorImp::operator ++

TSVectorIteratorImp class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TSVectorIteratorImp::operator int

TSVectorIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMIVectorImp template

Syntax

```
template <class T, class Alloc> class TMIVectorImp;
```

Header File

vectimp.h

Description

Implements a managed vector of pointers to objects of type T. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Type Definitions

CondFunc

IterFunc

Public Constructors

TMIVectorImp::TMIVectorImp

Public Member Functions

FirstThat

Flush

ForEach

GetDelta

LastThat

Limit

Resize

Top

Zero

Operators

[]

TMIVectorImp::CondFunc

TMIVectorImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMIVectorImp::IterFunc

TMIVectorImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMIVectorImp::TMIVectorImp

TMIVectorImp class

Syntax

```
TMIVectorImp( unsigned sz );
```

Description

Constructs a managed vector of pointers to objects. sz represents the vector size.

TMIVectorImp::FirstThat

TMIVectorImp class

Form 1

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *FirstThat( int ( *)(const T &, void *), void *, unsigned, unsigned )
  const;
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

Form 2: This version allows specifying a range to be searched. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

TMIVectorImp::Flush

TMIVectorImp class

Syntax

```
void Flush( unsigned stop = UINT_MAX, unsigned start = 0 );
```

Description

Flushes the vector without destroying it. The fate of any objects removed depends on the current ownership status and the value of the first argument. A range to be flushed can be specified with the last two arguments.

TMIVectorImp::ForEach

TMIVectorImp class

Form 1

```
void ForEach( IterFunc iter, void *args )
```

Form 2

```
void ForEach( IterFunc iter, void *args, unsigned start, unsigned stop );
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition.

Form 2: This version allows specifying a range.

See Also

TMIVectorImp::FirstThat

TMIVectorImp::GetDelta

TMIVectorImp class

Syntax

```
virtual unsigned GetDelta( ) const;
```

Description

Returns the growth delta for the vector.

TMIVectorImp::LastThat

TMIVectorImp class

Form 1

```
T *LastThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *LastThat( CondFunc cond, void *args, unsigned start, unsigned stop )
  const;
```

Description

Form 1: Returns a pointer to the last object in the vector that satisfies a given condition.

Form 2: This version allows specifying a range.

See Also

TMArrayAsVector::LastThat

TMIVectorImp::Limit

TMIVectorImp class

Syntax

```
unsigned Limit() const;
```

Description

Returns the number of items that the vector can hold.

TMIVectorImp::Resize

TMIVectorImp class

Syntax

```
void Resize( unsigned sz, unsigned offset = 0 );
```

Description

Creates a new vector of size sz. The existing vector is copied to the expanded vector, then deleted. In a vector of pointers the entries are zeroed. In an vector of objects the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the vector has to be extended downward.

TMIVectorImp::Top

TMIVectorImp class

Syntax

```
virtual unsigned Top() const;
```

Description

Returns the index of the current top element. For plain vectors, Top returns Lim; for counted and sorted vectors, Top returns the current insertion point.

TMIVectorImp::Zero

TMIVectorImp class

Syntax

```
virtual void Zero( unsigned, unsigned );
```

Description

Provides for zeroing vector contents within the specified range.

TMIVectorImp::operator [ ]

TMIVectorImp class

Form 1

```
T * & operator [] ( unsigned index )
```

Form 2

```
T * & operator [] ( unsigned index ) const;
```

Description

Returns a reference to the object at index.

TMIVectorIteratorImp template

Syntax

```
template <class T, class Alloc> class TMIVectorIteratorImp;
```

Header File

vectimp.h

Description

Implements a vector iterator that works with an indirect, managed vector.

Public Constructors

TMIVectorIteratorImp::TMIVectorIteratorImp

Public Member Functions

Current

Restart

Operators

++

int

TMIVectorIteratorImp::TMIVectorIteratorImp

TMIVectorIteratorImp class

Form 1

```
TMIVectorIteratorImp( const TMIVectorImp<T,Alloc> &v )
```

Form 2

```
TMIVectorIteratorImp( const TMIVectorImp<T,Alloc> &v, unsigned l, unsigned
  u )
```

Description

Form 1: Creates an iterator object to traverse TMIVectorImp objects.

Form 2: Creates an iterator object to traverse TMIVectorImp objects. A range can be specified.

TMIVectorIteratorImp::Current

TMIVectorIteratorImp class

Syntax

```
T *Current();
```

Description

Returns a pointer to the current object.

TMIVectorIteratorImp::Restart

TMIVectorIteratorImp class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TMIVectorIteratorImp::operator ++

TMIVectorIteratorImp class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMIVectorIteratorImp::operator int

TMIVectorIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TIVectorImp template

Syntax

```
template <class T> class TIVectorImp;
```

Header File

vectimp.h

Description

Implements a vector of pointers to objects of type T. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Public Constructors

TIVectorImp::TIVectorImp

Type Definitions

CondFunc

IterFunc

Public Member Functions

FirstThat

Flush

ForEach

GetDelta

LastThat

Limit

Resize

Top

Zero

Operators

[]

TIVectorImp::CondFunc

TIVectorImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TIVectorImp::IterFunc

TIVectorImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TIVectorImp::TIVectorImp

TIVectorImp class

Syntax

```
TIVectorImp( unsigned sz, unsigned d = 0 )
```

Description

Constructs an indirect vector of sz size, with default initialization of 0.

TIVectorImp::FirstThat

TIVectorImp class

Form 1

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *FirstThat( int ( *)(const T &, void *), void *, unsigned, unsigned )
  const;
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

Form 2: This version allows specifying a range to be searched. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

TIVectorImp::Flush

TIVectorImp class

Syntax

```
void Flush( unsigned stop = UINT_MAX, unsigned start = 0 );
```

Description

Flushes the vector without destroying it. The fate of any objects removed depends on the current ownership status and the value of the first argument. A range to be flushed can be specified with the last two arguments.

TIVectorImp::ForEach

TIVectorImp class

Form 1

```
void ForEach( IterFunc iter, void *args )
```

Form 2

```
void ForEach( IterFunc iter, void *args, unsigned start, unsigned stop );
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition.

Form 2: This version allows specifying a range.

See Also

TIVectorImp::FirstThat

TIVectorImp::GetDelta

TIVectorImp class

Syntax

```
virtual unsigned GetDelta( ) const;
```

Description

Returns the growth delta for the vector.

TIVectorImp::LastThat

TIVectorImp class

Form 1

```
T *LastThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *LastThat( CondFunc cond, void *args, unsigned start, unsigned stop )
  const;
```

Description

Form 1: Returns a pointer to the last object in the vector that satisfies a given condition.

Form 2: This version allows specifying a range.

See Also

TIVectorImp::LastThat

TIVectorImp::Limit

TIVectorImp class

Syntax

```
unsigned Limit() const;
```

Description

Returns the number of items that the vector can hold.

TIVectorImp::Resize

TIVectorImp class

Syntax

```
void Resize( unsigned sz, unsigned offset = 0 );
```

Description

Creates a new vector of size sz. The existing vector is copied to the expanded vector, then deleted. In a vector of pointers the entries are zeroed. In an vector of objects the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the vector has to be extended downward.

TIVectorImp::Top

TIVectorImp class

Syntax

```
virtual unsigned Top() const;
```

Description

Returns the index of the current top element. For plain vectors, Top returns Lim; for counted and sorted vectors, Top returns the current insertion point.

TIVectorImp::Zero

TIVectorImp class

Syntax

```
virtual void Zero( unsigned, unsigned );
```

Description

Provides for zeroing vector contents within the specified range.

TIVectorImp::operator [ ]

TIVectorImp class

Form 1

```
T * & operator [] ( unsigned index )
```

Form 2

```
T * & operator [] ( unsigned index ) const;
```

Description

Returns a reference to the object at index.

TIVectorIteratorImp template

Syntax

```
template <class T> class TIVectorIteratorImp;
```

Header File

vectimp.h

Description

Implements a vector iterator that works with an indirect, managed vector.

Public Constructors

TIVectorIteratorImp::TIVectorIteratorImp

Public Member Functions

Current

Restart

Operators

++

int

TIVectorIteratorImp::TIVectorIteratorImp

TIVectorIteratorImp class

Form 1

```
TIVectorIteratorImp( const TIVectorImp<T> &v )
```

Form 2

```
TIVectorIteratorImp( const TIVectorImp<T> &v, unsigned l, unsigned u )
```

Description

Form 1: Creates an iterator object to traverse TIVectorImp objects.

Form 2: Creates an iterator object to traverse TIVectorImp objects. A range can be specified.

TIVectorIteratorImp::Current

TIVectorIteratorImp class

Syntax

```
T *Current();
```

Description

Returns a pointer to the current object.

TIVectorIteratorImp::Restart

TIVectorIteratorImp class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TIVectorIteratorImp::operator ++

TIVectorIteratorImp class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TIVectorIteratorImp::operator int

TIVectorIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMICVectorImp template

Syntax

```
template <class T, class Alloc> class TMICVectorImp;
```

Header File

vectimp.h

Description

Implements a managed, counted vector of pointers to objects of type T. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Public Constructors

TMICVectorImp::TMICVectorImp

Public Member Functions

Add

Find

Protected Data Members

Find

TMICVectorImp::TMICVectorImp

TMICVectorImp class

Syntax

```
TMICVectorImp( unsigned sz, unsigned d = 0 )
```

Description

Default constructor.

TMICVectorImp::Add

TMICVectorImp class

Syntax

```
int Add( T *t );
```

Description

Adds an object to the vector.

TMICVectorImp::Find

TMICVectorImp class

Form 1

```
unsigned Find( T *t ) const;
```

Form 2

```
virtual unsigned Find( void * ) const;  // PROTECTED
```

Description

Form 1: Finds the specified object pointer, and returns its index.

Form 2: Finds the specified pointer and returns its index.

TMICVectorIteratorImp template

Syntax

```
template <class T, class Alloc> class TMICVectorIteratorImp;
```

Header File

vectimp.h

Description

Implements a vector iterator that works with an indirect, managed, counted vector.

Public Constructors

TMICVectorIteratorImp::TMICVectorIteratorImp

Public Member Functions

Current

Restart

Operators

++

int

TMICVectorIteratorImp::TMICVectorIteratorImp

TMICVectorIteratorImp class

Form 1

```
TMICVectorIteratorImp( const TMICVectorImp<T,Alloc> &v )
```

Form 2

```
TMICVectorIteratorImp( const TMICVectorImp<T,Alloc> &v, unsigned l,
  unsigned u )
```

Description

Form 1: Creates an iterator object to traverse TMICVectorImp objects.

Form 2: Creates an iterator object to traverse TMICVectorImp objects. A range can be specified.

TMICVectorIteratorImp::Current

TMICVectorIteratorImp class

Syntax

```
T *Current();
```

Description

Returns a pointer to the current object.

TMICVectorIteratorImp::Restart

TMICVectorIteratorImp class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TMICVectorIteratorImp::operator ++

TMICVectorIteratorImp class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMICVectorIteratorImp::operator int

TMICVectorIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TICVectorImp template

Syntax

```
template <class T> class TICVectorImp;
```

Header File

vectimp.h

Description

Implements a counted vector of pointers to objects of type T. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Public Constructors

TICVectorImp::TICVectorImp

Type Definitions

CondFunc

IterFunc

Public Member Functions

FirstThat

Flush

ForEach

GetDelta

LastThat

Resize

Top

Zero

Operators

[]

TICVectorImp::CondFunc

TICVectorImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TICVectorImp::IterFunc

TICVectorImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TICVectorImp::TICVectorImp

TICVectorImp class

Syntax

```
TICVectorImp( unsigned sz, unsigned d = 0 )
```

Description

Constructs a counted vector of pointers to objects. sz represents the vector size. d represents the initialization value.

TICVectorImp::FirstThat

TICVectorImp class

Form 1

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *FirstThat( int ( *)(const T &, void *), void *, unsigned, unsigned )
  const;
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

Form 2: This version allows specifying a range to be searched. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

TICVectorImp::Flush

TICVectorImp class

Syntax

```
void Flush( unsigned stop = UINT_MAX, unsigned start = 0 );
```

Description

Flushes the vector without destroying it. The fate of any objects removed depends on the current ownership status and the value of the first argument. A range to be flushed can be specified with the last two arguments.

TICVectorImp::ForEach

TICVectorImp class

Form 1

```
void ForEach( IterFunc iter, void *args )
```

Form 2

```
void ForEach( IterFunc iter, void *args, unsigned start, unsigned stop );
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition.

Form 2: This version allows specifying a range.

See Also

TICVectorImp::FirstThat

TICVectorImp::GetDelta

TICVectorImp class

Syntax

```
virtual unsigned GetDelta( ) const;
```

Description

Returns the growth delta for the vector.

TICVectorImp::LastThat

TICVectorImp class

Form 1

```
T *LastThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *LastThat( CondFunc cond, void *args, unsigned start, unsigned stop )
  const;
```

Description

Form 1: Returns a pointer to the last object in the vector that satisfies a given condition.

Form 2: This version allows specifying a range.

See Also

TICVectorImp::LastThat

TICVectorImp::Limit

TICVectorImp class

Syntax

```
unsigned Limit() const;
```

Description

Returns the number of items that the vector can hold.

TICVectorImp::Resize

TICVectorImp class

Syntax

```
void Resize( unsigned sz, unsigned offset = 0 );
```

Description

Creates a new vector of size sz. The existing vector is copied to the expanded vector, then deleted. In a vector of pointers the entries are zeroed. In an vector of objects the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the vector has to be extended downward.

TICVectorImp::Top

TICVectorImp class

Syntax

```
virtual unsigned Top() const;
```

Description

Returns the index of the current top element. For plain vectors, Top returns Lim; for counted and sorted vectors, Top returns the current insertion point.

TICVectorImp::Zero

TICVectorImp class

Syntax

```
virtual void Zero( unsigned, unsigned );
```

Description

Provides for zeroing vector contents within the specified range.

TICVectorImp::operator [ ]

TICVectorImp class

Form 1

```
T * & operator [] ( unsigned index )
```

Form 2

```
T * & operator [] ( unsigned index ) const;
```

Description

Returns a reference to the object at index.

TICVectorIteratorImp template

Syntax

```
template <class T> class TICVectorIteratorImp;
```

Header File

vectimp.h

Description

Implements a vector iterator that works with an indirect, managed, counted vector.

Public Constructors

TICVectorIteratorImp::TICVectorIteratorImp

Public Member Functions

Current

Restart

Operators

++

int

TICVectorIteratorImp::TICVectorIteratorImp

TICVectorIteratorImp class

Form 1

```
TICVectorIteratorImp( const TICVectorImp<T> &v )
```

Form 2

```
TICVectorIteratorImp( const TICVectorImp<T> &v, unsigned l, unsigned u )
```

Description

Form 1: Creates an iterator object to traverse TICVectorImp objects.

Form 2: Creates an iterator object to traverse TICVectorImp objects. A range can be specified.

TICVectorIteratorImp::Current

TICVectorIteratorImp class

Syntax

```
T *Current();
```

Description

Returns a pointer to the current object.

TICVectorIteratorImp::Restart

TICVectorIteratorImp class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TICVectorIteratorImp::operator ++

TICVectorIteratorImp class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TICVectorIteratorImp::operator int

TICVectorIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TMISVectorImp template

Syntax

```
template <class T, class Alloc> class TMISVectorImp;
```

Header File

vectimp.h

Description

Implements a managed, sorted vector of pointers to objects of type T. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Public Constructors

TMISVectorImp::TMISVectorImp

Type Definitions

CondFunc

IterFunc

Public Member Functions

Add

Find

FirstThat

Flush

ForEach

GetDelta

LastThat

Resize

Top

Zero

Operators

[]

TMISVectorImp::CondFunc

TMISVectorImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TMISVectorImp::IterFunc

TMISVectorImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TMISVectorImp::TMISVectorImp

TMISVectorImp class

Syntax

```
TMISVectorImp( unsigned sz, unsigned d = 0 );
```

Description

Constructs a managed, sorted vector of pointers to objects. sz represents the vector size. d represents the initialization value.

TMISVectorImp::Add

TMISVectorImp class

Syntax

```
int Add( T *t );
```

Description

Adds an object to the vector.

TMISVectorImp::Find

TMISVectorImp class

Form 1

```
unsigned Find( T *t ) const;
```

Form 2

```
virtual unsigned Find( void * ) const;
```

Description

Form 1: Finds the specified object pointer, and returns its index.

Form 2: Finds the specified pointer and returns its index.

TMISVectorImp::FirstThat

TMISVectorImp class

Form 1

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *FirstThat( int ( *)(const T &, void *), void *, unsigned, unsigned )
  const;
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

Form 2: This version allows specifying a range to be searched. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

TMISVectorImp::Flush

TMISVectorImp class

Syntax

```
void Flush( unsigned stop = UINT_MAX, unsigned start = 0 );
```

Description

Flushes the vector without destroying it. The fate of any objects removed depends on the current ownership status and the value of the first argument. A range to be flushed can be specified with the last two arguments.

TMISVectorImp::ForEach

TMISVectorImp class

Form 1

```
void ForEach( IterFunc iter, void *args )
```

Form 2

```
void ForEach( IterFunc iter, void *args, unsigned start, unsigned stop );
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition.

Form 2: This version allows specifying a range.

See Also

TMISVectorImp::FirstThat

TMISVectorImp::GetDelta

TMISVectorImp class

Syntax

```
virtual unsigned GetDelta( ) const;
```

Description

Returns the growth delta for the vector.

TMISVectorImp::LastThat

TMISVectorImp class

Form 1

```
T *LastThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *LastThat( CondFunc cond, void *args, unsigned start, unsigned stop )
  const;
```

Description

Form 1: Returns a pointer to the last object in the vector that satisfies a given condition.

Form 2: This version allows specifying a range.

See Also

TMISVectorImp::LastThat

TMISVectorImp::Limit

TMISVectorImp class

Syntax

```
unsigned Limit() const;
```

Description

Returns the number of items that the vector can hold.

TMISVectorImp::Resize

TMISVectorImp class

Syntax

```
void Resize( unsigned sz, unsigned offset = 0 );
```

Description

Creates a new vector of size sz. The existing vector is copied to the expanded vector, then deleted. In a vector of pointers the entries are zeroed. In an vector of objects the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the vector has to be extended downward.

TMISVectorImp::Top

TMISVectorImp class

Syntax

```
virtual unsigned Top() const;
```

Description

Returns the index of the current top element. For plain vectors, Top returns Lim; for counted and sorted vectors, Top returns the current insertion point.

TMISVectorImp::Zero

TMISVectorImp class

Syntax

```
virtual void Zero( unsigned, unsigned );
```

Description

Provides for zeroing vector contents within the specified range.

TMISVectorImp::operator [ ]

TMISVectorImp class

Form 1

```
T * & operator [] ( unsigned index )
```

Form 2

```
T * & operator [] ( unsigned index ) const;
```

Description

Returns a reference to the object at index.

TMISVectorIteratorImp template

Syntax

```
template <class T, class Alloc> class TMISVectorIteratorImp;
```

Header File

vectimp.h

Description

Implements a vector iterator that works with an indirect, managed, sorted vector.

Public Constructors

TMISVectorIteratorImp::TMISVectorIteratorImp

Public Member Functions

Current

Restart

Operators

++

int

TMISVectorIteratorImp::TMISVectorIteratorImp

TMISVectorIteratorImp class

Form 1

```
TMISVectorIteratorImp( const TMISVectorImp<T,Alloc> &v )
```

Form 1

```
TMISVectorIteratorImp( const TMISVectorImp<T,Alloc> &v, unsigned l,
  unsigned u )
```

Description

Form 1: Creates an iterator object to traverse TMISVectorImp objects.

Form 2: Creates an iterator object to traverse TMISVectorImp objects. A range can be specified.

TMISVectorIteratorImp::Current

TMISVectorIteratorImp class

Syntax

```
T *Current();
```

Description

Returns a pointer to the current object.

TMISVectorIteratorImp::Restart

TMISVectorIteratorImp class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TMISVectorIteratorImp::operator ++

TMISVectorIteratorImp class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TMISVectorIteratorImp::operator int

TMISVectorIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

TISVectorImp template

Syntax

```
template <class T> class TISVectorImp;
```

Header File

vectimp.h

Description

Implements a sorted vector of pointers to objects of type T. Since pointers always have meaningful copy semantics, this class can handle any type of object.

Public Constructors

TISVectorImp::TISVectorImp

Type Definitions

CondFunc

IterFunc

Public Member Functions

Add

Find

FirstThat

Flush

ForEach

GetDelta

LastThat

Resize

Top

Zero

Operators

[]

TISVectorImp::CondFunc

TISVectorImp class

Syntax

```
typedef int ( *CondFunc)(const T &, void *);
```

Description

Function type used as a parameter to FirstThat and LastThat member functions.

TISVectorImp::IterFunc

TISVectorImp class

Syntax

```
typedef void ( *IterFunc)(T &, void *);
```

Description

Function type used as a parameter to ForEach member function.

TISVectorImp::TISVectorImp

TISVectorImp class

Syntax

```
TISVectorImp( unsigned sz, unsigned d = 0 )
```

Description

Constructs a managed, sorted vector of pointers to objects. sz represents the vector size. d represents the initialization value.

TISVectorImp::Add

TISVectorImp class

Syntax

```
int Add( T *t );
```

Description

Adds an object to the vector.

TISVectorImp::Find

TISVectorImp class

Form 1

```
unsigned Find( T *t ) const;
```

Form 2

```
virtual unsigned Find( void * ) const;
```

Description

Form 1: Finds the specified object pointer, and returns its index.

Form 2: Finds the specified pointer and returns its index.

TISVectorImp::FirstThat

TISVectorImp class

Form 1

```
T *FirstThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *FirstThat( int ( *)(const T &, void *), void *, unsigned, unsigned )
  const;
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

Form 2: This version allows specifying a range to be searched. You supply a test-function pointer cond that returns true for a certain condition. You can pass arbitrary arguments via args. Returns 0 if no object in the vector meets the condition.

TISVectorImp::Flush

TISVectorImp class

Syntax

```
void Flush( unsigned stop = UINT_MAX, unsigned start = 0 );
```

Description

Flushes the vector without destroying it. The fate of any objects removed depends on the current ownership status and the value of the first argument. A range to be flushed can be specified with the last two arguments.

TISVectorImp::ForEach

TISVectorImp class

Form 1

```
void ForEach( IterFunc iter, void *args )
```

Form 2

```
void ForEach( IterFunc iter, void *args, unsigned start, unsigned stop );
```

Description

Form 1: Returns a pointer to the first object in the vector that satisfies a given condition.

Form 2: This version allows specifying a range.

See Also

TISVectorImp::FirstThat

TISVectorImp::GetDelta

TISVectorImp class

Syntax

```
virtual unsigned GetDelta( ) const;
```

Description

Returns the growth delta for the vector.

TISVectorImp::LastThat

TISVectorImp class

Form 1

```
T *LastThat( CondFunc cond, void *args ) const;
```

Form 2

```
T *LastThat( CondFunc cond, void *args, unsigned start, unsigned stop )
  const;
```

Description

Form 1: Returns a pointer to the last object in the vector that satisfies a given condition.

Form 2: This version allows specifying a range.

See Also

TISVectorImp::LastThat

TISVectorImp::Limit

TISVectorImp class

Syntax

```
unsigned Limit() const;
```

Description

Returns the number of items that the vector can hold.

TISVectorImp::Resize

TISVectorImp class

Syntax

```
void Resize( unsigned sz, unsigned offset = 0 );
```

Description

Creates a new vector of size sz. The existing vector is copied to the expanded vector, then deleted. In a vector of pointers the entries are zeroed. In an vector of objects the default constructor is invoked for each unused element. offset is the location in the new vector where the first element of the old vector should be copied. This is needed when the vector has to be extended downward.

TISVectorImp::Top

TISVectorImp class

Syntax

```
virtual unsigned Top() const;
```

Description

Returns the index of the current top element. For plain vectors, Top returns Lim; for counted and sorted vectors, Top returns the current insertion point.

TISVectorImp::Zero

TISVectorImp class

Syntax

```
virtual void Zero( unsigned, unsigned );
```

Description

Provides for zeroing vector contents within the specified range.

TISVectorImp::operator [ ]

TISVectorImp class

Form 1

```
T * & operator [] ( unsigned index )
```

Form 2

```
T * & operator [] ( unsigned index ) const;
```

Description

Returns a reference to the object at index.

TISVectorIteratorImp template

Syntax

```
template <class T> class TISVectorIteratorImp;
```

Header File

vectimp.h

Description

Implements a vector iterator that works with an indirect, managed, sorted vector.

members.

Public Constructors

TISVectorIteratorImp::TISVectorIteratorImp

Public Member Functions

Current

Restart

Operators

++

int

TISVectorIteratorImp::TISVectorIteratorImp

TISVectorIteratorImp class

Form 1

```
TISVectorIteratorImp( const TISVectorImp<T> &v )
```

Form 2

```
TISVectorIteratorImp( const TISVectorImp<T> &v, unsigned l, unsigned u )
```

Description

Form 1: Creates an iterator object to traverse TISVectorImp objects.

Form 2: Creates an iterator object to traverse TISVectorImp objects. A range can be specified.

TISVectorIteratorImp::Current

TISVectorIteratorImp class

Syntax

```
T *Current();
```

Description

Returns a pointer to the current object.

TISVectorIteratorImp::Restart

TISVectorIteratorImp class

Form 1

```
void Restart();
```

Form 2

```
void Restart( unsigned start, unsigned stop );
```

Description

Form 1: Restarts iteration over the whole vector.

Form 2: Restarts iteration over the given range.

TISVectorIteratorImp::operator ++

TISVectorIteratorImp class

Form 1

```
const T& operator ++(int);
```

Form 2

```
const T& operator ++();
```

Description

Form 1: Moves to the next object, and returns the object that was current before the move (post-increment).

Form 2: Moves to the next object, and returns the object that was current after the move (pre-increment).

TISVectorIteratorImp::operator int

TISVectorIteratorImp class

Syntax

```
operator int();
```

Description

Converts the iterator to an integer value for testing if objects remain in the iterator. The iterator converts to 0 if nothing remains in the iterator.

iostream Classes (C++)

Hierarchy

The stream class library in C++ consists of several classes distributed in two separate hierarchical trees. This reference presents some of most useful details of these classes, in alphabetical order.

To get Help about an iostream class, choose it from the list below:

conbuf

constream

filebuf

fstream

fstreambase

ifstream

ios

iostream_withassign

iostream

istream_withassign

istream

istrstream

ofstream

ostream_withassign

ostream

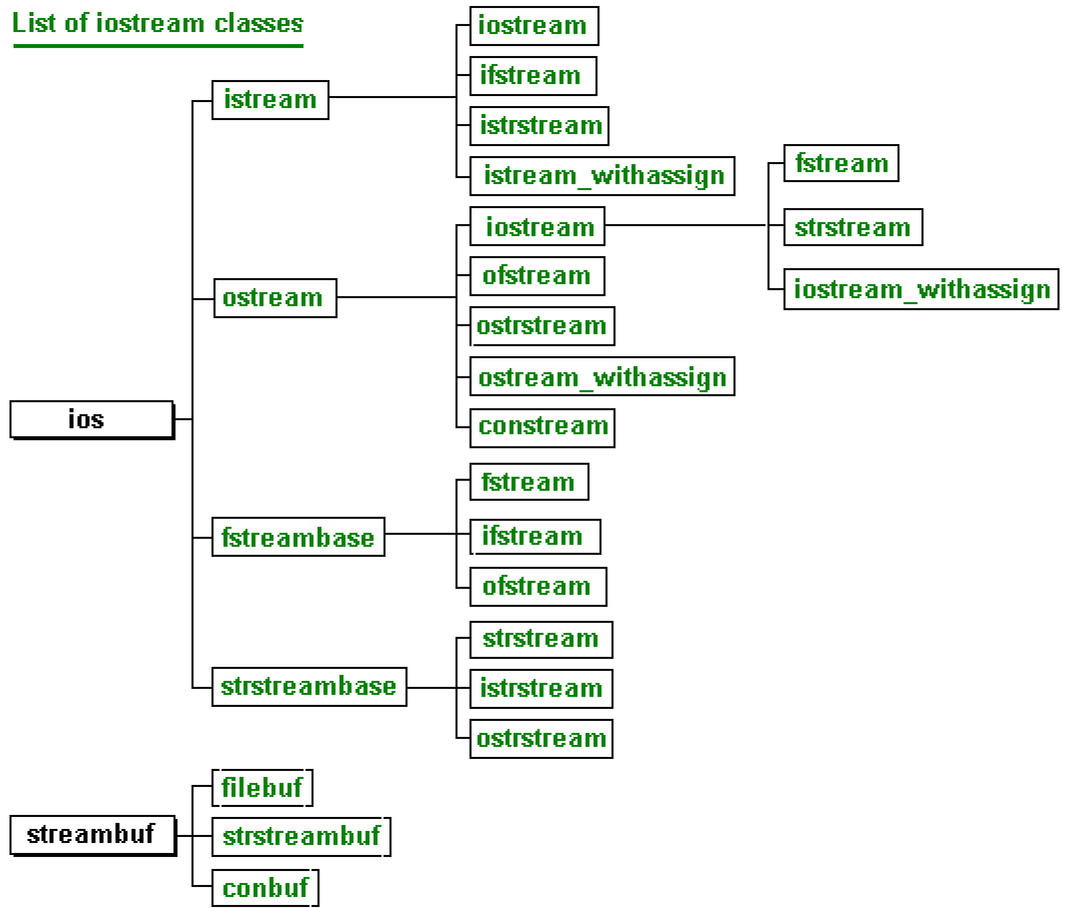
ostrstream

streambuf

strstream

strstreambase

strstreambuf



List of iostream classes
ios
istream
iostream
ifstream
istrstream
istream_withassign
ostream
iostream
fstream
strstream
iostream_withassign
ofstream
ostrstream
ostream_withassign
constream
fstreambase
fstream
ifstream
ofstream
strstreambase
strstream
istrstream
ostrstream
streambuf
filebuf
strstreambuf
conbuf

constrea.h

See Also   Header Files

The constrea.h header file defines the class constream, which writes output to the screen using the iostream interface.

Includes

CONIO.H

IOMANIP.H

IOSTREAM.H

Classes

conbuf

constream

fstream.h

See Also Header Files

Declares the C++ stream classes that support file input and output.

Replaces the older, now outdated stdiostr.h.

Includes

IOSTREAM.H

Classes

To get more Help on the classes in the fstream.h header file (and their attendant member functions and data members), choose one of these Help links:

filebuf

fstream

fstreambase

ifstream

ofstream

iostream.h

See Also Header Files

Declares the basic C++ streams (I/O) routines.

Includes

MEM.H

Classes

Choose one of the following topics for more information on the classes in the iostream.h header file:

ios

iostream

iostream_withassign

istream

istream_withassign

ostream

ostream_withassign

streambuf

See Also

I/O Stream Classes

Precompiled Headers

strstrea.h

See Also Header Files

Declares the C++ stream classes for use with byte arrays in memory.

Includes

IOSTREAM.H

Classes

To get more Help on the classes in the strstrea.h header file (and their attendant member functions and data members), choose one of these Help links:

istrstream

ostrstream

strstream

strstreambase

strstreambuf

conbuf class

See Also   Hierarchy

Header File
constrea.h

Description
Specializes streambuf to handle console output.

Note: *conbuf* is available only for console-mode applications.

Public Constructor
conbuf::conbuf

Public Member Functions

clreol

clrscr

delline

gotoxy

highvideo

insline

lowvideo

normvideo

overflow

setcursortype

textattr

textbackground

textcolor

textmode

wherex

wherey

window

See Also

constream class

conbuf::conbuf

conbuf class

Syntax

```
conbuf()
```

Description

Makes an unattached *conbuf*.

conbuf::clreol

conbuf class

Syntax

```
void clreol()
```

Description

Clears to end of line in text window.

conbuf::clrscr

conbuf class

Syntax

```
void clrscr()
```

Description

Clears the defined screen.

conbuf::delline

conbuf class

Syntax

```
void delline()
```

Description

Deletes a line in the window.

conbuf::gotoxy

conbuf class

Syntax

```
void gotoxy(int x, int y)
```

Description

Positions the cursor in the window at the specified location.

conbuf::highvideo

conbuf class

Syntax

```
void highvideo()
```

Description

Selects high-intensity characters.

conbuf::insline

conbuf class

Syntax

```
void insline()
```

Description

Inserts a blank line.

conbuf::lowvideo

conbuf class

Syntax

```
void lowvideo()
```

Description

Selects low-intensity characters.

conbuf::normvideo

conbuf class

Syntax

```
void normvideo()
```

Description

Selects normal-intensity characters.

conbuf::overflow

conbuf class

Syntax

```
virtual int overflow( int = EOF )
```

Description

Flushes the *conbuf* to its destination.

conbuf::setcursortype

conbuf class

Syntax

```
void setcursortype(int cur_type)
```

Description

Selects the cursor appearance.

conbuf::textattr

conbuf class

Syntax

```
void textattr(int newattribute)
```

Description

Selects the cursor appearance.

conbuf::textbackground

conbuf class

Syntax

```
void textbackground(int newcolor)
```

Description

Selects the text background color.

conbuf::textcolor

conbuf class

Syntax

```
void textcolor( int newcolor)
```

Description

Selects character color in text mode.

conbuf::textmode

conbuf class

Syntax

```
static void textmode(int newmode)
```

Description

Puts the screen in text mode.

conbuf::wherex

conbuf class

Syntax

```
int wherex()
```

Description

Gets the horizontal cursor position.

conbuf::wherey

conbuf class

Syntax

```
int wherey()
```

Description

Gets the vertical cursor position.

conbuf::window

conbuf class

Syntax

```
void window(int left, int top, int right, int bottom)
```

Description

Defines the active window.

streambuf — conbuf

constream class

See Also Hierarchy

Header File
constrea.h

Description
Provides console output streams. This class is derived from ostream.

Note: constream is available only for console-mode applications.

Constructor
constream::constream

Public Member Functions
clrscr
rdbuf
textmode
window

See Also
conbuf class

constream::constream

constream class

Syntax

```
constream()
```

Description

Provides an unattached output stream to the console.

constream::clrscr

constream class

Syntax

```
void clrscr()
```

Description

Clears the screen.

constream::rdbuf

constream class

Syntax

```
conbuf *rdbuf()
```

Description

Returns a pointer to this constream's assigned *conbuf*.

constream::textmode

constream class

Syntax

```
void textmode(int newmode)
```

Description

Puts the screen in text mode.

constream::window

constream class

Syntax

```
void window(int left, int top, int right, int bottom)
```

Description

Defines the active window.

ostream — constream

fstreambase class

Hierarchy

Header File
fstream.h

Description
fstreambase provides access to filebuf functions not accessible through *ios::bp* to *fstreambase* and its derived classes.

If a member function of *filebuf* is not a virtual member of *filebuf*'s base class (streambuf), it is not accessible. For example: *attach*, *open* and *close* are not accessible.

The constructors of *fstreambase* initialize the *ios::bp* data member to point to the *filebuf*.

Constructors
fstreambase::fstreambase

Member Functions
attach
close
open
rdbuf
setbuf

fstreambase::fstreambase

fstreambase class

Form 1

```
fstreambase();
```

Form 2

```
fstreambase(const char *name, int mode, int = filebuf::openprot);
```

Form 3

```
fstreambase(int fd);
```

Form 4

```
fstreambase(int fd, char *buf, int len);
```

Description

Form 1: Makes an *fstreambase* that is not attached to a file.

Form 2: Makes an *fstreambase*, opens a file specified by *name* in the mode specified by *mode*, and connects to it.

Form 3: Makes an *fstreambase* and connects to an open file descriptor specified by *fd*.

Form 4: Makes an *fstreambase* connected to an open file descriptor specified by *fd* and uses a buffer specified by *buf* with a size specified by *len*.

fstreambase::attach

fstreambase class

Syntax

```
void attach(int);
```

Description

Connects to an open file descriptor.

fstreambase::close

fstreambase class

Syntax

```
void close();
```

Description

Closes the associated filebuf and file.

fstreambase::open

fstreambase class

Syntax

```
void open(const char *name, int mode,int prot=filebuf::openprot);
```

Description

Opens a file for the specific class object.

The *mode* parameter can be set using the *open_mode* enumeration defined in class ios.

Class	***mode*** **parameter**
fstream	ios::in
ofstream	ios::out

prot corresponds to the DOS access permission, and it is used unless ios::nocreate is specified in *mode*. The default parameter is set to read and write permission.

fstreambase::rdbuf

fstreambase class

Syntax

```
filebuf* rdbuf();
```

Description

Returns the buffer used.

fstreambase::setbuf

fstreambase class

Syntax

```
void setbuf(char*, int);
```

Description

Assigns a user-specified buffer to the filebuf.

ios
fstreambase
fstream
ifstream
ofstream

fstream class

Hierarchy Example

Header File
fstream.h

Description
Provides for simultaneous input and output on a filebuf.

Input and output are initiated using the functions of the base classes istream and ostream. For example, *fstream* can use the function *istream::getline()* to extract characters from the file.

Constructors
fstream::fstream

Member Functions
open

rdbuf

fstream::fstream

fstream class

Form 1

```
fstream();
```

Form 2

```
fstream(const char *name, int mode = ios::in, int prot =
  filebuf::openprot);
```

Form 3

```
fstream(int fd);
```

Form 4

```
fstream(int fd, char *buf, int n);
```

Description

Form 1: Makes an *fstream* that is not attached to a file.

Form 2: Makes an *fstream*, opens a file with access specified by *mode*, and connects to it.

Form 3: Makes an *fstream*, and connects to an open file descriptor specified by *fd*.

Form 4: Makes an *fstream* specified by the file descriptor *fd*, and uses *buf* as the storage area. The size of *buf* is sufficient to store *n* bytes. If *buf* is NULL or *n* is non-positive, the *fstream* is unbuffered.

fstream::open

fstream class

Syntax

```
void open(const char *name, int mode,int prot=filebuf::openprot);
```

Description

Opens a file for the specific class object.

The mode parameter can be set using the *open_mode* enumeration defined in class ios.

Class	***mode*** **parameter**
fstream	ios::in
ofstream	ios::out

prot corresponds to the DOS access permission, and it is used unless *ios::nocreate* is specified in *mode*. The default parameter is set to read and write permission.

fstream::rdbuf

fstream class

Syntax

```
filebuf* rdbuf();
```

Description

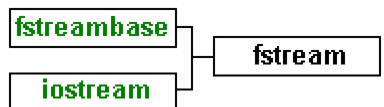
Returns the buffer used.

fstream Example

```
// Create a file stream.
#include <fstream.h>

void main(void) {
   char ch;
   const char *name = "_junk_.$$$";
   int mode = ios::in | ios::app;

   fstream outf( name, mode );  // Output file stream.
   cout << "Ready for input: Use Control-Z to end.";
   while ( cin.get(ch) )
      outf.put( ch );
   }
```


fstreambase
iostream
fstream

filebuf class

Hierarchy   Example

Header File
fstream.h

Description
Specializes streambuf to handle files for input and output of characters. (Since streambuf does not provide streams for input or output, the derived classes of streambuf must do so.)

The I/O functions of classes istream and ostream make calls to the functions of filebuf to do the actual insertion or extraction on the streams. This occurs if the 'bp' (pointer to streambuf) fdata member of class ios has been assigned a pointer to class filebuf.

Constructors
filebuf::filebuf

Data Members
openprot

Member Functions
attach

close

fd

is_open

open

overflow

seekoff

setbuf

sync

underflow

filebuf::filebuf

filebuf class

Form 1

```
filebuf::filebuf();
```

Form 2

```
filebuf::filebuf(int fd);
```

Form 3

```
filebuf::filebuf(int fd, char *, int n);
```

Description

Form 1: Makes a *filebuf* that isn't attached to a file.

Form 2: Makes a *filebuf* attached to a file by file descriptor *fd*.

Form 3: Makes a *filebuf* attached to a file specified by the file descriptor *fd*, and uses *buf* as the storage area. The size of *buf* is sufficient to store *n* bytes. If *buf* is NULL or *n* is non-positive, the *filebuf* is unbuffered.

filebuf::openprot

filebuf class

Syntax

```
static const int openprot;
```

Description

The *openprot* data member is the default file protection. It sets the permissions to read and write.

filebuf::attach

filebuf class

Syntax

```
filebuf* attach(int);
```

Description

Attaches this closed *filebuf* to opened file descriptor.

filebuf::close

filebuf class

Syntax

```
filebuf* close();
```

Description

Flushes and closes the file. Returns 0 on error.

filebuf::fd

filebuf class

Syntax

```
int fd();
```

Description

The *fd* member function returns the file descriptor or EOF.

filebuf::is_open

filebuf class

Syntax

```
int is_open();
```

Description

The is_open member function returns nonzero if the file is open.

See Also

filebuf class

filebuf::open

filebuf class

Syntax

```
filebuf* open(const char *name, int mode, int prot = filebuf::openprot);
```

Description

Opens a file for the specific class object. The *mode* parameter can be set using the *open_mode* enumeration defined in class ios.

Class	***mode*** **Parameter**
fstream	ios::in
ofstream	ios::out

prot corresponds to the DOS access permission, and it is used unless ios::nocreate is specified in *mode*. The default parameter is set to read and write permission.

filebuf::overflow

filebuf class

Syntax

```
virtual int overflow(int = EOF);
```

Description

Flushes a buffer to its destination. Every derived class should define the actions to be taken.

filebuf::seekoff

filebuf class

Syntax

```
virtual streampos seekoff(streamoff offset, ios::seek_dir, int mode);
```

Description

Moves the file pointer relative to the current position in the direction of *seek_dir*.

seek_dir is set using the *seek_dir* enumeration definition in class ios.

ios::beg	seek from beginning of file
ios::cur	seek from current location
ios::end	seek from end of file

Since the long can be a negative value, seeking can occur "backward" in the file from the end or current location.

mode specifies the type of move in the get or put area of the internal buffer by using *ios::in*, *ios::out*, or both.

When this virtual member function is redefined in a derived class, it could be seeking into the stream and not streambuf's internal buffer.

filebuf::setbuf

filebuf class

Syntax

```
virtual streambuf* setbuf(char*, int);
```

Description

Specifies a buffer of a specified size for the class object. When used as a strstreambuf and the function is overloaded, the first argument is not meaningful and should be set to zero.

filebuf::sync

filebuf class

Syntax

```
virtual int sync();
```

Description

Syncronizes the internal data structures and the external stream representation.

filebuf::underflow

filebuf class

Form 1

```
virtual int underflow();
```

Description

Makes input available. This is called when no more data exists in the input buffer. Every derived class should define the actions to be taken.

streambuf — filebuf

filebuf class Example

filebuf class

```
// OPERATIONS WITH filebuf

#include <fstream.h>
const char *OUTF = "_junk_.$$$";

int main(void) {
   filebuf fbuf;  // Unattached file buffer.

   fbuf.open(OUTF, ios::out);
   if (!fbuf.is_open() ) {
       cerr << "Error opening input file " << OUTF;
       return(-1);
       }
   return(0);
   }


//**************************************************************
// OPERATIONS WITH filebuf(fd)
#include <fstream.h>
#include <io.h>
#include <fcntl.h>

int main(void) {
   const char *filename = "_junk_.$$$";
   int fd;  // The file descriptor.

   fd = open(filename, O_RDWR | O_CREAT); // Open file; get descriptor

   // Make a filebuf; use file descriptor.
   filebuf iofile(fd);

   if (!iofile.is_open()) {
       cerr << "The filebuf is not open.";
      return(1);
      }

   // Do things with filebuf.
   iofile.sputn("Borland International", 21);

   return(0);
   }
```

ofstream class

Hierarchy

Header File
fstream.h

Description
Provides an output stream to extract from a file using a *filebuf*.

Constructors
ofstream::ofstream

Member Functions
open

rdbuf

ofstream::ofstream

ofstream class

Form 1
```
ofstream();
```

Form 2
```
ofstream(const char *name, int mode = ios::out, int prot =
  filebuf::openprot);
```

Form 3
```
ofstream(int fd);
```

Form 4
```
ofstream(int fd, char *buf, int len);
```

Description

Form 1: Makes an *ofstream* that is not attached to a file.

Form 2: Makes an *ofstream*, opens a file for writing, and connects to it.

Form 3: Makes an *ofstream*, connects to an open file descriptor specified by *fd*.

Form 4: Makes an *ofstream* connected to an open file descriptor specified by *fd*. The buffer specified by *buf* of *len* is used by the *ofstream*.

ofstream::open

ofstream class

Syntax

```
void open(const char *name, int mode,int prot=filebuf::openprot);
```

Description

Opens a file for the specific class object.

The *mode* parameter can be set using the *open_mode* enumeration defined in class ios.

Class	***mode*** **parameter**
fstream	ios::in
ofstream	ios::out

prot corresponds to the DOS access permission, and it is used unless ios::nocreate is specified in *mode*. The default parameter is set to read and write permission.

ofstream::rdbuf

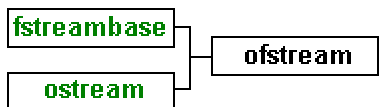
ofstream class

Syntax

```
filebuf* rdbuf();
```

Description

Returns the buffer used.


fstreambase
ostream
ofstream

ifstream class

Hierarchy

Header File
fstream.h

Description
Provides an input stream to input from a file using a filebuf.

Constructors
ifstream::ifstream

Member Functions
open

rdbuf

ifstream::ifstream

ifstream class

Form 1

```
ifstream();
```

Form 2

```
ifstream(const char *name, int mode = ios::in, int = filebuf::openprot);
```

Form 3

```
ifstream(int fd);
```

Form 4

```
ifstream(int fd, char *buf, int buf_len);
```

Description

Form 1: Makes an *ifstream* that is not attached to a file.

Form 2: Makes an *ifstream*, opens an input file in protected mode, and connects to it. The existing file contents are preserved; new writes are appended. By default, a file is not created if it does not exist.

Form 3: Makes an *ifstream*, connects to an open file descriptor *fd*.

Form 4: Makes an *ifstream* connected to an open file specified by its descriptor, *fd*. The *ifstream* uses the buffer specified by *buf* of length *buf_len*.

ifstream::open

ifstream class

Syntax

```
void open(const char *name, int mode,int prot=filebuf::openprot);
```

Description

Opens a file for the specific class object.

The mode parameter can be set using the open_mode enumeration defined in class ios.

Class	***mode*** **parameter**
fstream	ios::in
ofstream	ios::out

prot corresponds to the DOS access permission, and it is used unless ios::nocreate is specified in mode. The default parameter is set to read and write permission.

ifstream::rdbuf

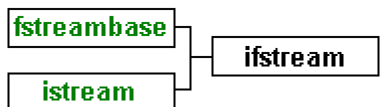
ifstream class

Syntax

```
filebuf* rdbuf();
```

Description

Returns the buffer used.


fstreambase
istream
ifstream

iostream_withassign class

See Also   Hierarchy

Header File
iostream.h

Description
This class is an iostream that overloads the = operator which allows you to reassign *ios::bp* to a different derived class of streambuf.

Constructors
iostream_withassign::iostream_withassign

Member Functions
None (although the **=** operator is overloaded).

See Also
Overloading Operators

iostream_withassign::iostream_withassign

iostream_withassign class

Syntax

```
iostream_withassign();
```

Description

Null constructor (calls the default constructor for iostream).

iostream — iostream_withassign

iostream class

Hierarchy

Header File
iostream.h

Description
This class, derived from istream and ostream, is a mixture of its base classes, allowing both input and output on a stream. It is a base for fstream and strstream.

The stream is implemented by the class ios::bp is pointing to. Depending on which derived class of streambuf bp is pointing to, determines if the input stream and output stream will be the same.

For example, iostream using a filebuf will input and output to the same file. Yet iostream using a strstreambuf can have the input and output stream go to the same or different memory locations.

Constructors
iostream::iostream

Member Functions
None.

iostream::iostream

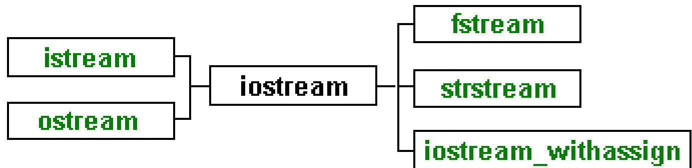
iostream class

Syntax

```
iostream(streambuf *);
```

Description

Associates a given streambuf with the class.


istream
ostream
iostream
fstream
strstream
iostream_withassign

istream_withassign class

See Also Hierarchy

Header File
iostream.h

Description
This class is an istream that overloads the **=** operator and lets you reassign the pointer ios::bp to a different derived class of streambuf.

Constructors
istream_withassign::istream_withassign

Member Functions
None (although the **=** operator is overloaded).

istream_withassign::istream_withassign

istream_withassign class

Syntax

```
istream_withassign()
```

Description

Null constructor (calls the default constructor for istream).

See Also
Overloading Operators

istream — istream_withassign

ostream_withassign class

See Also   Hierarchy

Header File
iostream.h

Description
This class is an ostream that overloads the = operator and allows you to reassign the pointer ios::bp to a different derived class of streambuf.

Constructors
ostream_withassign::ostream_withassign

Member Functions
None (although the **=** operator is overloaded).

See Also
Overloading Operators

ostream_withassign::ostream_withassign

ostream_withassign class

Syntax

```
ostream_withassign();
```

Description

Null constructor (calls the default constructor for ostream).

ostream — ostream_withassign

istream class

See Also Hierarchy Example

Header File
iostream.h

Description
Provides formatted and unformatted input from a derived class of class streambuf via *ios::bp*.

An instance of class istream does not perform the actual input, but the member functions of class istream call the member functions of the class *bp* is pointing to extract the characters from the input stream.

The **>>** operator, which is overloaded for all fundamental types, can then format the data.

istream provides the generic code for formatting the data after it is extracted from the input stream.

Constructors
istream::istream

Protected Member Functions
eatwhite

Public Member Functions
gcount

get

getline

ignore

ipfx

peek

putback

read

seekg

tellg

See Also

Operator >>

istream::istream

istream class

Syntax

```
istream(streambuf *);
```

Description

Associates a given derived class of streambuf to the class thus providing an input stream. This is done by assigning *ios::bp* to the parameter of the constructor.

istream::eatwhite

istream class

Syntax

```
void eatwhite(); // PROTECTED
```

Description

Extracts consecutive whitespace.

istream::gcount

Example istream class

Syntax

```
int gcount();
```

Description

The gcount member function returns the number of unformatted characters last extracted. Unformatted extraction occurs within the member functions get, getline, and read.

// istream::gcount and istream::getline Example

```
#include <iostream.h>

void main(void) {
  char *name;
  int buf_size = 100;
  int count = 0;          // Character counter.

  name = new char[buf_size];

  // Notice that the output buffer is flushed.
  cout << "\n Enter your name:" << endl;
  cin.getline(name, buf_size);

  count = cin.gcount();
  // Since getline() retains the linefeed, gcount()
  // will count it as input.
  cout << "\nName character count: " << count - 1;
}
```

istream::get

istream class

Form 1

```
int get();
```

Form 2

```
istream& get(char*, int len, char = '\n');
istream& get(signed char*, int len, char = '\n');
istream& get(unsigned char*, int len, char = '\n')
```

Form 3

```
istream& get(char&);
istream& get(signed char&);
istream& get(unsigned char&);
```

Form 4

```
istream& get(streambuf&, char = '\n');
```

Description

Form 1: Extracts the next character or EOF.

Form 2: Extracts characters into the given char * until the delimiter (third parameter) or end-of-file is encountered, or until (*len* - 1) bytes have been read. A terminating null is always placed in the output string. The delimiter is not extracted from the input stream. Fails only if no characters were extracted.

Form 3: Extracts a single character into the given character reference.

Form 4: Extracts characters into the given *streambuf* until the delimiter is encountered.

istream::getline

Example istream class

Syntax

```
istream& getline(char*, int, char = '\n');
istream& getline(signed char*, int, char = '\n');
istream& getline(unsigned char*, int, char = '\n');
```

Description

The getline member function extracts up to the delimiter, puts the characters in the buffer, removes the delimiter from the input stream and does not put the delimiter into the buffer.

istream::ignore

istream class

Syntax

```
istream& ignore(int n = 1, int delim = EOF);
```

Description

The ignore member function causes up to *n* characters in the input stream to be skipped; stops if *delim* is encountered.

The deliminator is extracted from the stream.

See Also

istream class

istream::ipfx

istream class

Syntax

```
istream& ipfx(int n = 0);
```

Description

The *ipfx* function is called by input functions prior to fetching from an input stream. Functions which perform formatted input call *ipfx(0)*; unformatted input functions call *ipfx(1)*.

istream::peek

istream class

Syntax

```
int peek();
```

Description

The peek member function returns the next character without extraction.

istream::putback

istream class

Syntax

```
istream& putback(char);
```

Description

The putback member function pushes back a character into the stream.

istream::read

istream class

Syntax

```
istream& read(char*, int);
istream& read(signed char*, int);
istream& read(unsigned char*, int);
```

Description

The read member function extracts a given number of characters into an array. Use *gcount()* for the number of characters actually extracted if an error occurred.

istream::seekg

istream class

Form 1

```
istream& seekg(streampos pos);
```

Form 2

```
istream& seekg(streamoff offset, seek_dir dir);
```

Description

Form 1: Moves to an absolute position (as returned from tellg).

Form 2: Moves *offset* number of bytes relative to the current position for the input stream. The offset is in the direction specified by *dir* following the definition: **enum** *seek_dir {beg, cur, end}*;

Use ostream::seekp for positioning in an output stream.

Use seekpos or seekoff for positioning in a stream buffer.

istream::tellg

istream class

Syntax

```
long tellg();
```

Description

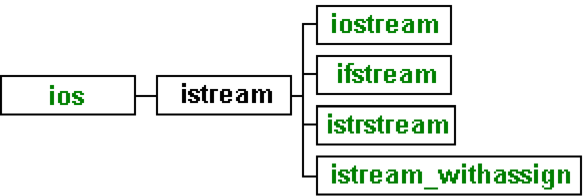
The *tellg* member function returns the current stream position.

istream Example

```
// Illustrates positioning within an input stream.
#include <iostream.h>
#include <fstream.h>

void main(void) {
   const char *filename = "_junk_.$$$";
   int size = 0;
   ifstream inf( filename, ios::in | ios::nocreate );

   inf.seekg(0L, ios::end );
   if ( (size = inf.tellg()) < 0) {
      cerr << filename << " not found";
      return;
      }
   cout << filename << " size = " << size;
   }
```


ios
istream
iostream
ifstream
istrstream
istream_withassign

ostream class

Hierarchy

Header File
iostream.h

Description
ostream provides formatted and unformatted output to a streambuf.

An instance of class ostream will not perform the actual output, but the member functions of ostream will call the member functions of the class *bp* is pointing to and insert the characters to the output stream.

The overloaded operator **<<** formats the data before it is sent to bp.

ostream provides the generic code for formatting the data before it is inserted to the output stream.

Constructors
ostream::ostream

Member Functions
flush
opfx
osfx
put
seekp
tellp
write

ostream::ostream

ostream class

Syntax

```
ostream(streambuf *buf);
```

Description

Associates a given streambuf to the class, providing an output stream. This is done by assigning the pointer ios::bp to buf.

ostream::flush

ostream class

Syntax

```
ostream& flush();
```

Description

This member function flushes the stream.

ostream::opfx

ostream class

Syntax

```
int opfx();
```

Description

The *opfx* function is called by output functions prior to inserting to an output stream. *opfx* returns 0 if the *ostream* has a nonzero error state. Otherwise, *opfx* returns a nonzero value.

ostream::osfx

ostream class

Syntax

```
void osfx();
```

Description

The *osfx* function performs post output operations. If ios::unitbuf is on, *osfx* flushes the ostream. On failure, *osfx* sets *ios::failbit*.

ostream::seekp

ostream class

Form 1

```
ostream& seekp(streampos);
```

Form 2

```
ostream& seekp(streamoff, seek_dir);
```

Description

Form 1: Moves to an absolute position (as returned from tellp).

Form 2: Moves to a position relative to the current position, following the definition: **enum** *seek_dir beg, cur, end*.

ostream::put

ostream class

Syntax

```
ostream& put(char ch);
ostream& put(signed char ch);
ostream& put(unsigned char ch);
```

Description

The put member function inserts the character.

ostream::tellp

ostream class

Syntax

```
streampos tellp();
```

Description

The tellp member function returns the current stream position.

ostream::write

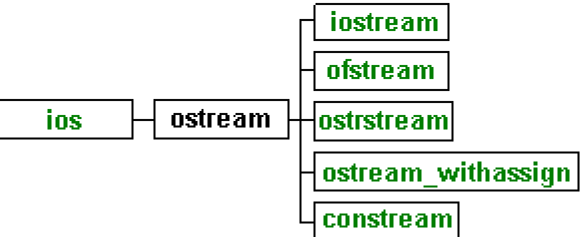
ostream class

Syntax

```
ostream& write(const char*, int n);
ostream& write(const signed char*, int n);
ostream& write(const unsigned char*, int n);
```

Description

The write member function inserts *n* characters (nulls included).


ios
ostream
iostream
ofstream
ostrstream
ostream_withassign
constream

ostrstream class

Hierarchy

Header File
strstrea.h

Description
Provides an output stream to insert from an array using a *strstreambuf*.

Constructors
ostrstream::ostrstream

Member Functions
pcount

str

ostrstream::ostrstream

ostrstream class

Form 1

```
ostrstream();
```

Form 2

```
ostrstream(char *buf, int len, int mode = ios::out);
ostrstream(signed char *buf, int len, int mode = ios::out);
ostrstream(unsigned char *buf, int len, int mode = ios::out);
```

Description

Form 1: Makes an *ostrstream* with a dynamic array for the input stream.

Form 2: Makes an *ostrstream* with a buffer specified by *buf* and size specified by *len*. If *mode* is *ios::app* or *ios::ate*, the get/put pointer is positioned at the null character of the string.

ostrstream::pcount

ostrstream class

Syntax

```
int pcount();
```

Description

Returns the number of bytes currently stored in the buffer.

ostrstream::str

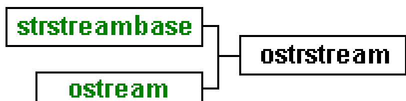
ostrstream class

Syntax

```
char *str();
```

Description

Returns and freezes the buffer. The user must deallocate it if the buffer was dynamic.


strstreambase
ostream
ostrstream

strstream class

Hierarchy

Header File
strstrea.h

Description
Provides simultaneous input and output to and from an array using a *strstreambuf*. Input and output is initiated using the functions of the base classes istream and ostream.

For example, *strstream* can use the function *istream::getline()* to extract characters from the buffer.

Constructors
strstream::strstream

Member Functions
str

strstream::strstream

strstream class

Form 1
```
strstream();
```

Form 2
```
strstream(char*, int sz, int mode);
```

Form 3
```
strstream(signed char*, int sz, int mode);
```

Form 4
```
strstream(unsigned char*, int sz, int mode);
```

Description

Form 1: Makes a *strstream* with the base class strstreambase's streambuf data member's buffer dynamically allocated the first time it is used. The put area and get areas are the same.

Form 2: Makes a *strstream* with a specified *n*-byte buffer. If *mode* is *ios::app* or *ios::ate*, the get/put pointer is positioned at the null character of the string.

Form 3: Makes a *strstream* with a specified *n*-byte buffer. If *mode* is *ios::app* or *ios::ate*, the get/put pointer is positioned at the null character of the string.

Form 4: Makes a strstream with a specified n-byte buffer. If *mode* is ios::app or ios::ate, the get/put pointer is positioned at the null character of the string.

strstream::str

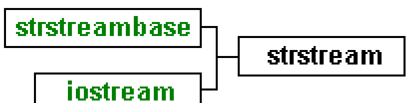
strstream class

Syntax

```
char *str();
```

Description

Returns and freezes the buffer. The user must de-allocate it if the buffer was dynamic.


strstreambase
iostream
strstream

strstreambase class

Hierarchy

Header File
strstrea.h

Description
Specializes ios to string streams by initializing ios::bp to point to a strstreambuf. This provides the condition checks necessary for any string I/O in memory. For this reason, strstreambase is almost entirely protected and accessible only to derived classes which perform I/O. It makes virtual use of ios.

Constructors
strstreambase::strstreambase

Member Functions
rdbuf

strstreambase::strstreambase

strstreambase class

Form 1
```
strstreambase();
```

Form 2
```
strstreambase(const char*, int, char *start);
```

Description

Form 1: Makes a *strstreambase* with its streambuf data member's buffer dynamically allocated the first time it is used. The put area and get areas are the same.

Form 2: Makes an *strstreambase* with a specified buffer and starting position.

strstreambase::rdbuf

strstreambase class

Syntax

```
strstreambuf * rdbuf();
```

Description

Returns a pointer to the *strstreambuf* associated with this object.

ios
strstreambase
strstream
istrstream
ostrstream

ios class

Hierarchy

Header File
iostream.h

Description
Provides operations common to both and input and output. Its derived classes (istream, ostream, and iostream) specialize I/O with high-level formatting operations. The ios base class is a base for istream, ostream, fstreambase, and strstreambase.

Protected Constructors
ios::ios

Public Constructors
ios::ios

Protected Data Members
bp

state

x_fill

x_flags

x_precision

*x_tie

x_width

Public Data Members
seek_dir

open_mode

adjustfield

basefield

floatfield

Protected Member Functions
init

setstate

Public Member Functions
bad

bitalloc

clear

eof

fail

fill

flags

flags

good

precision

rdbuf

rdstate

setf
sync_with_stdio
tie
unsetf
width
xalloc

ios::ios

ios class

Form 1

```
ios();    // PROTECTED
```

Form 2

```
ios(streambuf *);
```

Description

Form 1: Constructs an ios object that has no corresponding streambuf. A derived class should call ios::init(streambuf *) to provide a streambuf.

Form 2: Associates a given streambuf with the stream by assigning the pointer ios::bp to point to the streambuf passed in as a parameter.

ios::seek_dir

ios class

Syntax

```
enum seek_dir { beg=0, cur=1, end=2 };
```

Description

Stream seek direction.

ios::open_mode

ios class

Syntax

```
enum open_mode  {
   app,           // Append data--always write at end of file.
   ate,           // Seek to end of file upon original open.
   in,            // Open for input (default for ifstreams).
   out,           // Open for output (default for ofstreams).
   binary,        // Open file in binary mode.
   trunc,         // Discard contents if file exists (default if out
                  // is specified and neither ate nor app is specified).
   nocreate,      // If file does not exist, open fails.
   noreplace,     // If file exists, open for output fails unless ate or app
  is set.
};
```

Description

Stream operation mode. These parameters can be logically ORed.

ios::adjustfield

ios class

Syntax

```
static const long adjustfield;
```

Description

Use the adjustfield data member with setf to control padding to the left, right, or for internal fill.

Examples

```
cout<<setf(ios::left, ios::adjustfield)<<hex<<0xFE;
```

Result: 000xFE   left filled

```
cout<<setf(ios::internal, ios::adjustfield)<<hex<<0xFE;
```

Result: 0x00FE   internal filled

ios::basefield

ios class

Syntax

```
static const long basefield;
```

Description

Use the basefield data member with setf to set the notation to a decimal, octal, or hexidecimal base.

Example

The following example sets a decimal base.

```
cout<<setf(ios::dec, ios::basefield)<<i;
```

ios::bp

ios class

Syntax

```
streambuf *bp();   // PROTECTED
```

Description

The bp data member points to the associated streambuf.

ios::floatfield

ios class

Syntax

```
static const long floatfield;
```

Description

Use the floatfield data member with setf to set the floating-point notation to scientific or fixed.

Example

The following example sets scientific notation.

```
cout<<setf(ios::scientific, ios::floatfield)<<f;
```

ios::state

ios class

Syntax

```
int state;    // PROTECTED
```

Description

The state data member is the current state of the streambuf.

ios::x_fill

ios class

Syntax

```
int x_fill;   // PROTECTED
```

Description

Use the x_fill data member for padding character on output.

ios::x_flags

ios class

Syntax

```
long x_flags;    // PROTECTED
```

Description

Use the x_flags data member for formatting flag bits.

ios::x_precision

ios class

Syntax

```
int x_precision;   // PROTECTED
```

Description

Use the x_precision data member for floating-point precision on output.

ios::*x_tie

ios class

Syntax

```
ostream *x_tie
```

Description

Use the *x_tie data member to specify the tied ostream, if any.

ios::x_width

ios class

Syntax

```
int x_width;    // PROTECTED
```

Description

Use the x_width data member to specify the field width on output.

ios::bad

ios class

Syntax

```
int bad();
```

Description

The bad member function returns nonzero if error occured by checking ios::badbit and ios::hardfail in ios::state.

ios::bitalloc

ios class

Syntax

```
static long bitalloc();
```

Description

The bitalloc member function acquires a new flag bit set.

The return value may be used to set, clear, and test the flag. This is for user-defined formatting flags.

ios::clear

ios class

Syntax

```
void clear(int = 0);
```

Description

The clear member function sets the stream state to the given value by setting ios::state to the given value.

The constants of the io_state enumeration in class ios are normally used as the parameter.

The values of io_state can be ORed together to set more than one bit in *state*.

ios::eof

ios class

Syntax

```
int eof();
```

Description

The eof member function returns nonzero on end of file by checking the ios::eofbit in ios::state.

ios::fail

ios class

Syntax

```
int fail();
```

Description

The fail member function returns nonzero if an operation failed by checking the ios::failbit, ios::badbit, or ios::hardfail bits in ios::state.

ios::fill

ios class

Form 1

```
char fill();
```

Form 2

```
char fill(char);
```

Description

Form 1: Returns the current fill character.

Form 2: Resets the fill character; returns the previous one.

ios::flags

ios class

Form 1

```
long flags();
```

Form 2

```
long flags(long);
```

Description

Form 1: Returns the current format flags. The format flags can be compared to the values in the formatting flags enumeration of class ios. flags(0) resets the formatting flags as the default value.

Form 2: Sets the format flags to be identical to the given long. The flags of the long are set using the values in the formatting flags enumeration in class ios. It returns the previous flags. flags(0) resets the default format.

ios::good

ios class

Syntax

```
int good();
```

Description

The good member function returns nonzero if no state bits were set (no errors occurred) in ios::state.

ios::init

ios class

Syntax

```
void init(streambuf *);    // PROTECTED
```

Description

The init member function associates the ios with the specified streambuf.

ios::precision

ios class

Form 1

```
int precision(int);
```

Form 2

```
int precision();
```

Description

Form 1: Sets the floating-point precision, and returns the previous setting. This must be reset for each data item being output if a precision other than the default is desired.

Form 2: Returns the current floating-point precision.

ios::rdbuf

Syntax

```
streambuf* rdbuf();
```

Description

Returns a pointer to this stream's assigned streambuf.

ios::rdstate

ios class

Syntax

```
int rdstate();
```

Description

Returns the stream state by returning the value of the data member state of class ios.

ios::setf

ios class

Form 1

```
long setf(long _setbits, long _field);
```

Form 2

```
long setf(long);
```

Description

Form 1: Clears the bits corresponding to those marked in _field in the data member x_flags, and then resets those marked in _setbits.   _setbits can be specifed by using the constants in the format flags. enumeration of class ios.

Form 2: Sets the flags corresponding to those marked in the given long. The flags are set in ios::x_flags. The long can be specified by using the constants in the formatting flags enumeration of class ios. It returns the previous settings.

ios::setstate

ios class

Syntax

```
void setstate(int);    // PROTECTED
```

Description

Sets specified status bits.

ios::sync_with_stdio

ios class

Syntax

```
static void sync_with_stdio();
```

Description

The sync_with_stdio member function synchronizes stdio files and iostreams.

Note: Do not use in new code. It will slow performance.

ios::tie

ios class

Form 1

```
ostream* tie();
```

Form 2

```
ostream* tie(ostream*);
```

Description

Form 1: Returns the tied stream, or 0 if none. Tied streams are stream that are connected so that when one is used, the other is affected in some way. For example, cin and cout are tied; when cin is used, it flushes cout first.

Form 2: Ties another stream to this one and returns the previously tied stream, if any. When an input stream has characters to be consumed, or if an output stream needs more characters, the tied stream is first flushed automatically. By default, cin, cerr and clog are tied to cout.

ios::unsetf

ios class

Syntax

```
long unsetf(long);
```

Description

The unsetf member function clears the bits corresponding to those marked in the given long.

The bits are cleared in ios::x_flags.

The flags of the long can be set using the constants in the format flags. enumeration of class ios.

unsetf returns the previous settings.

ios::width

ios class

Form 1

```
int width();
```

Form 2

```
int width(int);
```

Description

Form 1: Returns the current width setting.

Form 2: Sets the width, and returns the previous width. This must be reset for each data item input or output if a width other than the default is desired.

ios::xalloc

ios class

Syntax

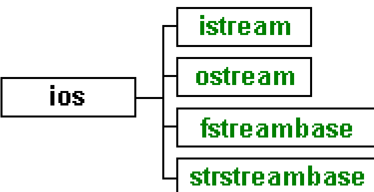
```
static int xalloc();
```

Description

The xalloc member function returns an array index of previously unused words that can be used as user-defined formatting flags.

Format Flags

```
enum {
   skipws,        Skip whitespace on input.
   left,          Left-adjust output.
   right,         Right-adjust output.
   internal,      Pad after sign or base indicator.
   dec,           Decimal conversion.
   oct,           Octal conversion.
   hex,           Hexadecimal conversion.
   showbase,      Show base indicator on output.
   showpoint,     Show decimal point for floating-point output.
   uppercase,     Uppercase hex output.
   showpos,       Show '+' with positive integers.
   scientific,    Suffix floating-point numbers with exponential (E) notation on output.
   fixed,         Use fixed decimal point for floating-point numbers.
   unitbuf,       Flush all streams after insertion.
   stdio,         Flush stdout, stderr after insertion.
};
```



ios
istream
ostream
fstreambase
strstreambase

istrstream class

Hierarchy

Header File
strstrea.h

Description
Provides input operations on a strstreambuf.

The cluster (ios, istream, ostream, iostream, and streambuf), provides a base for specialized cluster that deals with memory.

Constructors
istrstream::istrstream

Member Functions
None.

istrstream::istrstream

istrstream class

Form 1

```
istrstream(unsigned char *);
istrstream(char *);
istrstream(signed char *);
```

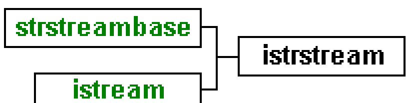
Form 2

```
istrsteam(signed char *str, int);
istrstream(char *str, int n);
istrstream(unsigned char *str, int);
```

Description

Form 1: Makes an *istrstream* with a specified string (a null character is never extracted).

Form 2: Makes an *istrstream* using up to *n* bytes of *str*.


strstreambase
istream
istrstream

streambuf class

Hierarchy Example

Header File
iostream.h

Description
.This is a base class for all other buffering classes. It provides a buffer interface between your data and storage areas such as memory or physical devices. The buffers created by *streambuf* are referred to as get, put, and reserve areas. The contents are accessed and manipulated by pointers that point between characters.

Buffering actions performed by *streambuf* are rather primitive. Normally, applications gain access to buffers and buffering functions through a pointer to *streambuf* that is set by *ios*. Class *ios* provides a pointer to *streambuf* that provides a transparent access to buffer services for high-level classes. The high-level classes provide I/O formatting.

Constructors
streambuf::streambuf

Protected Member Functions
allocate
base
blen
eback
ebuf
egptr
epptr
gbump
gptr
pbase
pptr
setg
setb
setp
unbuffered

Public Member Functions
in_avail
out_waiting
pbump
sbumpc
seekoff
seekpos
setbuf
sgetc
sgetn
snextc
sputbackc

sputc
sputn
stossc

streambuf::streambuf

streambuf class

Form 1

```
streambuf();
```

Form 2

```
streambuf(char *buf, int size);
```

Description

Form 1: Creates an empty buffer object.

Form 2: Constructs an empty buffer *buf* and sets up a reserve area for *size* number of bytes.

streambuf::allocate

streambuf class

Syntax

```
int allocate();   // PROTECTED
```

Description

The *allocate* member function sets up a buffer area.

streambuf::base

streambuf class

Syntax

```
char *base();  // PROTECTED
```

Description

The *base* member function returns the start of the buffer area.

streambuf::blen

streambuf class

Syntax

```
int blen();  // PROTECTED
```

Description

The *blen* member function returns the length of the buffer area.

streambuf::eback

streambuf class

Syntax

```
char *eback();  // PROTECTED
```

Description

The *eback* member function returns the base of putback section of get area.

streambuf::ebuf

streambuf class

Syntax

```
char *ebuf();  // PROTECTED
```

Description

The *ebuf* member function returns the end+1 of the buffer area.

streambuf::egptr

streambuf class

Syntax

```
char *egptr();  // PROTECTED
```

Description

The *egptr* member function returns the end+1 of the get area.

streambuf::epptr

streambuf class

Syntax

```
char *epptr();  // PROTECTED
```

Description

The *epptr* member function returns the end+1 of the put area.

streambuf::gbump

streambuf class

Syntax

```
void gbump(int n);   // PROTECTED
```

Description

The *gbump* member function advances the get pointer by *n* which may be positive or negative.

No checks are performed on the new value.

streambuf::gptr

streambuf class

Syntax

```
char *gptr();   // PROTECTED
```

Description

The *gptr* member function returns the next location in get area.

streambuf::in_avail

streambuf class

Syntax

```
int in_avail();
```

Description

The *in_avail* member function returns the number of characters remaining in the internal input buffer.

This may be the input stream, depending on which derived class of streambuf the function call originated from.

streambuf::out_waiting

streambuf class

Syntax

```
int out_waiting();
```

Description

The *out_waiting* member function returns the number of characters remaining in the internal output buffer.

This may be the output stream, depending on which derived class of streambuf the function call originated from.

streambuf::pbase

streambuf class

Syntax

```
char *pbase();  // PROTECTED
```

Description

The *pbase* member function returns the start of put area.

streambuf::pbump

streambuf class

Syntax

```
void pbump(int); // PROTECTED
```

Description

The *pbump* member function increments the put pointer pptr() by n which may be positive or negative.

No checks are performed on the new value of pptr().

streambuf::pptr

streambuf class

Syntax

```
char *pptr();  // PROTECTED
```

Description

The pptr member function returns a pointer to the next location in the put area.

streambuf::sbumpc

streambuf class

Syntax

```
int sbumpc();
```

Description

The *sbumpc* member function returns the current character from the internal input buffer, then advances.

This may be the input stream depending on which derived class of streambuf the function call originated from.

streambuf::seekpos

streambuf class

Syntax

```
virtual streampos seekpos(streampos, int = (ios::in | ios::out));
```

Description

The *seekpos* member function moves the get and/or put pointer to an absolute position in the internal buffer of the streambuf.

Because seekpos is virtual, it may be redefined in a derived class to reposition in the input and/or ouput stream.

streambuf::setb

streambuf class

Syntax

```
void setb(char *, char *, int = 0);   // PROTECTED
```

Description

The *setb* member function sets the buffer area.

streambuf::setg

streambuf class

Syntax

```
void setg(char *, char *, char *);  // PROTECTED
```

Description

The *setg* member function initializes the get pointers.

streambuf::setp

streambuf class

Syntax

```
void setp(char *, char *);   // PROTECTED
```

Description

The *setp* member function initializes the put pointers.

streambuf::sgetc

streambuf class

Syntax

```
int sgetc();
```

Description

The *setc* member function peeks at the next character in the internal input buffer.

streambuf::sgetn

streambuf class

Syntax

```
int sgetn(char*, int n);
```

Description

The *sgetn* member function gets the next *n* characters from the internal input buffer.

streambuf::snextc

streambuf class

Syntax

```
int snextc();
```

Description

The *snextc* member function advances to and returns the next character from the internal input buffer.

streambuf::sputbackc

streambuf class

Syntax

```
int sputbackc(char);
```

Description

The *sputbackc* member function returns a character to the internal input buffer.

streambuf::sputc

streambuf class

Syntax

```
int sputc(int);
```

Description

The *sputc* member function puts one character into the internal output buffer.

streambuf::sputn

streambuf class

Syntax

```
int sputn(const char*, int n);
```

Description

The *sputn* member function puts *n* characters into the internal output buffer.

streambuf::stosc

streambuf class

Syntax

```
void stossc();
```

Description

The *stossc* member function advances to the next character in the internal input buffer.

streambuf::unbuffered

streambuf class

Form 1

```
void unbuffered(int);
```

Form 2

```
int unbuffered();  // PROTECTED
```

Description

Form 1: Sets the buffering state.

Form 2: Returns non-zero if not buffered.

streambuf::seekoff

streambuf class

Syntax

```
virtual streampos seekoff(streamoff offset, ios::seek_dir, int mode);
```

Description

Moves the file pointer relative to the current position in the direction of *seek_dir*.

seek_dir is set using the seek_dir enumeration definition in class ios.

ios::beg	seek from beginning of file
ios::cur	seek from current location
ios::end	seek from end of file

Since the long can be a negative value, seeking can occur "backward" in the file from the end or current location.

mode specifies the move to be in get or put area of the internal buffer by using *ios::in, ios::out* or both.

When this virtual member function is redefined in a derived class, it could be seeking into the stream and not streambuf's internal buffer.

streambuf::setbuf

streambuf class

Syntax

```
streambuf* setbuf(unsigned char*, int);
```

Description

Uses the specified array for the internal buffer.

streambuf Example

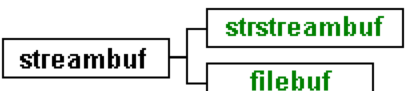
```
// Operations with streambufs.
#include <iostream.h>
#include <fstream.h>

int main(void) {
   int c;
   const char *filename = "_junk_.$$$";
   ofstream outfile;
   streambuf *out, *input = cin.rdbuf();

   // Position at the end of file. Append all text.
   outfile.open( filename, ios::ate | ios::app);
   if (!outfile) {
      cerr << "Could not open " << filename;
      return(-1);
      }

   out = outfile.rdbuf();  // Connect ofstream and streambuf.

   clog << "Input some text. Use Control-Z to end." << endl;
   while ( (c = input -> sbumpc() ) != EOF) {
      cout << char(c);                          // Echo to screen.
      if (out -> sputc(c) == EOF)
         cerr << "Output error";
      }
   return(0);
   }
```


streambuf
strstreambuf
filebuf

strstreambuf class

Hierarchy

Header File
strstrea.h

Description
strstreambuf specializes streambuf to create a buffer for in-memory string formatting.

strstreambuf is one of the two classes defined in the C++ stream library which provide a place for input to be gathered from and a place for output to go. The other class is filebuf.

The I/O functions of istream and ostream make calls to the functions of *strstreambuf* to do the actual insertion or extraction on the streams.

Constructors
strstreambuf::strstreambuf

Member Functions
doallocate

freeze

overflow

seekoff

setbuf

str

sync

underflow

strstreambuf::strstreambuf

strstreambuf class

Form 1

```
strstreambuf();
```

Form 2

```
strstreambuf(void * (*alloc)(long n), void (*release)(void *buffer));
```

Form 3

```
strstreambuf(int n);
```

Form 4

```
strstreambuf(char *buf, int n, char *strt = 0);
strstreambuf(signed char *buf, int n, signed char *strt = 0);
strstreambuf(unsigned char *buf, int n, unsigned char *strt = 0);
```

Description

Form 1: Makes a dynamic *strstreambuf*. Memory will be dynamically allocated as needed. The put area and get areas are the same.

Form 2: Makes a dynamic buffer with specified allocation and free functions.

Form 3: Makes a dynamic *strstreambuf*, initially allocating a buffer of at least *n* bytes.

Form 4: This *strstreambuf* constructor creates a static *strstreambuf*. The streambuf uses *n* bytes starting at the position pointed to by *buf*. The *buf* pointer indicates the get area.

N = 0	*buf* points to a null-terminated string which constitutes the strstreambuf
N < 0	strstreambuf is not terminated and continues indefinitely
N > 0	indicates the number of bytes used by strstreambuf beginning at the position pointed to by *buf*

When *pstrt* is NULL the array is only available for get options.

strstreambuf::doallocate

strstreambuf class

Syntax

```
virtual int doallocate ();
```

Description

Performs low-level buffer allocation.

strstreambuf::freeze

strstreambuf class

Syntax

```
void freeze(int = 1);
```

Description

The freeze member function disallows storing any characters in the buffer, if the input parameter is nonzero.

Unfreeze the buffer by passing a zero.

strstreambuf::overflow

strstreambuf class

Syntax

```
virtual int overflow(int = EOF);
```

Description

Flushes a buffer to its destination. Every derived class should define the actions to be taken.

strstreambuf::seekoff

strstreambuf class

Syntax

```
virtual streampos seekoff(streamoff offset, ios::seek_dir, int mode);
```

Description

Moves the file pointer relative to the current position in the direction of *seek_dir*.

seek_dir is set using the *seek_dir* enumeration definition in class ios.

ios::beg	seek from beginning of file
ios::cur	seek from current location
ios::end	seek from end of file

Since the long can be a negative value, seeking can occur "backward" in the file from the end or current location.

Mode specifies the move to be in get or put area of the internal buffer by using *ios::in, ios::out* or both.

When this virtual member function is redefined in a derived class, it could be seeking into the stream and not streambuf's internal buffer.

strstreambuf::setbuf

strstreambuf class

Syntax

```
virtual streambuf* setbuf(char*, int);
```

Description

Specifies a buffer of a specified size for the class object. When used as a strstreambuf and the function is overloaded, the first argument is not meaningful and should be set to zero.

strstreambuf::str

strstreambuf class

Syntax

```
char *str();
```

Description

Returns and freezes the buffer. The user must deallocate it if the buffer was dynamic.

strstreambuf::sync

strstreambuf class

Syntax

```
virtual int sync();
```

Description

Establishes consistency between internal data structures and the external stream representation.

strstreambuf::underflow

strstreambuf class

Syntax

```
virtual int underflow();
```

Description

Makes input available. This is called when no more data exists in the input buffer. Every derived class should define the actions to be taken.

streambuf — strstreambuf

Persistent Streams (C++)

Streamable Class Hierarchy

Borland support for persistent streams consists of a class hierarchy and macros to help you develop streamable objects. These topics are a reference for these classes and macros. They alphabetically list and describe all the public classes that support persistent objects.

To get Help about an persistent stream class, choose it from the list below:

fpbase

ifpstream

ipstream

ofpstream

opstream

pstream

TStreamableBase

TStreamableClass

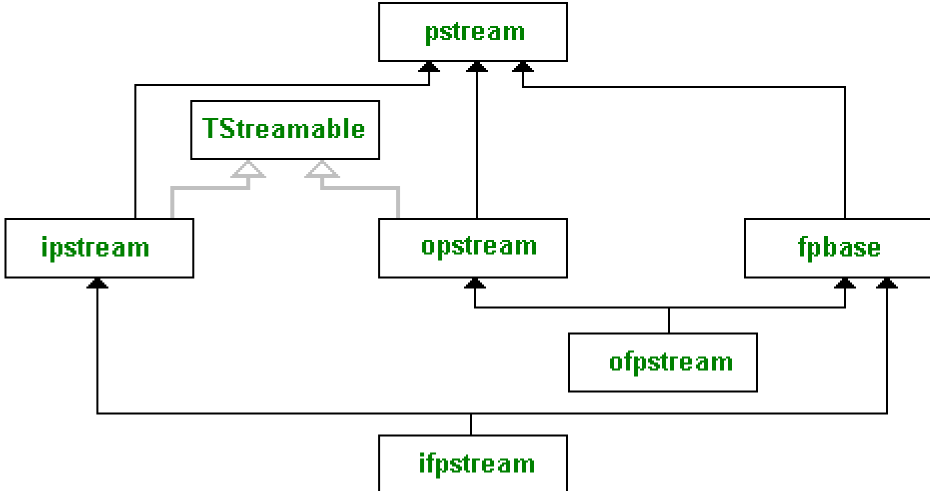
TStreamer

Streaming Macros

Persistent Stream Class Hierarchy

See Also

In this persistent streams hierarchy diagram, parenthood proceeds from left-to-right. persistent stream classes have multiple inheritance.



The gray arrows connecting TStreamableClass indicate a friend class of ipstream and opstream.

See Also

Streaming Macros

fpbase class

Hierarchy

Syntax

```
class fpbase : virtual public pstream
```

Header File

objstrm.h

Description

Provides the basic operations common to all object file stream I/O. It is a base class for handling streamable objects on file streams.

Constructors

fpbase::fpbase

Member Functions

attach

close

open

rdbuf

setbuf

fpbase::fpbase

fpbase class

Form 1

```
fpbase();
```

Form 2

```
fpbase(const char *name, int omode, int prot = filebuf::openprot);
```

Form 3

```
fpbase(int f);
```

Form 4

```
fpbase(int f, char *b, int len);
```

Description

Form 1: Creates a buffered *fpbase* object.

Form 2: Creates a buffered *fpbase* object. It opens the file specified by *name*, using the mode *omode* and protection *prot*; and attaches this file to the stream.

Form 3: Creates a buffered *fpbase* object, and attaches the file specified by the file descriptor *f* to the stream.

Form 4: Creates a buffered *fpbase* object. It initializes the file buffer to be associated with the file descriptor *f,* and to use the buffer specified by *b* with a length of *len*.

fpbase::attach

fpbase class

Syntax

```
void attach(int f);
```

Description

Attaches the file with descriptor *f* to this stream if possible and sets *ios::state* accordingly.

fpbase::close

fpbase class

Syntax

```
void close();
```

Description

Closes the stream and associated file.

fpbase::open

fpbase class

Syntax

```
void open(const char *name, int mode, int prot = filebuf::openprot);
```

Description

Opens the named file in the given *mode* (*app*, *ate*, *in*, *out*, *binary*, *trunc*, *nocreate*, or *noreplace*) and protection. The opened file is attached to this stream.

fpbase::rdbuf

fpbase class

Syntax

```
filebuf * rdbuf();
```

Description

Returns a pointer to the current file buffer.

fpbase::setbuf

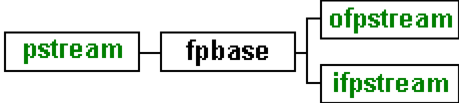
fpbase class

Syntax

void setbuf(char *buf, int len);

Description

Sets the location of the buffer to *buf* and the buffer size to *len*.


pstream
fpbase
ofpstream
ifpstream

ifpstream class

Hierarchy

Syntax

```
class ifpstream : public fpbase, public ipstream
```

Header File

objstrm.h

Description

ifpstream is a simple "mix" of its bases, fpbase and ipstream. It provides the base class reading (extracting) streamable objects from file streams.

Constructors

ifpstream::ifpstream

Member Functions

open

rdbuf

ifpstream::ifpstream

ifpstream class

Form 1
```
ifpstream();
```

Form 2
```
ifpstream(const char *name, int mode=ios::in, int prot =
  filebuf::openprot);
```

Form 3
```
ifpstream(int f);
```

Form 4
```
ifpstream(int f, char *b, int len);
```

Description

Form 1: Creates a buffered *ifpstream* object using a default buffer.

Form 2: Creates a buffered *ifpstream* object. It opens the file specified by *name* using the mode *mode* and protection *prot*; and attaches this file to the stream.

Form 3: Creates a buffered *ifpstream* object and attaches the file specified by the file descriptor, *f* to the stream.

Form 4: Creates a buffered *ifpstream* object. It initializes the file buffer to be associated with the file descriptor *f* and to use the buffer specified by *b* with a length of *len*.

ifpstream::open

ifpstream class

Syntax

```
void open(const char *name, int mode, int prot = filebuf::openprot);
```

Description

It opens the named file in the given *mode* (*app*, *ate*, *in*, *out*, *binary*, *trunc*, *nocreate*, or *noreplace*) and protection. The default *mode* for ifpstream is ios::in (input) with *openprot* protection. The opened file is attached to this stream.

ifpstream::rdbuf

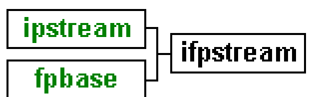
ifpstream class

Syntax

```
filebuf * rdbuf();
```

Description

Returns a pointer to the current file buffer.


ipstream
fpbase
ifpstream

ipstream class

Hierarchy

Syntax

```
class ipstream : virtual public pstream
```

Header File

objstrm.h

Description

ipstream, a specialized input stream derivative of pstream, is the base class for reading (extracting) streamable objects.

Public Constructors

ipstream::ipstream

Protected Constructors

ipstream::ipstream

Public Member Functions

find

freadBytes

freadString

getVersion

readByte

readBytes

readString

readWord

readWord16

readWord32

registerObject

seekg

tellg

Protected Member Functions

readData

readPrefix

readSuffix

readVersion

Operators

operator >>

Friend Operators

operator >>

ipstream::ipstream

ipstream class

Form 1

```
ipstream(streambuf *buf);
```

Form 2

```
ipstream();    // PROTECTED
```

Description

Form 1: Creates a buffered *ipstream* with the given buffer. The state is set to 0.

Form 2: Creates a buffered *ipstream* without initializing the buffer pointer, bp. Use psteam::init to set the buffer and state.

ipstream::find

ipstream class

Syntax

```
TStreamableBase _BIDSFAR * find(P_id_type Id);
```

Description

Returns a pointer to the object corresponding to *Id*.

ipstream::freadBytes

ipstream class

Syntax

```
void freadBytes( void _BIDSFARDATA *data, size_t sz );
```

Description

Reads the number of bytes specified by *sz* into the supplied buffer (*data*).

ipstream::freadString

ipstream class

Form 1

```
char _BIDSFARDATA *freadString();
```

Form 2

```
char _BIDSFARDATA *freadString( char _BIDSFARDATA *buf, unsigned maxLen );
```

Description

Form 1: Reads a string from the stream. It determines the length of the string and allocates a far character array of the appropriate length. It reads the string into this array and returns a pointer to the string. The caller is expected to free the allocated memory block.

Form 2: Reads a string from the stream into the supplied far buffer (*buf*). If the length of the string is greater than *maxLen*-1, it reads nothing. Otherwise, it reads the string into the buffer and appends a null-terminating byte.

ipstream::getversion

ipstream class

Syntax

```
getVersion() const;
```

Description

Returns the object version number.

ipstream::readByte

ipstream class

Syntax

```
uint8 readByte();
```

Description

Returns the byte at the current stream position.

ipstream::readBytes

ipstream class

Syntax

```
void readBytes(void data, size_t sz);
```

Description

Reads *sz* bytes from current stream position, and writes them to *data*.

ipstream::readData

See Also ipstream class

Syntax

```
void _BIDSFAR * readData(const ObjectBuilder _BIDSFAR* ,TStreamableBase
  _BIDSFAR *& mem);
```

Description

If *mem* is 0, it calls the appropriate *build* function to allocate memory and initialize the virtual table pointer for the object.

Finally, it invokes the appropriate read function to read the object from the stream into the memory pointed to by *mem*.

See Also

ipstream class

TStreamableClass

ipstream::readPrefix

ipstream class

Syntax

```
const ObjectBuilder _BIDSFAR * readPrefix();
```

Description

Returns the TStreamableClass object corresponding to the class *name* stored at the current position in the stream.

ipstream::readString

ipstream class

Form 1

```
char _BIDSFAR * readString();
```

Form 2

```
char _BIDSFAR * readString( char _BIDSFAR *buf, unsigned maxLen);
```

Description

Form 1: Allocates a buffer large enough to contain the string at the current stream position and reads the string into the buffer. The caller must free the buffer.

Form 2: Reads the string at the current stream position into the buffer specified by *buf*. If the length of the string is greater than *maxLen* - 1, it reads nothing. Otherwise, it reads the string into the buffer and appends a null-terminating byte.

ipstream::readSuffix

See Also   ipstream class

Syntax

```
void readSuffix();
```

Description

Reads and checks the suffix of the object.

See Also
ipstream class::readPrefix

ipstream::readWord

ipstream class

Syntax

```
uint32 readWord();
```

Description

Returns the word at the current stream position.

ipstream::readWord16

ipstream class

Syntax

```
uint16 readWord16();
```

Description

Returns the 16-bit word at the current stream position.

ipstream::readWord32

ipstream class

Syntax

```
uint32 readWord32();
```

Description

Returns the 32-bit word at the current stream position.

ipstream::registerObject

ipstream class

Syntax

```
void registerObject(TStreamableBase * adr);
```

Description

Registers the object pointed to by *adr*.

ipstream::seekg

ipstream class

Form 1

```
ipstream& seekg(streampos pos);
```

Form 2

```
ipstream& seekg(streamoff off, ios::seek_dir);
```

Description

Form 1: Moves the stream position to the absolute position given by *pos*.

Form 2: Moves to a position relative to the current position by an offset *off* (+ or -) starting at *ios::seek_dir*. You can set *ios::seek_dir* to one of the following:

__*beg*(start of stream)

__*cur* (current stream position)

  end (end of stream).

ipstream::tellg

ipstream class

Syntax

```
streampos tellg();
```

Description

Returns the (absolute) current stream position.

ipstream::readVersion

ipstream class

Syntax

```
void readVersion();
```

Description

Reads the version number of the input stream.

ipstream::operator >>

ipstream class

Syntax

```
ipstream _BIDSFAR& operator >> ( ipstream _BIDSFAR& is, string _BIDSFAR&
  str );
```

Description

This operator of *ipstream* extracts (reads) from the *ipstream is*, to the string *str*. It returns a reference to the stream that lets you chain **>>** operations in the usual way.

operator >>

ipstream class

Syntax

```
friend ipstream& operator >> (ipstream& ps, signed char _BIDSFAR & ch);
friend ipstream& operator >> (ipstream& ps, unsigned char _BIDSFAR & ch);
friend ipstream& operator >> (ipstream& ps, signed short _BIDSFAR & sh);
friend ipstream& operator >> (ipstream& ps, unsigned short _BIDSFAR & sh);
friend ipstream& operator >> (ipstream& ps, signed int _BIDSFAR & i);
friend ipstream& operator >> (ipstream& ps, unsigned int _BIDSFAR & i);
friend ipstream& operator >> (ipstream& ps, signed long _BIDSFAR & l);
friend ipstream& operator >> (ipstream& ps, unsigned long _BIDSFAR & l);
friend ipstream& operator >> (ipstream& ps, float _BIDSFAR & f);
friend ipstream& operator >> (ipstream& ps, double _BIDSFAR & d);
friend ipstream& operator >> (ipstream& ps, long double _BIDSFAR & d);
friend ipstream& operator >> (ipstream& ps, TStreamableBase t);
friend ipstream& operator >> (ipstream& ps, void *t);
```

Description

This friend operator of *ipstream* extracts (reads) from the *ipstream ps*, to the given argument. It returns a reference to the stream that lets you chain **>>** operations in the usual way.

The data type of the argument determines how the read is performed. For example, reading a signed *char* is implemented using *readByte*.

pstream — ipstream — ifpstream

ofpstream class

Hierarchy

Header File
objstrm.h

Syntax
```
class ofpstream : public fpbase, public opstream
```

Description
Provides the base class for writing (inserting) streamable objects to file streams.

Constructors
ofpstream::ofpstream

Member Functions
open
rdbuf

ofpstream::ofpstream

ofpstream class

Form 1

```
ofpstream();
```

Form 2

```
ofpstream(const char *name, int mode = ios::out, int prot =
  filebuf::openprot);
```

Form 3

```
ofpstream(int f);
```

Form 4

```
ofpstream(int f, char *b, int len);
```

Description

Form 1: Creates a buffered *ofpstream* object using a default buffer.

Form 2: Creates a buffered *ofpstream* object. It opens the file specified by *name*, using the mode *mode*, and protection *prot*; and attaches this file to the stream

Form 3: Creates a buffered *ofpstream* object and attaches the file specified by the file descriptor, *f* to the stream.

Form 4: Creates a buffered *ofpstream* object. It initializes the file buffer to be associated with the file descriptor *f* and to use the buffer specified by *b* with a length of *len*.

ofpstream::open

ofpstream class

Syntax

```
void open(char *name, int mode = ios::out, int prot = filebuf::openprot);
```

Description

Opens the named file in the given *mode* (*app*, *ate*, *in*, *out*, *binary*, *trunc*, *nocreate*, or *noreplace*) and protection. The default mode for ofpstream is ios::out (output) with *openprot* protection. The opened file is attached to this stream.

ofpstream::rdbuf

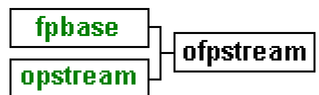
ofpstream class

Syntax

```
filebuf * rdbuf();
```

Description

Returns a pointer to the current file buffer.


fpbase
opstream
ofpstream

opstream class

Hierarchy

Header
objstrm.h

Syntax
```
class opstream : virtual public pstream
```

Description

opstream, a specialized derivative of pstream, is the base class for writing (inserting) streamable objects.

Public Constructors
opstream::opstream

Protected Constructors
opstream::opstream

Public Member Functions
findObject
findVB
flush
fwriteBytes
fwriteString
registerObject
registerVB
seekp
tellp
writeByte
writeBytes
writeObject
writeObjectPointer
writeString
writeWord
writeWord16
writeWord32

Public Member Functions
writeData
writePrefix
writeSuffix

Friend Operator
opstream::<<

opstream::opstream

opstream class

Form 1

```
opstream(streambuf *buf);
```

Form 2

```
opstream(); // PROTECTED
```

Description

Form 1: Creates a buffered *opstream* with the given buffer and sets the *bp* data member to *buf*. The state is set to 0.

Form 2: Creates an *opstream* object without initializing the buffer pointer, bp. Use pstream::init to set the buffer and state.

opstream::findObject

opstream class

Syntax

```
P_id_type findObject(TStreamableBase *adr);
```

Description

Returns the type ID for the object pointed to by *adr*.

opstream::findVB

opstream class

Syntax

```
P_id_type findVB(TStreamableBase *adr);
```

Description

Returns a pointer to the virtual base.

opstream::flush

opstream class

Syntax

```
opstream& flush();
```

Description

Flushes the stream.

opstream::fwriteBytes

opstream class

Syntax

```
void fwriteBytes( const void *data, size_t sz );
```

Description

Writes the specified number of bytes (*sz*) from the supplied buffer (*data*) to the stream.

opstream::fwriteString

opstream class

Syntax

```
void fwriteString( const char *str );
```

Description

Writes the specified far character string (*str*) to the stream.

opstream::registerObject

opstream class

Syntax

```
void registerObject(TStreamableBase *adr);
```

Description

Registers the class of the object pointed to by *adr*.

opstream::registerVB

opstream class

Syntax

```
void registerVB(TStreamableBase *adr);
```

Description

Registers a virtual base class.

opstream::seekp

opstream class

Form1

```
opstream& seekp(streampos pos);
```

Form2

```
opstream& seekp(streamoff off,ios::seek_dir);
```

Description

Form 1: Moves the current position of the stream to the absolute position given by *pos*.

Form 2: Moves to a position relative to the current position by an offset *off* (+ or -) starting at *ios::seek_dir*. You can set *ios::seek_dir* to one if the following:

beg (start of stream)

cur (current stream position)

end (end of stream).

opstream::tellp

opstream class

Syntax

```
streampos tellp();
```

Description

Returns the (absolute) current stream position.

opstream::writeByte

opstream class

Syntax

```
void writeByte(uint8 ch);
```

Description

Writes the byte *ch* to the stream.

opstream::writeBytes

opstream class

Form 1

```
void writeBytes(const void *data, size_t sz);
```

Form 2

```
void writeBytes(const void far *data, size_t sz);
```

Description

Writes *sz* bytes from the *data* buffer to the stream.

opstream::writeObject

opstream class

Syntax

```
void writeObject( const TStreamableBase *t, int isPtr = 0, ModuleId mid =
GetModuleId() );
```

Description

Writes the object, pointed to by *t*, to the output stream. The *isPtr*argument indicates whether the object was allocated from the heap.

opstream::writeObjectPointer

opstream class

Syntax

```
void writeObjectPointer( const TStreamableBase *t, ModuleId mid =
GetModuleId() );
```

Description

Writes the object pointer *t* to the output stream.

opstream::writeString

opstream class

Syntax

```
void writeString(const char *str);
```

Description

Writes *str* to the stream.

opstream::writeWord

opstream class

Syntax

```
void writeWord(uint32 us);
```

Description

Writes the 32-bit word *us* to the stream.

opstream::writeWord16

opstream class

Syntax

```
void writeWord16(uint16 us);
```

Description

Writes the 16-bit word *us* to the stream.

opstream::writeWord32

opstream class

Syntax

```
void writeWord32(uint32 us);
```

Description

Writes the 32-bit word *us* to the stream.

opstream::writeData

See Also   opstream class

Syntax

```
void writeData(TStreamableBase *t);
```

Description

Writes data to the stream by calling the *write* member function of the appropriate class for the object being written.

See Also

opstream class

TStreamableBase

opstream::writePrefix

See Also opstream class

Syntax

```
void writePrefix(const TStreamableBase *t);
```

Description

Writes the class name prefix to the stream. The **<<** operator uses this function to write a prefix and suffix around the data written with writeData. The prefix/suffix is used to ensure type-safe stream I/O.

See Also
opstream class
ipstream::readPrefix

opstream::writeSuffix

See Also　　　opstream class

Syntax

```
void writeSuffix(const TStreamableBase *t);
```

Description

Writes the class name suffix to the stream. The << operator uses this function to write a prefix and suffix around the data written with writeData. The prefix/suffix is used to ensure type-safe stream I/O.

opstream::<<

opstream class

Syntax

```
friend opstream& operator << (opstream&  ps, signed char ch);
friend opstream& operator << (opstream&  ps, unsigned char ch);
friend opstream& operator << (opstream&  ps, signed short sh);
friend opstream& operator << (opstream&  ps, unsigned short sh);
friend opstream& operator << (opstream&  ps, signed int i);
friend opstream& operator << (opstream&  ps, unsigned int i);
friend opstream& operator << (opstream&  ps, signed long l);
friend opstream& operator << (opstream&  ps, unsigned long l);
friend opstream& operator << (opstream&  ps, float f);
friend opstream& operator << (opstream&  ps, double d);
friend opstream& operator << (opstream&  ps, long double d);
friend opstream& operator << (opstream&  ps, TStreamableBase& t);
```

Description

This friend operator of opstream inserts (writes) the given argument to the given ipstream object.

The data type of the argument determines the form of write operation employed.

pstream — opstream — ofpstream

pstream class

Hierarchy

Header File
objstrm.h

Syntax
```
class pstream
```

Description
Provides the base class for handling streamable objects.

Public Constructors
pstream::pstream

Protected Constructors
pstream::pstream

Public Data Members
PointerTypes

Protected Data Members
bp
state

Public Member Functions
bad
clear
eof
fail
good
rdbuf
rdstate

Protected Member Functions
init
setstate

Operators
void
!

pstream::pstream

pstream class

Form 1

```
pstream(streambuf *buf);
```

Form 2

```
pstream();  // PROTECTED
```

Description

Form 1: Creates a buffered *pstream* with the given buffer. The state is set to 0.

Form 2: Creates a *pstream* without initializing the buffer pointer *bp* or state. Use init to set the buffer and setstate to set the state.

pstream::bp

pstream class

Syntax

```
streambuf *bp;
```

Description

The *bp* data member is a pointer to the stream buffer.

pstream::state

pstream class

Syntax

```
int state;
```

Description

Formats state flags. Use rdstate to access the current state.

pstream::PointerTypes

pstream class

Syntax

```
enum PointerTypes {ptNull, ptIndexed, ptObject};
```

Description

Enumerates object pointer types.

pstream::bad

pstream class

Syntax

```
int bad() const;
```

Description

Returns nonzero if an error occurs.

pstream::clear

pstream class

Syntax

```
void clear(int aState = 0);
```

Description

Sets the stream state to the given value (defaults to 0).

pstream::eof

pstream class

Syntax

```
int eof() const;
```

Description

Returns nonzero on end of stream.

pstream::fail

pstream class

Syntax

```
int fail() const;
```

Description

Returns nonzero if a previous stream operation failed.

pstream::good

pstream class

Syntax

```
int good() const;
```

Description

Returns nonzero if no error states have been recorded for the stream (that is, no errors have occurred).

pstream::init

pstream class

Syntax

```
void init(streambuf *sbp);
```

Description

The init member function initializes the stream and sets state to 0 and bp to *sbp*.

pstream::operator void *()

See Also pstream class

Syntax

```
operator void * () const;
```

Description

Converts to a *void* pointer.

See Also
pstream::fail

pstream::operator ! ()

pstream class

Syntax

```
int operator ! () const;
```

Description

Overloads the NOT operator. Returns 0 if the operation has failed (that is, if pstream::fail returned nonzero); otherwise, returns nonzero.

pstream::rdbuf

Syntax

```
streambuf * rdbuf() const;
```

Description

Returns the *pb* pointer to the buffer assigned to the stream.

pstream::rdstate

pstream class

Syntax

```
int rdstate() const;
```

Description

Returns the current state value.

pstream::setstate

pstream class

Syntax

```
void setstate(int b);
```

Description

Updates the state data member with `state |= (b & 0xFF)`.

TStreamableBase class

Syntax

```
class _RTTI TStreamableBase
```

Header File

objstrm.h

Description

Classes that inherit from *TStreamableBase* are known as streamable classes (their objects can be written to and read from streams). If you develop your own streamable classes, make sure that *TStreamableBase* is somewhere in their ancestry.

Using an existing streamable class as a base is the easiest way to create a streamable class. If your class must also fit into an existing class hierarchy, you can use multiple inheritance to derive a class from *TStreamableBase* .

Type Definitions

Type_id

Member Functions

CastableID

FindBase

MostDerived

TStreamableBase::Type_id

TStreamableBase class

Syntax

```
typedef const char *Type_id;
```

Description

Describes type identifiers.

TStreamableBase::CastableID

TStreamableBase class

Syntax

```
virtual Type_id CastableID() const = 0Description
```

Description

Provides support for typesafe downcasting. Returns string containing the type name.

Note: This function is available only when the library is built without run-time type information (RTTI).

TStreamableBase::FindBase

TStreamableBase class

Syntax

```
virtual void *FindBase( Type_id id ) const;
```

Description

Returns a pointer to the base class.

Note: This function is available only when the library is built without run-time type information (RTTI).

TStreamableBase::MostDerived

TStreamableBase class

Syntax

```
virtual void *MostDerived() const = 0;
```

Description

Returns a **void** pointer to the actual streamed object.

Note: This function is available only when the library is built without run-time type information (RTTI).

TStreamableClass class

See Also

Syntax

```
class TStreamableClass : public ObjectBuilder
```

Header File

streambl.h

Description

TStreamableClass is used by the private database class and pstream in the registration of streamable classes.

Constructor

TStreamableClass::TStreamableClass

Friend Classes

ipstream

opstream

TStreamableClass::TStreamableClass

TStreamableClass class

Syntax

```
TStreamableClass(const char *n, BUILDER b, int d=NoDelta, ModuleId
  mid=GetModuleId());
```

Description

Creates a *TStreamableClass* object with the given name (*n*) and the given builder function (*b*), then registers the type.

For example, each streamable has a *Build* member function of type BUILDER. For type-safe object-stream I/O, the stream manager needs to access the names and the type information for each class. To ensure that the appropriate functions are linked into any application using the stream manager, you must provide a reference such as:

```
  TStreamableClass RegClassName;
```

where *TClassName* is the name of the class for which objects need to be streamed. (Note that *RegClassName* is a single identifier.) This not only registers *TClassName* (telling the stream manager which *Build* function to use), it also automatically registers any dependent classes. You can register a class more than once without any harm or overhead.

Invoke this function to provide raw memory of the correct size into which an object of the specified class can be read. Because the *Build* procedure invokes a special constructor for the class, all virtual table pointers are initialized correctly.

The distance, in bytes, between the base of the streamable object and the beginning of the *TStreamableBase* component of the object is *d*. Calculate *d* by using the _ _DELTA macro.

Example

```
TStreamableClass RegTClassName = TStreamableClass("TClassName",
  TClassName::build, _ _DELTA(TClassName));
```

See Also
TStreamableBase class

TStreamer class

Header File
objstrm.h

Syntax
```
class _RTTI TStreamer
```

Description
Provides a base class for all streamable objects.

Protected Constructor
TStreamer::TStreamer

Member Functions
GetObject
Read
StreamableName
Write

TStreamer::TStreamer

TStreamer class

Syntax

```
TStreamer( TStreamableBase *obj ) : object(obj) {} // PROTECTED
```

Description

Constructs the *TStreamer* object, and initializes the streamable object pointer.

TStreamer::GetObject

TStreamer class

Syntax

```
TStreamableBase *GetObject() const
```

Description

Returns the address of the TStreamableBase component of the streamable object.

TStreamer::Read

TStreamer class

Syntax

```
virtual void *Read( ipstream&, uint32 ) const = 0;
```

Description

This pure virtual member function must be redefined for every streamable class. It must read the necessary data members for the streamable class from the supplied ipstream.

TStreamer::StreamableName

TStreamer class

Syntax

```
virtual const char *StreamableName() const = 0;
```

Description

This pure virtual member function must be redefined for every streamable class. It returns the name of the streamable class, which is used by the stream manager to register the streamable class. The name returned must be a zero-terminated string.

TStreamer::Write

TStreamer class

Syntax

```
virtual void Write( opstream& ) const = 0;
```

Description

This pure virtual function must be redefined for every streamable class. It must write the necessary streamable class data members to the supplied opstream object. *Write* is usually implemented by calling the *Write* member function (if available) of a base class, and then inserting any additional data members for the derived class.

Streaming Macros

See Also

These macros are provided to simplify the declaration and definition of streamable classes.

DECLARE_STREAMABLE

DECLARE_STREAMABLE_FROM_BASE

DECLARE_ABSTRACT_STREAMABLE

DECLARE_STREAMER

DECLARE_STREAMER_FROM_BASE

DECLARE_ABSTRACT_STREAMER

DECLARE_CASTABLE

DECLARE_STREAMABLE_OPS

DECLARE_STREAMABLE_CTOR

IMPLEMENT_STREAMABLE

IMPLEMENT_STREAMABLE_CLASS

IMPLEMENT_STREAMABLE_CTOR

IMPLEMENT_STREAMABLE_POINTER

IMPLEMENT_CASTABLE_ID

IMPLEMENT_CASTABLE

IMPLEMENT_STREAMER

IMPLEMENT_ABSTRACT_STREAMABLE

IMPLEMENT_STREAMABLE_FROM_BASE

See Also
Persistent Streams

DECLARE_STREAMABLE macro

See Also Streaming Macros

Syntax

```
DECLARE_STREAMABLE(exp, cls, ver)
```

Header File

objstrm.h

Description

The DECLARE_STREAMABLE macro is used within a class definition to add the members that are needed for streaming. Since it contains access specifiers, it should be followed by an access specifier or be used at the end of the class definition.

- The first parameter should be a macro, which in turn should conditionally expand to either _ **_import** and _ **_export**, depending on whether the class is to be imported or exported from a DLL.
- The second parameter is the streamable class name.
- The third parameter is the object version number.

DECLARE_STREAMABLE_FROM_BASE macro

See Also   Streaming Macros

Header File

objstrm.h

Syntax

```
DECLARE_STREAMABLE_FROM_BASE(exp, cls, ver)
```

Description

This macro is used in the same way as DECLARE_STREAMABLE, but in the case where the class being defined can be written and read using *Read* and *Write* functions defined in its base class without change. This usually occurs when a derived class overrides virtual functions in its base or provides different constructors, but does not add any data members.

If you used DECLARE_STREAMABLE in this case, you would have to write *Read* and *Write* functions that merely called the base's *Read* and *Write* functions. Using DECLARE_STREAMABLE_FROM_BASE prevents this.

DECLARE_ABSTRACT_STREAMABLE macro

See Also Streaming Macros

Syntax

```
DECLARE_ABSTRACT_STREAMABLE(exp, cls, ver)
```

Header File

objstrm.h

Description

This macro is used in an abstract class. DECLARE_STREAMABLE doesn't work with an abstract class because an asbtract class can never be instantiated, and the code that attempts to instantiate the object (*Build*) causes compiler errors.

DECLARE_STREAMER macro

See Also   Streaming Macros

Syntax

```
DECLARE_STREAMER( exp, cls, ver )
```

Header File

objstrm.h

Description

This macro defines a nested class within your streamable class, and contains the core of the streaming code. DECLARE_STREAMER declares the *Read* and *Write* function declarations, whose definitions you must provide, and the *Build* function that calls the *TStreamableClass* constructor.

- The first parameter should be a macro, which in turn should conditionally expand to either **_ _import** and **_ _export**, depending on whether or not the class is to be imported or exported from a DLL.
- The second parameter is the streamable class name.
- The third parameter is the object version number.

DECLARE_STREAMER_FROM_BASE macro

See Also   Streaming Macros

Syntax

```
DECLARE_STREAMER_FROM_BASE( exp, cls, ver )
```

Header File

objstrm.h

Description

This macro is used by DECLARE_STREAMABLE_FROM_BASE. It declares a nested *Streamer* class without the *Read* and *Write* functions.

- The first parameter should be a macro, which in turn should conditionally expand to either **_ _import** and **_ _export**, depending on whether or not the class is to be imported or exported from a DLL.
- The second parameter is the streamable class name.
- The third parameter is the object version number.

DECLARE_ABSTRACT_STREAMER macro

See Also   Streaming Macros

Syntax

```
DECLARE_ABSTRACT_STREAMER ( exp, cls, ver )
```

Header File

objstrm.h

Description

This macro is used by DECLARE_ABSTRACT_STREAMABLE. It declares a nested *Streamer* class without the *Build* function.

- The first parameter should be a macro, which in turn should conditionally expand to either **_ _import** and **_ _export**, depending on whether or not the class is to be imported or exported from a DLL.
- The second parameter is the streamable class name.
- The third parameter is the object version number.

DECLARE_CASTABLE macro

See Also Streaming Macros

Syntax

```
DECLARE_CASTABLE
```

Header File

objstrm.h

Description

This macro provides declarations that provide a rudimentary typesafe downcast mechanism. This is useful for compilers that don't support run-time type information.

DECLARE_STREAMABLE_OPS macro

See Also Streaming Macros

Syntax

```
DECLARE_STREAMABLE_OPS(cls)
```

Header File

objstrm.h

Description

Declares the inserters and extractors. For template classes, DECLARE_STREAMABLE_OPS must use class<...> as the macro argument, other DECLAREs take only the class name.

DECLARE_STREAMABLE_CTOR macro

See Also Streaming Macros

Syntax

```
DECLARE_STREAMABLE_CTOR(cls)
```

Header File

objstrm.h

Description

Declares the constructor called by the *Streamer::Build* function.

IMPLEMENT_STREAMABLE macros

See Also Streaming Macros

Syntax

```
IMPLEMENT_STREAMABLE(cls)
IMPLEMENT_STREAMABLE1(cls, base1)
IMPLEMENT_STREAMABLE2(cls, base1, base2)
IMPLEMENT_STREAMABLE3(cls, base1, base2, base3)
IMPLEMENT_STREAMABLE4(cls, base1, base2, base3, base4)
IMPLEMENT_STREAMABLE5(cls, base1, base2, base3, base4, base5)
```

Header File

objstrm.h

Description

These macros generate the registration object for the class via IMPLEMENT_STREAMABLE_CLASS, and generate the various member functions that are needed for a streamable class via IMPLEMENT_ABSTRACT_STREAMABLE.

IMPLEMENT_STREAMABLE is used when the class has no base classes other than TStreamableBase. Its only parameter is the name of the class. The numbered versions (IMPLEMENT_STREAMABLE1, IMPLEMENT_STREAMABLE2, etc.) are for classes that have bases. Each base class, including all virtual bases, must be listed in the IMPLEMENT_STREAMABLE macro invocation.

The individual components comprising the above macros can be used separately for special situations, such as custom constructors.

IMPLEMENT_STREAMABLE_CLASS macro

See Also   Streaming Macros

Syntax

```
IMPLEMENT_STREAMABLE_CLASS(cls)
```

Header File

objstrm.h

Description

Constructs a TStreamableClass class instance.

IMPLEMENT_STREAMABLE_CTOR macros

See Also   Streaming Macros

Syntax

```
IMPLEMENT_STREAMABLE_CTOR(cls)
IMPLEMENT_STREAMABLE_CTOR1(cls, base1)
IMPLEMENT_STREAMABLE_CTOR2(cls, base1, base2)
IMPLEMENT_STREAMABLE_CTOR3(cls, base1, base2, base3)
IMPLEMENT_STREAMABLE_CTOR4(cls, base1, base2, base3, base4)
IMPLEMENT_STREAMABLE_CTOR5(cls, base1, base2, base3, base4, base5)
```

Header File

objstrm.h

Description

Defines the constructor called by the *Build* function. All base classes must be listed in the appropriate macro.

IMPLEMENT_STREAMABLE_POINTER macro

See Also Streaming Macros

Syntax

```
IMPLEMENT_STREAMABLE_POINTER(cls)
```

Header File

objstrm.h

Description

Creates the instance pointer extraction operator (>>).

IMPLEMENT_CASTABLE_ID macro

See Also Streaming Macros

Syntax

```
IMPLEMENT_CASTABLE_ID( cls )
```

Header File

objstrm.h

Description

Sets the typesafe downcast identifier.

IMPLEMENT_CASTABLE macros

See Also Streaming Macros

Syntax

```
IMPLEMENT_CASTABLE( cls )
IMPLEMENT_CASTABLE1( cls )
IMPLEMENT_CASTABLE2( cls )
IMPLEMENT_CASTABLE3( cls )
IMPLEMENT_CASTABLE4( cls )
IMPLEMENT_CASTABLE5( cls )
```

Header File

objstrm.h

Description

These macros implement code that supports the typesafe downcast mechanism.

IMPLEMENT_STREAMER macro

See Also Streaming Macros

Syntax

```
IMPLEMENT_STREAMER( cls )
```

Header File

objstrm.h

Description

Defines the *Streamer* constructor.

IMPLEMENT_ABSTRACT_STREAMABLE macros

See Also Streaming Macros

Syntax

```
IMPLEMENT_ABSTRACT_STREAMABLE( cls )
IMPLEMENT_ABSTRACT_STREAMABLE1( cls )
IMPLEMENT_ABSTRACT_STREAMABLE2( cls )
IMPLEMENT_ABSTRACT_STREAMABLE3( cls )
IMPLEMENT_ABSTRACT_STREAMABLE4( cls )
IMPLEMENT_ABSTRACT_STREAMABLE5( cls )
```

Header File

objstrm.h

Description

Expands to

IMPLEMENT_STREAMER (defines the *Streamer* constructor)

IMPLEMENT_STREAMABLE_CTOR (defines the TStreamableClass constructor)

IMPLEMENT_STREAMABLE_POINTER (defines the instance pointer extraction operator)

IMPLEMENT_STREAMABLE_FROM_BASE macro

See Also Streaming Macros

Syntax

```
IMPLEMENT_STREAMABLE_FROM_BASE( cls, base1 )
```

Header File

objstrm.h

Description

This macro expands to

IMPLEMENT_STREAMABLE_CLASS (constructs a TStreamableClass instance)

IMPLEMENT_STREAMABLE_CTOR1 (defines a one base class constructor that is called by *Build*)

IMPLEMENT_STREAMABLE_POINTER (defines the instance pointer extraction operator)

Mathematical Classes (C++)

See Also

The C++ mathematical classes provide mathematical operations that are available only in C++ programs. C++ programs, however, that use these classes, the numerical types that they define, or any of their friend and member functions can use any of the ANSI C standard mathematical routines.

These classes construct numerical types, define the functions used to carry out operations with their respective types (for example, converting to and from the BCD and complex type), and overload all necessary operators. These classes are independent of any hierarchy, but each class includes the iostream.h header file.

The C++ mathematical classes are:

bcd

complex

See Also
bcd
complex
real
iostream.h

bcd class

See Also Example Portability

Header File

bcd.h

Purpose

Creates binary-coded decimals (BCD) from integers or floating-point numerical types. The friend function real converts bcd numbers back to long double.

Constructors

bcd::bcd

Friend Functions

real

You can also use BCD numbers in any of the ANSI C standard math functions. The following ANSI C math functions are overloaded to operate with BCD types:

```
friend bcd  abs(bcd &);
friend bcd  acos(bcd &);
friend bcd  asin(bcd &);
friend bcd  atan(bcd &);
friend bcd  cos(bcd &);
friend bcd  cosh(bcd &);
friend bcd  exp(bcd &);
friend bcd  log(bcd &);
friend bcd  log10(bcd &);
friend bcd  pow(bcd & base, bcd & expon);
friend bcd  sin(bcd &);
friend bcd  sinh(bcd &);
friend bcd  sqrt(bcd &);
friend bcd  tan(bcd &);
friend bcd  tanh(bcd &);
```

Operators

The bcd class overloads the operators +, -, *, /, +=, -=, *=, /=, =, ==, and !=. These operators provide BCD arithmetic manipulation as when used with the standard mathematical functions.

The operators << and >> are overloaded for stream input and output of BCD numbers, as they are for other data types in iostream.h..

Range

The BCD numbers have about 17 decimal digits precision, and a range of 1×10^{-125} to 1×10^{125} (approximately).

Note: The number is rounded according to the rules of banker's rounding, which means round to nearest whole number, with ties being rounded to an even digit.

See Also

complex
real
iostream.h
Math Routines
Mathematical Classes

Portability

DOS	UNIX	Win16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+			+

bcd::bcd

bcd class

Form 1
```
bcd();
```

Form 2
```
bcd(int x);
```

Form 3
```
bcd(unsigned int x);
```

Form 4
```
bcd(long x);
```

Form 5
```
bcd(unsigned long x);
```

Form 6
```
bcd(double x, int decimals = Max);
```

Form 7
```
bcd(long double x, int decimals = Max);
```

Description

Once you construct BCD numbers, you can freely mix them in expressions with ints, doubles, and other numeric types.

Form 1: Default used to declare a variable of type BCD.

```
bcd i;       // Construct a bcd-type number.
bcd j = 37;  // Construct and initialize a bcd-type number.
```

Form 2: Defines a BCD variable from an int variable or directly from an integer. This example produces i = 15, j = 15, k = 12.

```
int i = 15;
bcd j = bcd(i);   // Initialize j with a previously declared type.
bcd k = bcd(12);  // Construct k from the integer provided.
```

Form 3: Defines a BCD variable from one that was previously declared to be an unsigned int type. An unsigned integer can be provided directly to the constructor.

Form 4: Defines a BCD variable from an long variable or directly from a long value.

Form 5: Defines a BCD variable from one that was previously declared to be an unsigned long type.

Form 6: Defines a BCD variable from one that was previously declared to be a floating point double type. The constructor also creates a variable directly from a double value.

To specify a precision level (that is, the number of digits after the decimal point) that is different from the default, use the variable decimals. For example,

```
double x = 1.2345;  // Declare and initialize in the usual manner.
bcd y = bcd(x, 2);  // Create a bcd numerical type from x.
```

The precision level for y is set to 2. Therefore, y is initialized with 1.23.

Form 7: Defines a BCD variable from one that was previously declared to be a floating point long double type. Alternately, you can supply a long double value directly in the place of x.

To specify a precision level (that is, the number of digits after the decimal point) that is different from the default, use the variable decimals.

/* bcd and complex example */

```
/* Show that an ANSI math function can handle the bcd, */
/*  complex, and double types. Use the tan() function. */
#include <bcd.h>
#include <complex.h>

void main(void) {
    double PI = 3.1416;         /* Use to define radian angles. */

    double  x = (PI * 0.250);  /* 45 degree angle approximation. */
    bcd     y = bcd(x);
    complex z = complex(x);

    cout  << " double x = "  << x << "\t\t tan(x) = " << tan(x)
          << "\n bcd y = "     << y << "\t\t\t tan(y) = " << tan(y)
          << "\n complex z = " << z << "\t tan(z) = " << tan(z);
   }

/*
```

Program Output

```
double x = 0.7854          tan(x) = 1.000004
bcd y = 0.7854             tan(y) = 1.000004
complex z = (0.7854, 0)    tan(z) = (1.000004, 0)
*/
```

complex class

See Also　　Example　　Portability

Header File

complex.h

Purpose

Creates complex numbers. The real function converts complex numbers back to long double. The friend function returns the real part of a complex number or converts a complex number back to double. The data associated to a complex number consists of two floating-point numbers. real returns the one considered to be the real part.

Constructors

complex::complex

Friend Functions

You can also use complex numbers in any of the ANSI C standard mathematical functions.

abs	log
acos	log10
arg	polar
asin	pow
atan	real
conj	sin
cos	sinh
cosh	sqrt
exp	tan
imag	tanh
norm	

Operators

The complex class overloads the operators +, -, *, /, +=, -=, *=, /=, =, ==, and !=. These operators provide complex arithmetic manipulation in the usual sense.

The operators << and >> are overloaded for stream input and output of complex numbers, as they are for other data types in iostream.h.

See Also

bcd

real

math.h

iostream.h.

Mathematical Classes

complex::complex

complex class

Form 1

```
complex();
```

Form 2

```
complex(double real, double imag = 0);
```

Description

Once you construct complex numbers, you can freely mix them in expressions with ints, doubles, and other numeric types.

Note: If you do not want to program in C++, but instead want to program in C, the only constructs available to you are struct complex and cabs, which give the absolute value of a complex number. Both of these alternates are defined in math.h.

Form 1: The default typically used to declare a variable of type complex. For example:

```
complex i;       /* Construct a complex-type number. */
complex j = 37;  /* Construct and initialize a complex-type number. */
```

Form 2: Creates a complex numerical type out of a double. Upon construction, a real and an imaginary part are provided. The imaginary part is taken to be zero if imag is omitted.

complex abs

See Also Portability Mathematical Classes

Syntax

```
friend double abs(complex& val);
```

Description

Returns the absolute value of a complex number.

The complex version of abs returns a double. All other math functions return a complex type when val is complex type.

complex acos

See Also Portability Mathematical Classes

Syntax

```
friend complex acos(complex& z);
```

Description

Calculates the arc cosine.

The complex inverse cosine is defined by:

```
acos(z) = -i * log(z + i sqrt(1 - z2))
arg
```

arg

See Also Example Portability Mathematical Classes

Syntax

```
double arg(complex x);
```

Description

arg gives the angle, in radians, of the number in the complex plane.

The positive real axis has angle 0, and the positive imaginary axis has angle pi/2. If the argument passed to arg is complex 0 (zero), arg returns zero.

`arg(x)` returns `atan2(imag(x), real(x))`.

/* arg example */

```
// Illustrate the use of each of the complex friend functions.
#include <complex.h>  // This also includes iostream.h.

int main(void)
{
   complex z(3.1, 4.2);
   cout << "z = " << z << "\n";
   cout << "  has real part = " << real(z) << "\n";
   cout << "  and imaginary part = " << imag(z) << "\n";
   cout << "z has complex conjugate = " << conj(z) << "\n";

   double mag = sqrt(norm(z));
   double ang = arg(z);

   cout << "The polar form of z is:\n";
   cout << "   magnitude = " << mag << "\n";
   cout << "   angle (in radians) = " << ang << "\n";
   cout << "Reconstructing z from its polar form gives:\n";
   cout << "   z = " << polar(mag,ang) << "\n";
   return 0;
}
```

Program Output

```
z = (3.1, 4.2)
  has real part = 3.1
  and imaginary part = 4.2
z has complex conjugate = (3.1, -4.2)
The polar form of z is:
   magnitude = 5.220153
   angle (in radians) = 0.934958
Reconstructing z from its polar form gives:
   z = (3.1, 4.2)
```

complex asin

See Also   Portability   Mathematical Classes

Syntax

```
friend complex asin(complex& z);
```

Description

Calculates the arc sine.

The complex inverse sine is defined by

```
asin(z) = -i * log(i * z + sqrt(1 - z2 ))
atan
```

complex atan

See Also Portability Mathematical Classes

Syntax

```
friend complex atan(complex& z);
```

Calculates the arc tangent.

Description

Calculates the arc tangent.

The complex inverse tangent is defined by

```
atan(z) = -0.5 i log((1 + i z)/(1 - i z))
```

conj

See Also Portability Mathematical Classes

Syntax

```
complex conj(complex z);
```

Description

Returns the complex conjugate of a complex number.

`conj(z)` is the same as `complex(real(z), -imag(z))`.

complex cos

See Also Portability Mathematical Classes

Syntax

```
friend complex cos(complex& z);
```

Description

Calculates the cosine of a value.

The complex cosine is defined by

```
cos(z) = ( exp(i * z) + exp(-i * z) ) / 2
```

complex cosh

See Also Portability Mathematical Classes

Syntax

```
friend complex cosh(complex& z);
```

Description

Calculates the hyperbolic cosine of a value.

The complex hyperbolic cosine is defined by

```
cosh(z) = ( exp(z) + exp(-z) ) / 2
exp
```

complex exp

See Also Portability Mathematical Classes

Syntax

```
friend complex exp(complex& y);
```

Description

Calculates the exponential e to the y.

The complex exponential function is defined by

```
exp(x + y * i) = exp(x) (cos(y) + i * sin(y) )
```

imag

See Also   Portability   Mathematical Classes

Syntax

```
double imag(complex x);
```

Description

Returns the imaginary part of a complex number.

The data associated to a complex number consists of two floating-point (double) numbers. imag returns the one considered to be the imaginary part.

complex log

See Also Portability Mathematical Classes

Syntax

```
friend complex log(complex& z);
```

Description

Calculates the natural logarithm of z.

The complex natural logarithm is defined by

```
log(z) = log( abs(z) ) + i * arg(z)
```

complex log10

See Also Portability Mathematical Classes

Syntax

```
friend complex log10(complex& z);
```

Description

Calculates log10(z).

The complex common logarithm is defined by

```
log10(z) = log(z) / log(10)
```

norm

See Also Portability Mathematical Classes

Syntax

```
double norm(complex x);
```

Description

Returns the square of the absolute value. norm(x) returns the magnitude real(x) * real(x) + imag(x) * imag(x).

norm can overflow if either the real or imaginary part is sufficiently large.

polar

See Also Portability Mathematical Classes

Syntax

```
complex polar(double mag, double angle = 0);
```

Description

Returns a complex number with a given magnitude (absolute value) and angle.

`polar(mag, angle)` is the same as `complex(mag * cos(angle), mag * sin(angle))`.

complex pow

See Also Portability Mathematical Classes

Syntax

```
friend complex pow(complex& base, double expon);
```

Description

```
friend complex pow(double base, complex& expon);
friend complex pow(complex& base, complex& expon);
```

Calculates base to the power of expon.

The complex pow is defined by

```
pow(base, expon) = exp(expon * log(base))
```

real

See Also   Examples   Portability   Mathematical Classes

Syntax

```
long double real(bcd number);
double real(complex x);
```

Description

Converts a bcd or complex number back to a long double or returns the real part of complex number. The data associated to a complex number consists of two floating-point numbers; real returns the one considered to be the real part.

See Also

bcd
complex
Math Routines
Mathematical Classes

Real Examples

bcd version

complex version

/* real example for bcd */

```
/* Construct a bcd number and restrict its precision. */
#include <bcd.h> /* The iostream header is also included. */

int main(void)
{
   int k = 0;
   double x = 10000.0; /* ten thousand dollars */
   bcd a = bcd(x/3, 2); /* a third, rounded to nearest penny */

   cout << "Share of fortune = $" << a << "\n";

   k = real(a);  /* Now, assign bcd to an int. */
   cout << "The integer version of variable a:" << k;
   return 0;
}
```

Program Output

```
Share of fortune = $3333.33
The integer version of variable a:3333
```

/* real example for complex */

```
/* Show the components of a complex number. */
#include <complex.h>  /* This also includes iostream.h. */

int main(void)
{
   complex z(3.1, 4.2);
   cout << "z = " << z << "\n";
   cout << "  has real part = " << real(z) << "\n";
   cout << "  and imaginary real part = " << imag(z) << "\n";
   cout << "z has complex conjugate = " << conj(z) << "\n";
   return 0;
}
```

Program Output

```
z = (3.1, 4.2)
  has real part = 3.1
  and imaginary real part = 4.2
```

z has complex conjugate = (3.1, -4.2)

complex sin

See Also Portability Mathematical Classes

Syntax

```
friend complex sin(complex& z);
```

Description

Calculates the trignometric sine.

The complex sine is defined by

```
sin(z) = ( exp(i * z) - exp(-i * z) ) / (2 * i)
```

complex sinh

See Also Portability Mathematical Classes

Syntax

```
friend complex sinh(complex& z);
```

Description

Calculates the hyperbolic sine.

The complex hyperbolic sine is defined by

```
sinh(z) = ( exp(z) - exp(-z) ) / 2
sqrt
```

complex sqrt

See Also Portability Mathematical Classes

Syntax

```
friend complex sqrt(complex& x);
```

Description

Calculates the positive square root.

For any complex number x, sqrt(x) gives the complex root whose arg is arg(x)/2.

The complex square root is defined by

```
sqrt(x) = sqrt(abs(x)) (cos( arg(x) / 2) + i * sin(arg(x)/2))
```

complex tan

See Also Portability Mathematical Classes

Syntax

```
friend complex tan(complex& z);
```

Description

Calculates the trignometric tangent.

The complex tangent is defined by

```
tan(z) = sin(z) / cos(z)
```

complex tanh

See Also   Portability   Mathematical Classes

Syntax

```
friend complex tanh(complex& z);
```

Description

Calculates the hyperbolic tangent.

The complex hyperbolic tangent is defined by

```
tanh(z) = sinh(z) / cosh(z)
```

See Also

complex

Math Routines

Mathematical Classes

bcd.h

See Also   Header Files

Declares the C++ class bcd, plus the overloaded operators for class bcd and for BCD math functions.

Functions

abs
acos
asin
atan
cos
cosh
exp
log
log10
pow
pow10
real
sin
sinh
sqrt
tan
tanh

Constants, Data Types and Global Variables

_BCD_H
_BcdMaxDecimals
bcdexpo (enum)
__cplusplus

Overloaded Operators

!=	+=	+
-=	-	*=
*	==	<
<=	>	>=
/=	/	

complex.h

See Also Header Files

Declares the C++ complex math functions.

All function names, member names, and operators have been borrowed from AT&T C++, except for the addition of acos, asin, atan, log10, tan, and tanh.

Includes

IOSTREAM.H

MATH.H

Functions

abs

acos

arg

asin

atan

conj

cos

cosh

exp

imag

log

log10

norm

polar

pow

pow10

real

sin

sinh

sqrt

tan

tanh

Constants, Data Types and Global Variables

_COMPLEX_H

__cplusplus

Overloaded Operators

+	+=	-	-=
*	*=	/	/=
==	!=	<<	>>

Run-Time Support

These topics provide a detailed description of the functions and classes that provide run-time support. Any class operators or member functions are listed immediately after the class constructor.

Classes

Bad_cast class
Bad_typeid class
typeinfo class
xalloc class
xmsg class

Functions

set_new_handler
set_terminate
set_unexpected
terminate
unexpected

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+		+	

See Also

C++ Exception Handling

Run-time Support

See Also
Run-time Support

Bad_cast class

See Also Portability Example

Header File
typeinfo.h

Description
When dynamic_cast fails to make a cast to reference, the expression can throw *Bad_cast*. Note that when *dynamic_cast* fails to make a cast to pointer type, the result is the null pointer.

Bad_typeid class

See Also   Portability   Example

Header File

typeinfo.h

Description

When the operand of typeid is a dereferenced null pointer, the *typeid* operator can throw *Bad_typeid*.

set_new_handler

See Also   Portability

Header File
new.h

Syntax

```
typedef void (new * new_handler)() throw(xalloc);
new_handler set_new_handler(new_handler my_handler);
```

Description

set_new_handler installs the function to be called when the global operator new() or operator *new*[]() cannot allocate the requested memory. By default the *new* operators throw an xalloc exception if memory cannot be allocated. You can change this default behavior by calling *set_new_handler* to set a new handler. To retain the traditional version of *new*, which does not throw exceptions, you can use *set_new_handler*(0).

If new cannot allocate the requested memory, it calls the handler that was set by a previous call to *set_new_handler*. If there is no handler installed by *set_new_handler*, *new* returns 0. *my_handler* should specify the actions to be taken when new cannot satisfy a request for memory allocation. The new_handler type, defined in new.h, is a function that takes no arguments and returns void. A *new_handler* can throw an *xalloc* exception.

The user-defined *my_handler* should do one of the following:

- return after freeing memory
- throw an *xalloc* exception or an exception derived from *xalloc*
- call abort or exit functions

If *my_handler* returns, then *new* will again attempt to satisfy the request.

Ideally, *my_handler* would free up memory and return. *new* would then be able to satisfy the request and the program would continue. However, if *my_handler* cannot provide memory for *new*, *my_handler* must throw an exception or terminate the program. Otherwise, an infinite loop will be created.

Preferably, you should overload operator *new*() and operator *new*[]() to take appropriate actions for your applications.

Return Value

set_new_handler returns the old handler, if one has been registered.

The user-defined argument function, *my_handler*, should not return a value.

Portability

DOS	UNIX	Win 16	Win 32	ANSI C	ANSI C++	OS/2
+		+	+		+	+

set_terminate

See Also Portability

Header File

except.h

Syntax

```
typedef void (*terminate_function)();
terminate_function set_terminate(terminate_function t_func);
```

Description

set_terminate lets you install a function that defines the program's termination behavior when a handler for the exception cannot be found. The actions are defined in *t_func*, which is declared to be a function of type *terminate_function*. A *terminate_function* type, defined in except.h, is a function that takes no arguments, and returns void.

By default, an exception for which no handler can be found results in the program calling the terminate function. This will normally result in a call to abort. The program then ends with the message `Abnormal program termination`. If you want some function other than *abort* to be called by the *terminate* function, you should define your own *t_func* function. Your *t_func* function is installed by *set_terminate* as the termination function. The installation of *t_func* lets you implement any actions that are not taken by *abort*.

Return Value

The previous function given to *set_terminate* will be the return value.

The definition of *t_func* must terminate the program. Such a user-defined function must not return to its caller, the *terminate* function. An attempt to return to the caller results in undefined program behavior. It is also an error for *t_func* to throw an exception.

See Also

[abort]
[C++ Exception Handling]
[Run-time Support]
[set_unexpected]
[terminate]

set_unexpected

See Also Portability

Header File

except.h

Syntax

```
typedef void ( * unexpected_function )();
unexpected_function set_unexpected(unexpected_function unexpected_func);
```

Description

set_unexpected lets you install a function that defines the program's behavior when a function throws an exception not listed in its exception specification. The actions are defined in *unexpected_func*, which is declared to be a function of type *unexpected_function*. An *unexpected_function* type, defined in except.h, is a function that takes no arguments, and returns void.

By default, an unexpected exception causes unexpected to be called. If is defined, it is subsequently called by *unexpected*. Program control is then turned over to the user-defined *unexpected_func*. Otherwise, terminate is called.

Return Value

The previous function given to *set_unexpected* will be the return value.

The definition of *unexpected_func* must not return to its caller, the *unexpected* function. An attempt to return to the caller results in undefined program behavior.

unexpected_func can also call abort, exit, or *terminate*.

See Also

abort

C++ Exception Handling

exit

Run-time Support

set_terminate

terminate

terminate

See Also Portability

Header File
except.h

Syntax
```
void terminate();
```

Description
The function *terminate* can be called by unexpected or by the program when a handler for an exception cannot be found. The default action by *terminate* is to call abort. Such a default action causes immediate program termination.

You can modify the way that your program will terminate when an exception is generated that is not listed in the exception specification. If you do not want the program to terminate with a call to *abort*, you can instead define a function to be called. Such a function (called a *terminate_function*) will be called by *terminate* if it is registered with set_terminate.

Return Value
None.

typeinfo class

See Also Portability Example

Header File
typeinfo.h

Description
Provides information about a type.

Constructor
Only a private constructor is provided. You cannot create *typeinfo* objects. By declaring your objects to be __rtti types, or by using the -RT compiler switch, the compiler provides your objects with the elements of *typeinfo*. *typeinfo* references are generated by the typeid operator.

Public Member Functions
name

before

Operators
==

!=

typeinfo::name

typeinfo class

Syntax

```
const char* name() const;
```

Description

This function returns a printable string that identifies the type *name* of the operand to typeid. The space for the character string is overwritten on each call.

typeinfo::before

typeinfo class

Syntax

```
int before(const typeinfo&);
```

Description

Use this function to compare the lexical order of types. For example, to compare two types, *T1* and *T2*, use the following syntax:

```
typeid ( T1 ).before(typeid( T2 ));
```

The *before* function returns 0 or 1.

typeinfo::operator ==

typeinfo class

Syntax

```
int operator==(const typeinfo &) const;
```

Description

Provides comparison of *typeinfos*.

typeinfo::operator !=

typeinfo class

Syntax

```
int operator!=(const typeinfo &) const;
```

Description

Provides comparison of *typeinfos*.

// Example

```
// HOW TO GET RUNTIME TYPE INFORMATION.
#include <iostream.h>
#include <typeinfo.h>

class __rtti Alpha {
   virtual void func() {};  // This makes Alpha a polymorphic class type.
};

class B : public Alpha {};

int main(void) {
   B Binst;           // Instantiate class B
   B *Bptr;           // Declare a B-type pointer
   Bptr = &Binst;     // Initialize the pointer

   // THESE TESTS ARE DONE AT RUNTIME

   if (typeid( *Bptr ) == typeid( B ) )
       // Ask "WHAT IS THE TYPE FOR *Bptr?"
       cout << "Name is " << typeid( *Bptr).name();

   if (typeid( *Bptr ) != typeid( Alpha ) )
       cout << "\nPointer is not an Alpha-type.";

   return 0;
   }
```

// Program Output

```
// Name is B
// Pointer is not an Alpha-type.
```

unexpected

See Also Portability

Header File
except.h

Syntax

```
void unexpected();
```

Description

The *unexpected* function is called when a function throws an exception not listed in its exception specification. The program calls *unexpected*, which by default calls any user-defined function registered by set_unexpected. If no function is registered with *set_unexpected*, the *unexpected* function then calls terminate.

Return Value

None, although *unexpected* may throw an exception.

xalloc class

See Also Portability

Header File

except.h

Description

Reports an error on allocation request.

Constructor

xalloc::xalloc

Public Member Functions

raise

requested

xalloc::xalloc

xalloc class

Syntax

```
xalloc(const string &msg, size_t size);
```

Description

The *xalloc* class has no default constructor. Every use of *xalloc* must define the message to be reported when a size allocation cannot be fulfilled. The string type is defined in cstring.h header file.

xalloc::raise

xalloc class

Syntax

```
void raise() throw(xalloc);
```

Description

Calling *raise* causes an *xalloc* to be thrown. In particular, it throws *this.

xalloc::requested

xalloc class

Syntax

```
size_t requested() const;
```

Description

Returns the number of bytes that were requested for allocation.

xmsg class

See Also   Portability

Header File
except.h

Description
Reports a message related to an exception.

Constructor
xmsg::xmsg

Public Member Functions
raise

why

xmsg::xmsg

xmsg class

Syntax

```
xmsg(string msg);
```

Description

There is no default constructor for *xmsg*. Every *xmsg* object must have a string message explicitly defined. The string type is defined in cstring.h header file.

xmsg::raise

xmsg class

Syntax

```
void raise() throw(xmsg);
```

Description

Calling *raise* causes an *xmsg* to be thrown. In particular, it throws *this.

xmsg::why

xmsg class

Syntax

```
const string _FAR & why() const;
```

Description

Reports the string used to construct an *xmsg*. Because every *xmsg* must have its message explicitly defined, every instance should have a unique message.

except.h

See Also Header Files

The except.h header file contains the declarations and prototypes for exception-handling functions and classes, their data members, and member functions.

Includes

STDLIB.H

Classes

xalloc class

xmsg class

Functions

set_terminate

set_unexpected

terminate

unexpected

typeinfo.h

See Also Header Files

The typeinfo.h header file contains the declarations and prototypes for the run-time type information classes, their data members, and member functions.

Classes

Bad_cast class

Bad_typeid class

typeinfo class

Class Diagnostic Macros

See Also

Borland provides a set of macros for debugging C++ code. They are located in checks.h. There are two types of macros, *default* and *extended*.

The default macros are

- CHECK
- PRECONDITION
- TRACE
- WARN

The extended macros are

- CHECKX
- PRECONDITIONX
- TRACEX
- WARNX

See Also

Extended Diagnostic Macros
Macro Message Output
Run-time Macro Control
Using Preprocessor Symbols

Default Macros

Default macros provide straightforward value checking and message output.

Using Preprocessor Symbols

See Also

Three preprocessor symbols control diagnostic macro expansion: _ _DEBUG, _ _TRACE, and _ _WARN. If one of these symbols is defined when compiling, then the corresponding macros expand and diagnostic code is generated. If none of these symbols is defined, then the macros do not expand and no diagnostic code is generated. These symbols can be defined on the command line using the -D switch, or by using #define statements within your code.

	__DEBUG=1	__DEBUG=2	__TRACE	__WARN
PRECONDITION	X	X		
PRECONDITIONX	X	X		
CHECK		X		
CHECKX		X		
TRACE			X	
TRACEX			X	
WARN				X
WARNX				X

To create a diagnostic version of an executable, place the diagnostic macros at strategic points within the program code and compile with the appropriate preprocessor symbols defined. Diagnostic versions of the Borland class libraries are built in a similar manner.

See Also

Extended Diagnostic Macros
Macro Message Output
Run-time Macro Control

CHECK

See Also

Syntax

```
CHECK(<cond>)
```

Header File

checks.h

Description

Outputs <msg> and throws an exception if <cond> equals 0. Use CHECK to perform value checking within a function.

See Also
CHECKX

PRECONDITION

See Also

Syntax

```
PRECONDITION(<cond>)
```

Header File

checks.h

Description

Outputs <msg> and throws an exception if <cond> equals 0. Use PRECONDITION on entry to a function to check the validity of the arguments and to do any other checking to determine if the function has been invoked correctly.

See Also
PRECONDITIONX

TRACE

See Also Example Class Diagnostic Macros

Syntax

```
TRACE(<msg>)
```

Header File

checks.h

Description

Outputs <msg>. TRACE is used to output general messages that are not dependent upon a particular condition.

See Also
TRACEX

WARN

See Also Example Class Diagnostic Macros

Syntax

```
WARN(<cond>,<msg>)
```

Header File

checks.h

Description

Outputs <msg> if <cond> is nonzero. It is used to output conditional messages.

See Also
WARNX

TRACE and WARN Example

The following program illustrates the use of the default TRACE and WARN macros:

```
#include <checks.h>

int main()
{
   TRACE( "Hello World" );
   WARN( 5 != 5, "Math is broken!" );
   WARN( 5 != 7, "Math still works!" );

   return 0;
}
```

When the above code is compiled with __TRACE and __WARN defined, it produces the following output when run:

```
Trace PROG.C 5: [Def] Hello World
Warning PROG.C 7: [Def] Math still works!
```

The above output indicates that the message "Hello World" was output by the default TRACE macro on line 5 of PROG.C, and the message "Math still works!" was output by the default WARN macro on line 7 of PROG.C.

Default diagnostic macros expand to extended diagnostic macros with the group set to "Def" and the level set to 0. This "Def" group controls the behavior of the default macros and is initially enabled with a threshold level of 0.

CHECKX

See Also   Class Diagnostic Macros

Syntax

```
CHECKX(<cond>,<msg>)
```

Header File

checks.h

Description

Outputs <msg> and throws an exception if <cond> equals 0. Use CHECKX to perform value checking within a function.

See Also
CHECK

PRECONDITIONX

See Also Class Diagnostic Macros

Syntax

```
PRECONDITIONX(<cond>,<msg>)
```

Header File

checks.h

Description

Outputs <msg> and throws an exception if <cond> equals 0. Use PRECONDITIONX on entry to a function to check the validity of the arguments and to do any other checking to determine if the function has been invoked correctly.

See Also
PRECONDITION

TRACEX

See Also Example Class Diagnostic Macros

Syntax

```
TRACEX(<group>,<level>,<msg>)
```

Header File

checks.h

Description

Trace only if <group> and <level> are enabled.

See Also

TRACE
DIAG_DECLARE_GROUP
DIAG_DEFINE_GROUP
DIAG_ENABLE
DIAG_ISENABLED
DIAG_SETLEVEL
DIAG_GETLEVEL

WARNX

See Also Class Diagnostic Macros

Syntax

```
WARNX(<group>,<cond>,<level>,<msg>)
```

Header File

checks.h

Description

Warn only if <group> and <level> are enabled.

See Also

WARN

DIAG_DECLARE_GROUP

DIAG_DEFINE_GROUP

DIAG_ENABLE

DIAG_ISENABLED

DIAG_SETLEVEL

DIAG_GETLEVEL

DIAG_DECLARE_GROUP

See Also   Class Diagnostic Macros

Syntax

```
DIAG_DECLARE_GROUP(<name>)
```

Header File

checks.h

Description

Declare a group named <name>. You cannot use DIAG_DECLARE_GROUP and DIAG_DEFINE_GROUP in the same compilation unit.

See Also
DIAG_DEFINE_GROUP

DIAG_DEFINE_GROUP

See Also   Class Diagnostic Macros

Syntax

```
DIAG_DEFINE_GROUP(<name>,<enabled>,<level>)
```

Header File

checks.h

Description

Define a group named <name>. You cannot use DIAG_DECLARE_GROUP and DIAG_DEFINE_GROUP in the same compilation unit.

See Also
DIAG_DECLARE_GROUP

See Also

DIAG_DECLARE_GROUP

DIAG_DEFINE_GROUP

DIAG_ENABLE

See Also   Class Diagnostic Macros

Syntax

```
DIAG_ENABLE(<group>,<state>)
```

Header File

checks.h

Description

Sets <group>'s enable flag to <state>.

See Also

DIAG_DECLARE_GROUP

DIAG_DEFINE_GROUP

DIAG_ISENABLED

DIAG_ISENABLED

See Also   Class Diagnostic Macros

Syntax

```
DIAG_ISENABLED(<group>)
```

Header File

checks.h

Description

Returns nonzero if <group> is enabled.

See Also

DIAG_DECLARE_GROUP

DIAG_DEFINE_GROUP

DIAG_ENABLE

DIAG_SETLEVEL

See Also   Class Diagnostic Macros

Syntax

```
DIAG_SETLEVEL(<group>,<level>)
```

Header File

checks.h

Description

Sets <group>'s threshold level to <level>.

See Also

DIAG_DECLARE_GROUP
DIAG_DEFINE_GROUP
DIAG_ENABLE
DIAG_ISENABLED
DIAG_GETLEVEL

DIAG_GETLEVEL

See Also Class Diagnostic Macros

Syntax

```
DIAG_GETLEVEL(<group>)
```

Header File

checks.h

Description

Gets <group>'s threshold level.

Threshold levels are arbitrary numeric values that establish a threshold for enabling macros. A macro with a level greater than the group threshold level will not be executed. For example, if a group has a threshold level of 0 (the default value), all macros that belong to that group and have levels of 1 or greater are ignored.

See Also

DIAG_DECLARE_GROUP

DIAG_DEFINE_GROUP

DIAG_ENABLE

DIAG_ISENABLED

DIAG_SETLEVEL

Extended Diagnostics Macros Example

The following PROG.C example defines two diagnostic groups, Group1 and Group2, which are used as arguments to extended diagnostic macros:

```
#include <checks.h>

DIAG_DEFINE_GROUP(Group1,1,0);
DIAG_DEFINE_GROUP(Group2,1,0);

void    main( int argc, char **argv )
{
   TRACE( "Always works, argc=" << argc );

   TRACEX( Group1, 0, "Hello" );
   TRACEX( Group2, 0, "Hello" );

   DIAG_DISABLE(Group1);

   TRACEX( Group1, 0, "Won't execute - group is disabled!" );
   TRACEX( Group2, 3, "Won't execute - level is too high!" );
}
```

When the above code is compiled with _ _TRACE defined and run, it produces the following output:

```
Trace PROG.C 8: [Def] Always works, argc=1
Trace PROG.C 10: [Group1] Hello
Trace PROG.C 11: [Group2] Hello
```

Note that the last two macros are not executed. In the first case, the group Group1 is disabled. In the second case, the macro level exceeds Group2's threshold level (set by default to 0).

Extended Diagnostic Macros

See Also Class Diagnostic Macros

The extended macros CHECKX and PRECONDITIONX augment CHECK and PRECONDITION by letting you provide a message to be output when the condition fails.

The extended macros TRACEX and WARNX augment TRACE and WARN by providing a way to specify macro groups that can be independently enabled or disabled. TRACEX and WARNX require additional arguments that specify the group to which the macros belongs, and the threshold level at which the macro should be executed. The macro is excuted only if the specified group is enabled and has a threshold level which is greater than or equal to the threshhold level argument used in the macro.

See Also

Macro Message Output

Run-time Macro Control

Using Preprocessor Symbols

Macro Message Output

See Also   Class Diagnostic Macros

The TRACE, TRACEX, WARN, and WARNX macros take a <msg> argument that is conditionally inserted into an output stream. This means a sequence of objects can be inserted in the output stream (for example `TRACE( "Mouse @ " << x << "," << y );`). The use of streams is extensible to different object types and allows for parameters within trace messages.

Diagnostic macro message output can be viewed while the program is running. If the target environment is Windows, the output is sent to the `OutputDebugString` function, and can be viewed with the DBWIN.EXE or OX.SYS utilities. If Turbo Debugger is running, the output will be sent to its log window. If the target environment is DOS, the output is sent to the standard output stream and can be easily redirected at the command line.

See Also

Extended Diagnostic Macros

Run-time Macro Control

Using Preprocessor Symbols

Run-Time Macro Control

See Also Class Diagnostic Macros

Diagnostic groups can be controlled at runtime by using the control macros within your program or by directly modifying the group information within the debugger.

This group information is contained in a template class named TDiagGroup<TDiagGroupClass##Group>, where ##Group is the name of the group. This class contains a static structure Flags, which in turn contains the enabled flag and the threshold level. For example, to enable the group Group1, you would set the variable `TDiagGroup<TDiagGroupClassGroup1>::Flags.Enabled` to 1.

See Also

Extended Diagnostic Macros

Macro Message Output

Using Preprocessor Symbols

checks.h

See Also Header Files

The checks.h header file contains the declarations and prototypes for the class diagnostic macros.

Includes

CSTRING.H

EXCEPT.H

STRSTREA.H

SYS\TYPES.H

Macros

CHECK

CHECKX

PRECONDITION

PRECONDITIONX

TRACE

TRACEX

WARN

WARNX

See Also
Precompiled Headers

Service Classes (C++)

These are the C++ service classes you can use for accessing and manipulating time, date, file, and thread classes.

Class Type	Header File
Date class	date.h
File class	file.h
String classes	cstring.h, regexp.h
Thread classes	thread.h
Time class	time.h

The header files for these classes are found in either `\BC4\INCLUDE` or `\BC4\INCLUDE\CLASSLIB`.

Thread classes

These are the thread-related utility classes:

TCriticalSection class

TMutex class

TSync class

TThread class

String classes (C++)

These are the string-related classes:

string class

TSubstring class

TRegexp class (helper class)

classlib\date.h

See Also Header Files

The date.h header file contains the declarations and prototypes for the date class, their data members, and member functions.

Includes

_DEFS.H

Classes

TDate class

file.h

See Also Header Files

The thread.h header file contains the declarations and prototypes for the file class, their data members, and member functions.

Includes

DATE.H

_DEFS.H

FCNTL.H

STDLIB.H

STDIO.H

SHARE.H

SYS\STAT.H

SYS\TYPES.H

THREAD.H

TIME.H

Classes

TFile class

cstring.h

See Also   Header Files

The cstring.h header file contains the declarations and prototypes for the string and exception classes, their data members, and member functions.

If you are using cstring.h in a Windows program, you must either #define STRICT before you include windows.h or include cstring.h before you include windows.h (STRICT is defined in cstring.h).

Includes

CTYPE.H

EXCEPT.H

REF.H

STDDEF.H

STRING.H

WINDOWS.H

Classes

string

TSubstring

thread.h

See Also Header Files

The thread.h header file contains the declarations and prototypes for the thread classes, their data members, and member functions.

Includes

CSTRING.H

CHECKS.H

_DEFS.H

Classes

TCriticalSection class

TSync class

TThread class

See Also
Precompiled Headers

TDate class

Syntax

```
class TDate
```

Header File

date.h

Description

The *TDate* class represents a date. It has members that read, write, and store dates, and that convert dates to Gregorian calendar dates.

Type Definitions

DayTy

HowToPrint

JulTy

MonthTy

YearTy

Public Constructors

TDate::TDate

Public Member Functions

AsString

Between

CompareTo

Day

DayName

DayOfMonth

DayOfWeek

DaysInYear

DayWithinMonth

FirstDayOfMonth

Hash

IndexOfMonth

IsValid

Jday

Leap

Max

Min

Month

MonthName

NameOfDay

NameOfMonth

Previous

SetPrintOption

WeekDay

Year

Protected Member Functions

AssertIndexOfMonth

AssertWeekDayNumber

Operators

<

<=

>

>=

==

!=

-

+

++

--

+=

-=

<<

>>

Friend Operators

-

TDate::DayTy

TDate class

Syntax

```
typedef unsigned DayTy;
```

Description

Day type.

TDate::HowToPrint

TDate class

Syntax

```
enum HowToPrint{ Normal, Terse, Numbers, EuropeanNumbers, European };
```

Description

Lists different print formats.

TDate::JulTy

TDate class

Syntax

```
typedef unsigned long JulTy;
```

Description

Julian calendar type.

TDate::MonthTy

TDate class

Syntax

```
typedef unsigned MonthTy;
```

Description

Month type.

TDate::YearTy

TDate class

Syntax

```
typedef unsigned YearTy;
```

Description

Year type.

TDate::TDate

TDate class

Form 1
```
TDate();
```

Form 2
```
TDate( DayTy day, YearTy year );
```

Form 3
```
TDate( DayTy day, const char* month, YearTy year);
```

Form 4
```
TDate( DayTy day, MonthTy month, YearTy year);
```

Form 5
```
TDate( istream& is );
```

Form 6
```
TDate( const TTime& time);
```

Description

Form 1: Constructs a TDate object with the current date.

Form 2: Constructs a TDate object with the given day and year. The base date for this computation is December 31 of the previous year. If year == 0, it constructs a TDate with January 1, 1901, as the "day zero." For example, TDate(-1,0) = December 31, 1900, and TDate(1,0) = January 2, 1901.

Form 3: Constructs a TDate object for the given day, month, and year.

Form 4: Constructs a TDate object for the given day, month, and year.

Form 5: Constructs a TDate object, reading the date from input stream is.

Form 6: Constructs a TDate object from TTime object time.

TDate::AsString

TDate class

Syntax

```
string AsString() const;
```

Description

Converts the *TDate* object to a string object.

TDate::Between

TDate class

Syntax

```
int Between( const TDate& d1, const TDate& d2 ) const;
```

Description

Returns 1 if this *TDate* object is between *d1* and *d2*, inclusive.

TDate::CompareTo

TDate class

Syntax

```
int CompareTo( const TDate & ) const;
```

Description

Returns 1 if the target *TDate* is greater than parameter *TDate*, -1 if the target is less than the parameter, and 0 if the dates are equal.

TDate::Day

TDate class

Syntax

```
DayTy Day() const;
```

Description

Returns the day of the year (1-365).

TDate::DayName

TDate class

Syntax

```
const char *DayName( DayTy weekDayNumber );
```

Description

Returns a string name for the day of the week, where Monday is 1 and Sunday is 7.

TDate::DayOfMonth

TDate class

Syntax

```
DayTy DayOfMonth() const;
```

Description

Returns the day of the month (1-31).

TDate::DayOfWeek

TDate class

Syntax

```
DayTy DayOfWeek( const char* dayName );
```

Description

Returns the number associated with a string naming the day of the week, where Monday is 1 and Sunday is 7.

TDate::DaysInYear

TDate class

Syntax

```
DayTy DaysInYear( YearTy );
```

Description

Returns the number of days in the year specified (365 or 366).

TDate::DayWithinMonth

TDate class

Syntax

```
int DayWithinMonth( MonthTy, DayTy, YearTy );
```

Description

Returns 1 if the given day is within the given month for the given year.

TDate::FirstDayOfMonth

TDate class

Form 1

```
DayTy FirstDayOfMonth() const;
```

Form 2

```
DayTy FirstDayOfMonth( MonthTy month) const;
```

Description

Form 1: Returns the number of the first day of the month for this *TDate*.

Form 2: Returns the number of the first day of a given month. Returns 0 if *month* is outside the range 1 through 12.

TDate::Hash

TDate class

Syntax

```
unsigned Hash() const;
```

Description

Returns a hash value for the date.

TDate::IndexOfMonth

TDate class

Syntax

```
MonthTy IndexOfMonth( const char *monthName );
```

Description

Returns the number (1-12) of the month *monthname*.

TDate::IsValid

TDate class

Syntax

```
int IsValid() const;
```

Description

Returns 1 if this *TDate* is valid, 0 otherwise.

TDate::Jday

TDate class

Syntax

```
JulTy Jday( MonthTy, DayTy, YearTy );
```

Description

Converts the given Gregorian calendar date to the corresponding Julian day number. Gregorian calendar started on Sep. 14, 1752. This function is not valid before that date. Returns 0 if the date is invalid.

TDate::Leap

TDate class

Syntax

```
int Leap() const;
```

Description

Returns 1 if this *TDate's* year is a leap year, 0 otherwise.

TDate::Max

TDate class

Syntax

```
TDate Max( const TDate& dt ) const;
```

Description

Compares this *TDate* with *dt* and returns the date with the greater Julian number.

TDate::Min

TDate class

Syntax

```
TDate Min( const TDate& dt ) const;
```

Description

Compares this *TDate* with *dt* and returns the date with the lesser Julian number.

TDate::Month

TDate class

Syntax

```
MonthTy Month() const;
```

Description

Returns the month number for this *TDate*.

TDate::MonthName

TDate class

Syntax

```
const char *MonthName( MonthTy monthNumber );
```

Description

Returns the string name for the given *monthNumber* (1-12). Returns 0 for an invalid *monthNumber*.

TDate::NameOfDay

TDate class

Syntax

```
const char *NameOfDay() const;
```

Description

Returns this *TDate's* day string name.

TDate::NameOfMonth

TDate class

Syntax

```
const char *NameOfMonth() const;
```

Description

Returns this *TDate's* month string name.

TDate::Previous

TDate class

Form 1

```
TDate Previous( const char *dayName ) const;
```

Form 2

```
TDate Previous( DayTy day ) const;
```

Description

Form 1: Returns the *TDate* of the previous *dayName*.

Form 2: Returns the TDate of the previous day.

TDate::SetPrintOption

See Also TDate class

Syntax

```
HowToPrint SetPrintOption( HowToPrint h );
```

Description

Sets the print option for all *TDate* objects and returns the old setting.

See Also
HowToPrint

TDate::WeekDay

TDate class

Syntax

```
DayTy WeekDay() const;
```

Description

Returns 1 (Monday) through 7 (Sunday).

TDate::Year

TDate class

Syntax

```
YearTy Year() const;
```

Description

Returns the year of this *TDate*.

TDate::AssertIndexOfMonth

TDate class

Syntax

```
static int AssertIndexOfMonth( MonthTy m );
```

Description

Returns 1 if *m* is between 1 and 12 inclusive, otherwise returns 0.

TDate::AssertWeekDayNumber

TDate class

Syntax

```
static int AssertWeekDayNumber( DayTy d );
```

Description

Returns 1 if *d* is between 1 and 7 inclusive, otherwise returns 0.

TDate::operator <

TDate class

Syntax

```
int operator < ( const TDate& date ) const;
```

Description

Returns 1 if this *TDate* precedes *date*. otherwise returns 0.

TDate::operator <=

TDate class

Syntax

```
int operator <= ( const TDate& date ) const;
```

Description

Returns 1 if this *TDate* is less than or equal to *date*, otherwise returns 0.

TDate::operator >

TDate class

Syntax

```
int operator >  ( const TDate& date ) const;
```

Description

Returns 1 if this *TDate* is greater than *date*. otherwise returns 0.

TDate::operator >=

TDate class

Syntax

```
int operator >= ( const TDate& date ) const;
```

Description

Returns 1 if this *TDate* is greater than or equal to *date*, otherwise returns 0.

TDate::operator ==

TDate class

Syntax

```
int operator == ( const TDate& date ) const;
```

Description

Returns 1 if this *TDate* is equal to *date*, otherwise returns 0.

TDate::operator !=

TDate class

Syntax

```
int operator != ( const TDate& date ) const;
```

Description

Returns 1 if this *TDate* is not equal to *date*, otherwise returns 0.

TDate::operator -

TDate class

Syntax

```
JulTy operator - ( const TDate& dt ) const;
```

Description

Subtracts *dt* from this *TDate* and returns the difference.

TDate::operator +

TDate class

Form 1

```
friend TDate operator + ( const TDate& dt, int dd );
```

Form 2

```
friend TDate operator + ( int dd, const TDate& dt );
```

Description

Returns a new *TDate* containing the sum of this *TDate* and *dd*.

TDate::operator -

TDate class

Syntax

```
friend TDate operator - ( const TDate& dt, int dd );
```

Description

Subtracts *dd* from this *TDate* and returns the difference.

TDate::operator ++

TDate class

Syntax

```
void operator ++ ();
```

Description

Increments this *TDate* by 1.

TDate::operator --

TDate class

Syntax

```
void operator -- ();
```

Description

Decrements this *TDate* by 1.

TDate::operator +=

TDate class

Syntax

```
void operator += ( int dd );
```

Description

Adds *dd* to this *TDate*.

TDate::operator -=

TDate class

Syntax

```
void operator -= ( int dd );
```

Description

Subtracts *dd* from this *TDate*.

TDate::operator <<

TDate class

Syntax

```
friend ostream& operator << ( ostream& os, const TDate& date );
```

Description

Inserts *date* into output stream *os*.

TDate::operator >>

TDate class

Syntax

```
friend istream& operator >> ( istream& is, TDate& date );
```

Description

Extracts *date* from input stream *is*.

TFile class

Syntax
`class TFile`

Header File
file.h

Description
The *TFile* class encapsulates standard file characteristics and operations.

Constructors
TFile::TFile

Public Data Members
FileNull
File Flags

Member Functions
Close
Flush
GetHandle
GetStatus
IsOpen
Length
LockRange
Open
Position
Read
Remove
Rename
Seek
SeekToBegin
SeekToEnd
SetStatus
UnlockRange
Write

TFile::TFile

TFile class

Form 1
```
TFile();
```

Form 2
```
TFile( int handle );
```

Form 3
```
TFile( const TFile& file );
```

Form 4
```
TFile( const char* name, uint16 access=ReadOnly, uint16 permission=PermRdWr
  );
```

Description

Form 1: Creates a *TFile* object with a file handle of *FileNull*.

Form 2: Creates a *TFile* object with a file handle of *handle*.

Form 3: Creates a *TFile* object with the same file handle file.

Form 4: Creates a *TFile* object and opens file name with the given attributes. The file is created if it does not exist.

TFile::FileNull

TFile class

Syntax

```
enum { FileNull };
```

Description

Represents a null file handle.

File Flags

TFile class

File Translation Modes and Sharing Capabilities

File Read and Write Permissions

File Types

File Pointer Seek Direction

File Translation Modes and Sharing Capabilities

```
enum{
   ReadOnly    = O_RDONLY,
   ReadWrite   = O_RDWR,
   WriteOnly   = O_WRONLY,
   Create      = O_CREAT | O_TRUNC,
   CreateExcl  = O_CREAT | O_EXCL,
   Append      = O_APPEND,
#if defined( __FLAT__ )
   Compat      = SH_COMPAT,
   DenyNone    = SH_DENYNONE,
#else
   DenyRead    = SH_DENYRD,
   DenyWrite   = SH_DENYWR,
#endif
   DenyRdWr    = SH_DENYRW,
   NoInherit   = O_NOINHERIT
   };
```

File Read and Write Permissions

```
enum{
   PermRead    = S_IREAD,
   PermWrite   = S_IWRITE,
   PermRdWr    = S_IREAD | S_IWRITE
   };
```

File Types

```
enum{
   Normal       = 0x00,
   RdOnly       = 0x01,
   Hidden       = 0x02,
   System       = 0x04,
   Volume       = 0x08,
   Directory    = 0x10,
   Archive      = 0x20
  };
```

File Pointer Seek Direction

```
enum seek_dir
   {
   beg = 0,
   cur = 1,
   end = 2
   };
```

TFileStatus structure

```
struct TFileStatus
{
   TTime createTime;
   TTime modifyTime;
   TTime accessTime;
   long size;
   uint8 attribute;
   char fullName[_MAX_PATH];
};
```

TFile::Close

TFile class

Syntax

```
int Close();
```

Description

Closes the file. Returns nonzero if successful, 0 otherwise.

TFile::Flush

TFile class

Syntax

```
void Flush();
```

Description

Performs any pending I/O functions.

TFile::GetHandle

TFile class

Syntax

```
int GetHandle() const;
```

Description

Returns the file handle.

TFile::GetStatus

TFile class

Form 1

```
int GetStatus( TFileStatus& status ) const;
```

Form 2

```
int GetStatus( const char *name, TFileStatus& status );
```

Description

Form 1: Fills *status* with the current file status. Returns nonzero if successful, 0 otherwise.

Form 2: Fills *status* with the status for *name*. Returns nonzero if successful, 0 otherwise.

TFile::IsOpen

TFile class

Syntax

```
int IsOpen() const;
```

Description

Returns 1 if the file is open, 0 otherwise.

TFile::Length

TFile class

Form 1

```
long Length() const;
```

Form 2

```
void Length( long newLen );
```

Description

Form 1: Returns the file length.

Form 2: Resizes file to *newLen*.

TFile::LockRange

See Also TFile class

Syntax

```
void LockRange( long position, uint32 count );
```

Description

Locks *count* bytes, beginning at *position* of the associated file.

See Also
TFile::UnlockRange

TFile::Open

TFile class

Syntax

```
int Open( const char* name, uint16 access, uint16 permission );
```

Description

Opens file *name* with the given attributes. The file will be created if it does not exist. Returns 1 if successful, 0 otherwise.

TFile::Position

TFile class

Syntax

```
long Position() const;
```

Description

Returns the current position of the file pointer. Returns -1 to indicate an error.

TFile::Read

TFile class

Form 1

```
int Read( void *buffer, int numBytes );
```

Form 2

```
long Read( void huge *buffer, long numBytes );
```

Description

Form 1: Reads *numBytes* from the file into *buffer*.

Form 2: Reads *numBytes* from the file into *buffer*. (32-bit Windows version)

TFile::Remove

TFile class

Syntax

```
static void Remove( const char *name );
```

Description

Removes file *name*. Returns 0 if successful, -1 if unsuccessful.

TFile::Rename

TFile class

Syntax

```
static void Rename( const char *oldName, const char *newName );
```

Description

Renames file *oldName* to *newName*.

TFile::Seek

TFile class

Syntax

```
long Seek( long offset, int origin = beg );
```

Description

Repositions the file pointer to *offset* bytes from the specified *origin*.

TFile::SeekToBegin

TFile class

Syntax

```
long SeekToBegin();
```

Description

Repositions the file pointer to the beginning of the file.

TFile::SeekToEnd

TFile class

Syntax

```
long SeekToEnd();
```

Description

Repositions the file pointer to the end of the file.

TFile::SetStatus

TFile class

Syntax

```
static int SetStatus( const char *name, const TFileStatus& status );
```

Description

Sets file name's status to *status*.

TFile::UnlockRange

See Also TFile class

Syntax

```
void UnlockRange(long Position, uint32 count );
```

Description

Unlocks the range at the given *Position*.

See Also
TFile::LockRange

TFile::Write

TFile class

Form 1

```
int Write( const void *buffer, int numBytes );
```

Form 2

```
long Write( const void huge *buffer, long numBytes );
```

Description

Form 1: Writes *numbytes* of *buffer* to the file.

Form 2: Writes *numbytes* of *buffer* to the file. (32-bit Windows version)

TCriticalSection class

Header File
thread.h

Syntax
```
class TCriticalSection
```

Description
TCriticalSection provides a system-independent interface to critical sections in threads. TCriticalSection objects can be used in conjunction with TCriticalSection::Lock objects to guarantee that only one thread can be executing any of the code sections protected by the lock at any given time.

Constructor
TCriticalSection

Destructor
~TCriticalSection

TCriticalSection::TCriticalSection

TCriticalSection class

Syntax

```
TCriticalSection();
```

Description

Constructs a TCriticalSection object.

TCriticalSection::~TCriticalSection

TCriticalSection class

Syntax

```
~TCriticalSection();
```

Description

Destroys a *TCriticalSection* object.

TCriticalSection::Lock class

Example TCriticalSection class

Header File
thread.h

Syntax
```
class Lock
```

Description
This nested class handles locking and unlocking critical sections.

Constructor
Lock

Destructors
~Lock

Example

Only one thread of execution will be allowed to execute the critical code inside function *f* at any one time.

```
TCriticalSection LockF;
void f()
{
   TCriticalSection::Lock(LockF);

   // critical processing here
}
```

TCriticalSection::Lock

TCriticalSection::Lock class

Syntax

```
Lock( const TCriticalSection& );
```

Description

Requests a lock on the *TCriticalSection* object. If no other *Lock* object holds a lock on that *TCriticalSection* object, the lock is allowed and execution continues. If another *Lock* object holds a lock on that object, the requesting thread is blocked until the lock is released.

TCriticalSection::~Lock

TCriticalSection::Lock class

Syntax

```
~Lock();
```

Description

Releases the lock.

TMutex class

Header File
thread.h

Description
TMutex provides a system-independent interface to critical sections in threads. *TMutex* objects can be used in conjunction with TMutex::Lock class objects to guarantee that only one thread can be executing any of the code sections protected by the lock at any given time.

The differences between the classes TCriticalSection and *TMutex* are that a timeout can be specified when creating a *Lock* on a *TMutex* object, and that a *TMutex* object has a HMTX that can be used outside the class. This mirrors the distinction made in Windows NT between a CRITICALSECTION and a *Mutex*. Under NT a *TCriticalSection* object is much faster than a *TMutex* object. Under operating systems that do not make this distinction a *TCriticalSection* object can use the same underlying implementation as a *TMutex*, losing the speed advantage that it has under NT.

Public Constructor
TMutex

Public Destructor
~TMutex

Operator
HMTX

TMutex::TMutex

TMutex class

Syntax

```
TMutex();
```

Description

Constructs a *TMutex* object.

TMutex::~TMutex

TMutex class

Syntax

```
~TMutex();
```

Description

Destroys a *TMutex* object.

TMutex::HMTX

TMutex class

Syntax

```
operator HMTX() const;
```

Description

Returns a handle to the underlying *TMutex* object, for use in operating system calls that require it.

TMutex::Lock class

TMutex class

Header File
thread.h

Description
This nested class handles locking and unlocking *TMutex* objects.

Public Constructor
Lock

Public Member Function
Release

TMutex::Lock

TMutex::Lock class

Syntax

```
Lock( const TMutex&, unsigned long timeOut = NoLimit );
```

Description

Requests a lock on the *TMutex* object. If no *Lock* object in another thread holds a lock on that *TMutex* object, the lock is allowed and execution continues. If a *Lock* object in another thread holds a lock on that object, the requesting thread is blocked until the lock is released.

TMutex::Release

TMutex::Lock class

Syntax

```
void Release();
```

Description

Releases the lock on the *TMutex* object.

TSync class

See Also Example

Header File
thread.h

Description
TSync provides a system-independent interface for building classes that act like monitors--classes in which only one member can execute on a particular instance at any one time. *TSync* uses *TCriticalSection*, has no public members, and can only be used as a base class.

Protected Constructors
TSync

Protected Operator
=

See Also
class TSync::Lock

// TSync Example

```
class ThreadSafe : private TSync
{
public:
   void f();
   void g();
private:
   int i;
};

void ThreadSafe::f()
{
   Lock(this);
   if( i == 2 )
     i = 3;
}

void ThreadSafe::g()
{
   Lock(this);
   if( i == 3 )
     i = 2;
}
```

TSync::TSync

TSync class

Form 1

```
TSync();
```

Form 2

```
TSync( const TSync& );
```

Description

Form 1: Default constructor.

Form 2: Copy constructor. Does not copy the TCriticalSection object.

TSync::Operator =

TSync class

Syntax

```
const TSync& operator = ( const TSync& s )
```

Description

Assigns *s* to the target, and does not copy the *TCriticalSection* object.

TSync::Lock class

TSync class

Header File
thread.h

Syntax
```
class Lock : private TCriticalSection::Lock
```

Description
This nested class handles locking and unlocking critical sections.

Constructor
Lock

Destructor
~Lock

TSync::Lock

TSync::Lock class

Syntax

```
Lock( const TSync *s );
```

Description

Requests a lock on the critical section of the *TSync* object pointed to by *s*. If no other *Lock* object holds a lock on that *TCriticalSection* object, the lock is allowed and execution continues. If another *Lock* object holds a lock on that object, the requesting thread is blocked until the lock is released.

TSync::~Lock

TSync::Lock class

Syntax

```
~Lock();
```

Description

Releases the lock.

TThread class

See Also Example

Syntax

```
class TThread
```

Header File

thread.h

Description

The *TThread* class provides a system-independent interface to threads.

Protected Constructors

TThread

Protected Destructor

~TThread

Type Definition

Status

Public Member Functions

GetPriority

GetStatus

Resume

SetPriority

Start

Suspend

Terminate

TerminateAndWait

WaitForExit

Protected Member Function

ShouldTerminate

Protected Operator

=

See Also
TThread::TThreadError class

// TThread class Example

```
class TimerThread : private TThread
{
public:
   TimerThread() : Count(0) {}
private:
   unsigned long Run();
   int Count;
};

unsigned long TimerThread::Run()
{
   // loop 10 times
   while( Count++ < 10 )
   {
      sleep(1000);    // delay 1 second
      cout << "Iteration " << Count << endl;
   }
   return 0L;
}

int main()
{
   TimerThread timer;
   timer.Start();
   Sleep( 20000 );     // delay 20 seconds
   return 0;
}
```

TThread::Status

TThread class

Syntax

```
enum Status { Created, Running, Suspended, Finished, Invalid };
```

Description

Describes the state of the thread, as follows:

Thread State	Description
Created	The object has been created but its thread has not been started. The only valid transition from this state is to Running, which happens on a call to Start. In particular, a call to Suspend or Resume when the object is in this state is an error and will throw an exception.
Running	The thread has been started successfully. There are two transitions from this state: • When the user calls Suspend, the object moves into the Suspended state. • When the thread exits the object moves into the Finished state. Calling Resume on an object that is in the Running state is an error and will throw an exception.
Suspended	The thread has been suspended by the user. Subsequent calls to Suspend nest, so there must be as many calls to Resume as there were to Suspend before the thread resumes execution.
Finished	The thread has finished executing. There are no valid transitions out of this state. This is the only state from which it is legal to invoke the destructor for the object. Invoking the destructor when the object is in any other state is an error and will throw an exception.

TThread::TThread

TThread class

Form 1

```
TThread();
```

Form 2

```
TThread( const TThread& );
```

Description

Form 1: Constructs an object of type *TThread*.

Form 2: Copy constructor. Puts the target object into the *Created* state.

TThread::~TThread

TThread class

Syntax

```
virtual ~TThread();
```

Description

Destroys the *TThread* object.

TThread::GetPriority

See Also TThread class

Syntax

```
int GetPriority() const;
```

Description

Gets the thread priority.

See Also
TThread::SetPriority

TThread::GetStatus

TThread class

Syntax

```
Status GetStatus() const;
```

Description

Returns the current status of the thread:

Thread State	Description
Created	The object has been created but its thread has not been started. The only valid transition from this state is to Running, which happens on a call to Start. In particular, a call to Suspend or Resume when the object is in this state is an error and will throw an exception.
Running	The thread has been started successfully. There are two transitions from this state: • When the user calls Suspend, the object moves into the Suspended state. • When the thread exits the object moves into the Finished state. Calling Resume on an object that is in the Running state is an error and will throw an exception.
Suspended	The thread has been suspended by the user. Subsequent calls to Suspend nest, so there must be as many calls to Resume as there were to Suspend before the thread resumes execution.
Finished	The thread has finished executing. There are no valid transitions out of this state. This is the only state from which it is legal to invoke the destructor for the object. Invoking the destructor when the object is in any other state is an error and will throw an exception.

TThread::Resume

TThread class

Syntax

```
unsigned long Resume();
```

Description

Resumes execution of a suspended thread.

TThread::SetPriority

See Also	TThread class

Syntax

```
int SetPriority(int);
```

Description

Sets the thread priority.

See Also
TThread::GetPriority

TThread::Start

TThread class

Syntax

```
HANDLE Start();
```

Description

Begins execution of the thread, and returns the thread handle.

TThread::Suspend

TThread class

Syntax

```
unsigned long Suspend();
```

Description

Suspends execution of the thread.

TThread::Terminate

TThread class

Syntax

```
void Terminate();
```

Description

Sets an internal flag that indicates that the thread should exit. The derived class can check the state of this flag by calling ShouldTerminate.

TThread::TerminateAndWait

TThread class

Syntax

```
void TerminateAndWait( unsigned long timeout = NoLimit );
```

Description

Combines the behavior of Terminate and WaitForExit. Sets an internal flag that indicates that the thread should exit and blocks the calling thread until the internal thread exits or until the time specified by timeout, in milliseconds, expires. A timeout of -1 says to wait indefinitely.

TThread::WaitForExit

TThread class

Syntax

```
void WaitForExit( unsigned long timeout = NoLimit );
```

Description

Blocks the calling thread until the internal thread exits or until the time specified by timeout, in milliseconds, expires. A timeout of -1 says wait indefinitely.

TThread::ShouldTerminate

TThread class

Syntax

```
int ShouldTerminate() const;
```

Description

Returns a nonzero value to indicate that Terminate or TerminateAndWait has been called and that the thread will finish its processing and exit.

TThread::Operator =

TThread class

Syntax

```
const TThread& operator = ( const TThread& );
```

Description

The *TThread* assignment operator. The target object must be in either the *Created* or *Finished* state. If so, assignment puts the target object into the *Created* state. If the object is not in either state, an exception will be thrown.

TThread::TThreadError class

TThread class

Syntax

```
class TThreadError
```

Header File

thread.h

Description

TThreadError defines the exceptions that are thrown when a threading error occurs.

Type Definition

ErrorType

Public Member Function

GetErrorType

TThread::ErrorType

TThread::TThreadError class

Syntax

```
enum ErrorType
   {
   SuspendBeforeRun,
   ResumeBeforeRun,
   ResumeDuringRun,
   SuspendAfterExit,
   ResumeAfterExit,
   CreationFailure,
   DestroyBeforeExit,
   AssignError
   };
```

Description

Identifies the type of error that occurred. The following list explains each error type:

Error Type	Description
SuspendBeforeRun	The user called Suspend on an object before calling Start.
ResumeBeforeRun	The user called Resume on an object before calling *Start*.
ResumeDuringRun	The user called *Resume* on a thread that was not suspended.
SuspendAfterExit	The user called *Suspend* on an object whose thread had already exited.
ResumeAfterExit	The user called *Resume* on an object whose thread had already exited.
CreationFailure	The operating system was unable to create the thread.
DestroyBeforeExit	The destructor of the object was invoked before its thread had exited.
AssignError	An attempt was made to assign to an object that was not in either the Created or Finished state.

TThread::GetErrorType

TThread::TThreadError class

Syntax

```
ErrorType GetErrorType() const;
```

Description

Returns the ErrorType for the error that occurred.

TTime class

Header File
time.h

Syntax
```
class TTime
```

Description
The *TTime* class encapsulates time functions and characteristics.

Type Definitions
HourTy
MinuteTy
SecondTy
ClockTy

Constructors
TTime::TTime

Protected Data Members
MaxDate
RefDate

Public Member Functions
AsString
BeginDST
Between
CompareTo
EndDST
Hash
Hour
HourGMT
IsDST
IsValid
Max
Min
Minute
MinuteGMT
PrintDate
Second
Seconds

Protected Member Functions
AssertDate

Operators
<
<=
>

>=

==

!=

++

--

+=

-=

+

+

-

-

<<

<<

>>

TTime::MaxDate

TTime class

Syntax

```
static const TDate MaxDate;
```

Description

The maximum valid date for *TTime* objects.

TTime::RefDate

TTime class

Syntax

```
static const TDate RefDate;
```

Description

The minimum valid date for *TTime* objects: January 1, 1901.

TTime::TTime

TTime class

Form 1
```
TTime();
```

Form 2
```
TTime( ClockTy s );
```

Form 3
```
TTime( HourTy h, MinuteTy m, SecondTy s = 0 );
```

Form 4
```
TTime( const TDate&, HourTy h=0, MinuteTy m=0, SecondTy s=0 );
```

Description

Form 1: Constructs a TTime object with the current time.

Form 2: Constructs a *TTime* object with the given *s* (seconds since January 1, 1901).

Form 3: Constructs a *TTime* object with the given time and today's date.

Form 4: Constructs a *TTime* object with the given time and date.

TTime::AsString

TTime class

Syntax

```
string AsString() const;
```

Description

Returns a string object containing the time.

TTime::BeginDST

TTime class

Syntax

```
static TTime BeginDST( unsigned year );
```

Description

Returns the start of daylight saving time for the given *year*.

TTime::Between

TTime class

Syntax

```
int Between( const TTime& a, const TTime& b ) const;
```

Description

Returns 1 if the target date is between *TTime a* and *TTime b*, 0 otherwise.

TTime::CompareTo

TTime class

Syntax

```
int CompareTo( const TTime & ) const;
```

Description

Compares *t* to this *TTime* object and returns 0 if the times are equal, 1 if *t* is earlier, and -1 if *t* is later.

TTime::EndDST

TTime class

Syntax

```
static TTime EndDST( unsigned year );
```

Description

Returns the time when daylight saving time ends for the given *year*.

TTime::Hash

TTime class

Syntax

```
unsigned Hash() const;
```

Description

Returns seconds since January 1, 1901.

TTime::Hour

TTime class

Syntax

```
HourTy Hour() const;
```

Description

Returns the hour in local time.

TTime::HourGMT

TTime class

Syntax

```
HourTy HourGMT() const;
```

Description

Returns the hour in Greenwich mean time.

TTime::IsDST

TTime class

Syntax

```
int IsDST() const;
```

Description

Returns 1 if the time is in daylight saving time, 0 otherwise.

TTime::IsValid

TTime class

Syntax

```
int IsValid() const;
```

Description

Returns 1 if this *TTime* object contains a valid time, 0 otherwise.

TTime::Max

TTime class

Syntax

```
TTime Max( const TTime& t ) const;
```

Description

Returns either this *TTime* object or *t*, whichever is greater.

TTime::Min

TTime class

Syntax

```
TTime Min( const TTime& t ) const;
```

Description

Returns either this *TTime* object or *t*, whichever is less.

TTime::Minute

TTime class

Syntax

```
MinuteTy Minute() const;
```

Description

Returns the minute in local time.

TTime::MinuteGMT

TTime class

Syntax

```
MinuteTy MinuteGMT() const;
```

Description

Returns the minute in Greenwich Mean Time.

TTime::PrintDate

TTime class

Syntax

```
static int PrintDate( int flag);
```

Description

Set *flag* to 1 to print the date along with the time; set to 0 to not print the date. Returns the old setting.

TTime::Second

TTime class

Syntax

```
SecondTy Second() const;
```

Description

Returns seconds.

TTime::Seconds

TTime class

Syntax

```
ClockTy Seconds() const;
```

Description

Returns seconds since January 1, 1901.

TTime::AssertDate

TTime class

Syntax

```
static int AssertDate( const TDate& d );
```

Description

Returns 1 if *d* is between the earliest valid date (RefDate) and the latest valid date (MaxDate).

TTime::Operator <

TTime class

Syntax

```
int operator <  ( const TTime& t ) const;
```

Description

Returns 1 if the target time is less than time *t*, 0 otherwise.

TTime::Operator <=

TTime class

Syntax

```
int operator <= ( const TTime& t ) const;
```

Description

Returns 1 if the target time is less than or equal to time *t*, 0 otherwise.

TTime::Operator >

TTime class

Syntax

```
int operator > ( const TTime& t ) const;
```

Description

Returns 1 if the target time is greater than time *t*, 0 otherwise.

TTime::Operator >=

TTime class

Syntax

```
int operator >= ( const TTime& t ) const;
```

Description

Returns 1 if the target time is greater than or equal to time *t*, 0 otherwise.

TTime::Operator ==

TTime class

Syntax

```
int operator == ( const TTime& t ) const;
```

Description

Returns 1 if the target time is equal to time *t*, 0 otherwise.

TTime::Operator !=

TTime class

Syntax

```
int operator != ( const TTime& t ) const;
```

Description

Returns 1 if the target time is not equal to time *t*, 0 otherwise.

TTime::Operator ++

TTime class

Syntax

```
void operator++();
```

Description

Increments the time by 1 second.

TTime::Operator --

TTime class

Syntax

```
void operator--();
```

Description

Decrements the time by 1 second.

TTime::Operator +=

TTime class

Syntax

```
void operator+=(long s);
```

Description

Adds *s* seconds to the time.

TTime::Operator -=

TTime class

Syntax

```
void operator-=(long s);
```

Description

Subtracts *s* seconds from the time.

TTime::Operator +

TTime class

Syntax

```
friend TTime operator + ( const TTime& t, long s );
friend TTime operator + ( long s, const TTime& t );
```

Description

Adds *s* seconds to time *t*.

TTime::Operator -

TTime class

Syntax

```
friend TTime operator - ( const TTime& t, long s );
friend TTime operator - ( long s, const TTime& t );
```

Description

Performs subtraction, in seconds, between *s* and *t*.

TTime::Operator <<

TTime class

Form 1

```
friend ostream& operator << ( ostream& os, const TTime& t);
```

Form 2

```
friend opstream& operator << ( opstream& s, const TTime& d );
```

Description

Form 1: Inserts time *t* into output stream *os*.

Form 2: Inserts time *t* into persistent stream *s*.

TTime::Operator >>

TTime class

Syntax

```
friend ipstream& operator >> ( ipstream& s, TTime& d );
```

Description

Extracts time *t* from persistent stream *s*.

TTime Type Definitions

TTime class

Syntax

```
typedef unsigned HourTy;
typedef unsigned MinuteTy;
typedef unsigned SecondTy;
typedef unsigned long ClockTy;
```

Header File

time.h

Description

Type definitions for hours, minutes, seconds, and seconds since January 1, 1901.

string class

See Also

Header File
cstring.h

Syntax
```
class string;
```

Description
This class uses a technique called "copy-on-write". Multiple instances of a string can refer to the same piece of data so long as it is in a "read only" situation. If a string writes to the data, then a copy is automatically made if more than one string is referring to it.

Constructors
string::string

Data Members
StripType

Public Member Functions
ansi_to_oem

append

assign

compare

contains

copy

c_str

find

find_first_of

find_first_not_of

find_last_of

find_last_not_of

get_at

get_case_sensitiveFlag

get_initial_capacity

get_max_waste

get_paranoid_check

get_resize_increment

get_skipwhitespace_flag

hash

initial_capacity

insert

is_null

length

MaxWaste

oem_to_ansi

prepend

put_at

read_file
read_line
read_string
read_to_delim
read_token
rfind
remove
replace
reserve
resize
resize_increment
set_case_sensitive
set_paranoid_check
skip_whitespace
strip
substr
substring
to_lower
to_upper

Protected Member Functions
assert_element
assert_index
cow
valid_element
valid_index

Operators
=
+=
+=
+
[]
()
==
!=
<
<=
>
>=

Related Global Operators and Functions
>>
<<
+
getline
to_lower

to_upper

See Also

TRegexp class

string::StripType

See Also string class

Syntax

```
enum StripType { Leading, Trailing, Both };
```

Description

Enumerates type of stripping.

See Also

string::strip

string::string

string class

Form 1
```
string();
```

Form 2
```
string(const string _FAR &s);
```

Form 3
```
string( const string _FAR &s, size_t start, size_t n = NPOS );
```

Form 4
```
string(const char _FAR *cp);
```

Form 5
```
string(const char _FAR *cp, size_t start, size_t n = NPOS);
```

Form 6
```
string( char c );
```

Form 7
```
string( char c, size_t n = NPOS );
```

Form 8
```
string( signed char c );
```

Form 9
```
string( signed char c, size_t n = NPOS );
```

Form 10
```
string( unsigned char c );
```

Form 11
```
string( unsigned char c, size_t n = NPOS );
```

Form 12
```
string(const TSubString _FAR &ss);
```

Form 13
```
string( const char __far *cp );
```

Form 14
```
string( const char __far *cp, size_t start, size_t n = NPOS );
```

Form 15
```
string( HINSTANCE instance, UINT id, int len = 255 );
```

Description

Form 1: The default constructor. Creates a string of length zero.

Form 2: Copy constructor. Creates a string that contains a copy of the contents of string *s*.

Form 3: A string containing a copy of the *n* bytes beginning at position start of string *s* is created.

Form 4: A string containing a copy of the bytes from the location pointed to by *cp* through the first 0 byte is created (conversion from *char**).

Form 5: A string containing a copy of the *n* bytes beginning at the location pointed to by *cp* is created.

Form 6: Constructs a string containing the character c.

Form 7: Constructs a string containing the character c repeated *n* times.

Form 8: Constructs a string containing the character c.

Form 9: Constructs a string containing the character c repeated *n* times.

Form 10: Constructs a string containing the character c.

Form 11: Constructs a string containing the character c repeated *n* times.

Form 12: Constructs a string from the substring *ss*.

Form 13: Constructs strings for Windows small and medium memory model.

Form 14: Constructs strings for Windows small and medium memory model.

Form 15: Windows version for constructing a string from a resource.

string::ansi_to_oem

string class

Syntax

```
void ansi_to_oem();
```

Description

Converts the target string from the ANSI character set into the OEM-defined character set.

string::append

string class

Form 1

```
string _FAR & append( const string _FAR &s );
```

Form 2

```
string _FAR & append( const string _FAR &s, size_t start, size_t n =
  NPOS );
```

Form 3

```
string _FAR & append( const char _FAR *cp, size_t start, size_t n = NPOS );
```

Description

Form 1: Appends string *s* to the target string.

Form 2: Beginning from the start position in *s*, appends the next *n* characters of string *s* to the target string.

Form 3: Beginning from the start position of the character array *cp*, appends the next *n* characters to the target string.

string::assign

See Also   string class

Form 1

```
string _FAR & assign( const string _FAR &s );
```

Form 2

```
string _FAR & assign( const string _FAR &s, size_t start, size_t n =
  NPOS );
```

Description

Form 1: Assigns string *s* to target string.

Form 2: Beginning from the start position in *s*, copies *n* characters to target string.

See Also
operator =

string::compare

string class

Form 1

```
int compare(const string _FAR &s) const throw();
```

Form 2

```
int compare(const string _FAR &s, size_t orig, size_t n = NPOS ) const
  throw();
```

Description

Form 1: Compares the target string to the string *s*. *compare* returns an integer less than, equal to, or greater than 0, depending on whether the target string is less than, equal to, or greater than *s*.

Form 2: Compares not more than *n* characters of string *s*, beginning at character position *orig*, with this string.

string::contains

string class

Form 1

```
int contains(const char _FAR * pat) const;
```

Form 2

```
int contains(const string _FAR & s) const;
```

Description

Form 1: Returns 1 if *pat* is found in the target string, 0 otherwise.

Form 2: Returns 1 if string *s* is found in the target string, 0 otherwise.

string::copy

string class

Form 1

```
size_t copy( char _FAR *cb, size_t n = NPOS );
```

Form 2

```
size_t copy( char _FAR *cb, size_t n, size_t pos );
```

Form 3

```
string copy() const throw( xalloc ).;
```

Description

Form 1: Copies at most *n* characters from the target string into the *char* array pointed to by *cb*. *copy* returns the number of characters copied.

Form 2: Copies at most *n* characters beginning at position *pos* from the target string into the *char* array pointed to by *cb*. *copy* returns the number of characters copied.

Form 3: Returns a distinct copy of the string.

string::c_str

string class

Syntax

```
const char _FAR *c_str() const;
```

Description

Returns a pointer to a zero-terminated character array, that holds the same characters contained in the string. The returned pointer may point to the actual contents of the string, or it may point to an array that the string allocates for this function call. The effects of any direct modification to the contents of this array are undefined, and the results of accessing this array after the execution of any non-**const** member function on the target string are undefined.

Conversions from a string object to a *char** are inherently dangerous, because they violate the class boundary and can lead to dangling pointers. For this reason class string does not have an implicit conversion to *char**, but provides *c_str* for use when this conversion is needed.

string::find

See Also		string class

Form 1

```
size_t find( const string _FAR &s );
```

Form 2

```
size_t find( const string _FAR &s, size_t pos );
```

Form 3

```
size_t find( const TRegexp _FAR &pat, size_t i = 0 );
```

Form 4

```
size_t find( const TRegexp _FAR &pat, size_t _FAR *ext, size_t i = 0 )
  const;
```

Description

Form 1: Locates the first occurrence of the string *s* in the target string. If the string is found, it returns the position of the beginning of *s* within the target string. If the string *s* is not found, it returns *NPOS*.

Form 2: Locates the first occurrence of the string *s* in the target string, beginning at the position *pos*. If the string is found, it returns the position of the beginning of *s* within the target string. If *s* is not found, it returns *NPOS*, and does not change *pos*.

Form 3: Searches the string for patterns matching regular expression *pat* beginning at location *i*. It returns the position of the beginning of *pat* within the target string. If the *pat* is not found, it returns *NPOS*, and does not change *pos*.

Form 4: Searches the string for patterns matching regular expression *pat* beginning at location *i*. Parameter *ext* returns the length of the matching string if found. It returns the position of the beginning of *pat* within the target string. If the *pat* is not found, it returns *NPOS*, and does not change *pos*.

See Also
string::rfind

string::find_first_of

string class

Form 1

```
size_t find_first_of( const string _FAR &s ) const;
```

Form 2

```
size_t find_first_of( const string _FAR &s, size_t pos ) const;
```

Description

Form 1: Locates the first occurrence in the target string of any character contained in string *s*. If the search is successful *find_first_of* returns the character location. If the search fails, *find_first_of* returns *NPOS*.

Form 2: Locates the first occurrence in the target string of any character contained in string *s*. If the search is successful, the function returns the character position within the target string, and *find_first_of* returns 1. If the search fails or if `pos > length()`, *find_first_of* returns *NPOS*.

string::find_first_not_of

string class

Form 1

```
size_t find_first_not_of( const string _FAR &s ) const;
```

Form 2

```
size_t find_first_not_of( const string _FAR &s, size_t pos ) const;
```

Description

Form 1: Locates the first occurrence in the target string of any character not contained in string *s*. If the search is successful, *find_first_not_of* returns the character position within the target string. If the search fails it returns *NPOS*.

Form 2: Locates the first occurrence in the target string of any character not contained in string *s*. If the search is successful, *find_first_not_of* returns the character position within the target string. If the search fails or if `pos > length()`, *find_first_not_of* returns *NPOS*.

string::find_last_of

string class

Form 1

```
size_t find_last_of( const string _FAR &s ) const;
```

Form 2

```
size_t find_last_of( const string _FAR &s, size_t pos ) const;
```

Description

Form 1: Locates the last occurrence in the target string of any character contained in string *s*. If the search is successful *find_last_of* returns the character position within the target string. If the search fails it returns *NPOS*.

Form 2: Locates the last occurrence in the target string of any character contained in string *s* after position *pos*. If the search is successful, *find_last_of* returns the character position within the target string. If the search fails or if `pos > length()`, *find_last_of* returns *NPOS*.

string::find_last_not_of

string class

Form 1

```
size_t find_last_not_of( const string _FAR &s ) const;
```

Form 2

```
size_t find_last_not_of( const string _FAR &s, size_t pos ) const;
```

Description

Form 1: Locates the last occurrence in the target string of any character not contained in string *s*. If the search is successful *find_last_not_of* returns the character position within the target string. If the search fails it returns *NPOS*.

Form 2: Locates the last occurrence in the target string of any character not contained in string *s* after position *pos*. If the search is successful, *find_last_not_of* returns the character position within the target string. If the search fails or if `pos > length()`, *find_last_not_of* returns *NPOS*.

string::get_at

See Also   string class

Syntax

```
char get_at( size_t pos ) const throw( outofrange );
```

Description

Returns the character at the specified position. If `pos > length()-1` an *outofrange* exception is thrown.

See Also
string:put_at

string::get_case_sensitive_flag

string class

Syntax

```
static int get_case_sensitive_flag();
```

Description

Returns 0 if the string comparisons are case sensitive, 1 if not.

string::get_initial_capacity

string class

Syntax

```
static unsigned get_initial_capacity();
```

Description

Returns the number of characters that will fit in the string without resizing.

string::get_max_waste

string class

Syntax

```
static unsigned get_max_waste();
```

Description

After a string is resized, returns the amount of free space available.

string::get_paranoid_check

string class

Syntax

```
static int get_paranoid_check();
```

Description

Returns 1 if paranoid checking is enabled, 0 if not.

string::get_resize_increment

string class

Syntax

```
static unsigned get_resize_increment();
```

Description

Returns the string resizing increment.

string::get_skipwhitespace_flag

string class

Syntax

```
static int get_skipwhitespace_flag();
```

Description

Returns 1 if whitespace is skipped, 0 if not.

string::hash

string class

Syntax

```
unsigned hash() const;
```

Description

Returns hash value.

string::initial_capacity

string class

Syntax

```
static size_t initial_capacity(size_t ic = 63);
```

Description

Sets initial string allocation capacity.

string::insert

string class

Form 1

```
string _FAR &insert( size_t pos, const string _FAR &s );
```

Form 2

```
string _FAR &insert( size_t pos, const string _FAR &s, size_t start, size_t
  n = NPOS );
```

Description

Form 1: Inserts string *s* at position *pos* in the target string. *insert* returns a reference to the resulting string.

Form 2: Beginning at position *start* in *s*, the insert function inserts not more than *n* characters from the target string to the string *s* at position *pos* in the target string. *insert* returns a reference to the resulting string. If *pos* is invalid, insert throws the *outofrange* exception.

string::is_null

string class

Syntax

```
int is_null() const;
```

Description

Returns 1 if the string is empty, 0 otherwise.

string::length

string class

Syntax

```
unsigned length() const;
```

Description

Returns the number of characters in the target string. Since null characters can be stored in a string, `length()` might be greater than `strlen(c_str())`.

string::MaxWaste

string class

Syntax

```
static size_t MaxWaste(size_t mw = 63);
```

Description

Sets the maximum empty space size and resizes the string.

string::oem_to_ansi

string class

Syntax

```
void oem_to_ansi();
```

Description

Windows function for converting the target string from the ANSI character set to the OEM-defined character set.

string::prepend

string class

Form 1

```
string _FAR &prepend( const string _FAR &s );
```

Form 2

```
string _FAR &prepend( const string _FAR &s, size_t start, size_t n =
  NPOS );
```

Form 3

```
string _FAR &prepend( const char _FAR *cp );
```

Form 4

```
string _FAR &prepend( const char _FAR *cp, size_t start, size_t n = NPOS );
```

Description

Form 1: Prepends string *s* to the target string.

Form 2: Beginning from start position in *s*, the prepend function prefixes the target string with *n* characters taken from string *s*.

Form 3: Prepends the character array *cp* to the target string.

Form 4: Beginning from start position in *cp*, the prepend function prefixes the target string with *n* characters taken from character array *cp*.

string::put_at

string class

Syntax

```
void put_at( size_t pos, char c ) throw( outofrange );
```

Description

Replaces the character at *pos* with *c*. If *pos* is greater than or equal to *length()* an *outofrange* exception is thrown.

string::read_file

string class

Syntax

```
istream _FAR &read_file(istream _FAR &is);
```

Description

Reads from input stream *is* until an EOF or a null terminator is reached.

string::read_line

string class

Syntax

```
istream _FAR &read_line(istream _FAR &is);
```

Description

Reads from input stream *is* until an EOF or a newline is reached.

string::read_string

string class

Syntax

```
istream _FAR &read_string(istream _FAR &is);
```

Description

Reads from input stream *is* until an EOF or a null terminator is reached.

string::read_to_delim

string class

Syntax

```
istream _FAR &read_to_delim(istream _FAR &is, char delim = '\n');
```

Description

Reads from input stream *is* until an EOF or a *delim* is reached.

string::read_token

string class

Syntax

```
istream _FAR &read_token(istream _FAR &is);
```

Description

Reads from input stream *is* until whitespace is reached. Note that this function skips any initial whitespace.

string::rfind

See Also		string class

Form 1

```
size_t rfind( const string _FAR &s );
```

Form 2

```
size_t rfind( const string _FAR &s, size_t pos );
```

Description

Form 1: Locates the last occurrence of the string *s* in the target string. If the string is found, it returns the position of the beginning of the string *s* within the target string. If *s* is not found, it returns *NPOS*.

Form 2: Locates the last occurrence of the string *s*, that is not beyond the position *pos* in the target string. If the string is found, it returns the position of the beginning of *s* within the target string. If *s* is not found it returns *NPOS*, and does not change *pos*.

See Also
string::find

string::remove

string class

Form 1

```
string _FAR &remove( size_t pos );
```

Form 2

```
string _FAR &remove( size_t pos, size_t n = NPOS );
```

Description

Form 1: Removes the characters from *pos* to the end of the target string and returns a reference to the resulting string.

Form 2: Removes at most *n* characters from the target string beginning at *pos* and returns a reference to the resulting string.

string::replace

string class

Form 1

```
string _FAR &replace( size_t pos, size_t n = NPOS, const string _FAR &s );
```

Form 2

```
string _FAR &replace( size_t pos, size_t n1, const string _FAR &s, size_t
  start, size_t n2 );
```

Description

Form 1: Removes at most *n* characters from the target string beginning at *pos*, and replaces them with a copy of the string *s*. *replace* returns a reference to the resulting string.

Form 2: Removes at most *n1* characters from the target string beginning at *pos*, and replaces them with *n2* characters of string *s* beginning at *start*. *replace* returns a reference to the resulting string.

string::reserve

string class

Form 1

```
size_t reserve() const;
```

Form 2

```
void reserve( size_t ic );
```

Description

Form 1: Returns an implementation-dependent value that indicates the current internal storage size. The returned value is always greater than or equal to `length()`.

Form 2: Suggests to the implementation that the target string may eventually require *ic* bytes of storage.

string::resize

string class

Syntax

```
void resize(size_t m);
```

Description

Resizes the string to *m* characters, truncating or adding blanks as necessary.

string::resize_increment

string class

Syntax

```
static size_t resize_increment(size_t ri = 64);
```

Description

Sets the resize increment for automatic resizing.

string::set_case_sensitive

string class

Syntax

```
static int set_case_sensitive(int tf = 1);
```

Description

Sets case sensitivity. 1 is case sensitive; 0 is not case sensitive.

string::set_paranoid_check

string class

Syntax

```
static int set_paranoid_check(int ck = 1);
```

Description

String searches use a hash value scheme to find the strings. There is a possibility that more than one string could hash to the same value. Calling *set_paranoid_check* with *ck* set to 1 forces checking the string found against the desired string with the C library function *strcmp*. When *set_paranoid_check* is called with *ck* set to 0, this final check is not made.

string::skip_whitespace

string class

Syntax

```
static int skip_whitespace(int sk = 1);
```

Description

Set to 1 to skip whitespace after a token read, 0 otherwise.

string::strip

string class

Syntax

```
TSubString strip( StripType s = Trailing, char c=' ');
```

Description

Strips away *c* characters from the beginning, end, or both (beginning and end) of string *s*, depending on StripType.

string::substr

string class

Form 1

```
string substr( size_t pos ) const;
```

Form 2

```
string substr( size_t pos, size_t n = NPOS ) const;
```

Description

Form 1: Creates a string containing a copy of the characters from *pos* to the end of the target string.

Form 2: Creates a string containing a copy of not more than *n* characters from *pos* to the end of the target string.

string::substring

string class

Form 1

```
TSubString substring( const char _FAR *cp );
```

Form 2

```
const TSubString substring( const char _FAR *cp ) const;
```

Form 3

```
TSubString substring( const char _FAR *cp, size_t start );
```

Form 4

```
const TSubString substring( const char _FAR *cp, size_t start ) const;
```

Description

Form 1: Creates a *TSubString* object containing a copy of the characters pointed to by **cp*.

Form 2: Creates a *TSubString* object containing a copy of the characters pointed to by **cp*.

Form 3: Creates a *TSubString* object containing a copy of the characters pointed to by **cp*, starting at character start.

Form 4: Creates a *TSubString* object containing a copy of the characters pointed to by **cp*, starting at character start.

string::to_lower

string class

Syntax

```
void to_lower();
```

Description

Changes the string to lowercase.

string::to_upper

string class

Syntax

```
void to_upper();
```

Description

Changes target string to uppercase.

string::assert_element

string class

Syntax

```
void assert_element( size_t pos ) const;
```

Description

Throws an *outofrange* exception if an invalid element is given.

string::assert_index

string class

Syntax

```
void assert_index( size_t pos ) const;
```

Description

Throws an *outofrange* exception if an invalid index is given.

string::cow

string class

Syntax

```
void cow();
```

Description

Copy-on-write. Multiple instances of a string can refer to the same piece of data as long as it is in a read-only situation. If a string writes to the data, then *cow* (copy-on-write) is called to make a copy if more than one string is referring to it.

string::valid_element

string class

Syntax

```
int valid_element( size_t pos ) const;
```

Description

Returns 1 if *pos* is an element of the string, 0 otherwise.

string::valid_index

string class

Syntax

```
int valid_index( size_t pos ) const;
```

Description

Returns 1 if *pos* is a valid index of the string, 0 otherwise.

string::operator =

string class

Syntax

```
string _FAR & operator=(const string _FAR &s);
```

Description

If the target string is the same object as the parameter passed to the assignment, the assignment operator does nothing. Otherwise it performs any actions necessary to free up resources allocated to the target string, then copies *s* into the target string.

string::operator +=

string class

Form 1

```
string _FAR & operator += (const string _FAR &s);
```

Form 2

```
string _FAR & operator+=(const char _FAR *cp);
```

Description

Form 1: Appends the contents of the string *s* to the target string.

Form 2: Appends the contents of *cp* to the target string.

string::operator +

string class

Syntax

```
friend string _Cdecl _FARFUNC operator+(const string _FAR &s, const char
  _FAR *cp);
```

Description

Concatenates string *s* and *cp*.

string::operator []

string class

Form 1

```
char _FAR & operator[](size_t pos);
```

Form 2

```
char operator[](size_t pos) const;
```

Description

Form 1: Returns a reference to the character at position *pos*.

Form 2: Returns the character at position *pos*.

string::operator ()

string class

Form 1
```
char _FAR & operator()(size_t pos);
```

Form 2
```
TSubString operator()(size_t start, size_t len);
```

Form 3
```
TSubString operator()(const TRegexp _FAR & re);
```

Form 4
```
TSubString operator()(const TRegexp _FAR & re, size_t start);
```

Form 5
```
char operator()(size_t pos) const;
```

Form 6
```
const TSubString operator()(size_t start, size_t len) const;
```

Form 7
```
const TSubString operator()(const TRegexp _FAR & pat) const;
```

Form 8
```
const TSubString operator()(const TRegexp _FAR & pat, size_t start) const;
```

Description

Form 1: Returns a reference to the character at position *pos*.

Form 2: Returns the substring beginning at location *start* and spanning *len* bytes.

Form 3: Returns the first occurrence of a substring matching regular expression *re*.

Form 4: Returns the first occurrence of a substring matching regular expression *re*, beginning at location *start*.

Form 5: Returns the character at position *pos*.

Form 6: Returns the substring beginning at location *start* and spanning *len* bytes.

Form 7: Returns the first occurrence of a substring matching regular expression *re*.

Form 8: Returns the first occurrence of a substring matching regular expression *re*, beginning at location *start*.

string::operator ==

string class

Form 1

```
friend int operator == ( const string _FAR &s1, const string _FAR &s2 );
```

Form 2

```
friend int operator == ( const string _FAR &s, const char _FAR *cp );
```

Form 3

```
friend int operator == ( const char _FAR *cp, const string _FAR &s );
```

Description

Form 1: Tests for equality of string *s1* and string *s2*. Two strings are equal if they have the same length, and if the same location in each string contains characters that compare equally. operator **==** returns a 1 to indicate that the strings are equal, and a 0 to indicate that they are not equal.

Form 2: Tests for equality of string *s1* and *char *cp*. The two are equal if they have the same length, and if the same location in each string contains characters that compare equally. operator **==** returns a 1 to indicate that the strings are equal, and a 0 to indicate that they are not equal.

Form 3: Tests for equality of string *s1* and *char *cp*. The two are equal if they have the same length, and if the same location in each string contains characters that compare equally. operator **==** returns a 1 to indicate that the strings are equal, and a 0 to indicate that they are not equal.

string::operator !=

string class

Form 1

```
friend int operator != ( const string _FAR &s1, const string _FAR &s2 );
```

Form 2

```
friend int operator != ( const string _FAR &s, const char _FAR *cp );
```

Form 3

```
friend int operator != ( const char _FAR *cp, const string _FAR &s );
```

Description

Form 1: Tests for inequality of strings *s1* and *s2*. Two strings are equal if they have the same length, and if the same location in each string contains characters that compare equally. operator **!=** returns a 1 to indicate that the strings are not equal, and a 0 to indicate that they are equal.

Form 2: Tests for inequality between string *s* and *char *cp*. The two are equal if they have the same length, and if the same location in each string contains characters that compare equally. operator **!=** returns a 1 to indicate that the strings are not equal, and a 0 to indicate that they are equal.

Form 3: Tests for inequality between string *s* and *char *cp*. The two are equal if they have the same length, and if the same location in each string contains characters that compare equally. operator **!=** returns a 1 to indicate that the strings are not equal, and a 0 to indicate that they are equal.

string::operator <

string class

Form 1

```
friend int operator < ( const string _FAR &s1, const string _FAR &s2 );
```

Form 2

```
friend int operator < ( const string _FAR &s, const char _FAR *cp );
```

Form 3

```
friend int operator < ( const char _FAR *cp, const string _FAR &s );
```

Description

Form 1: Compares the string *s1* to string *s2*. Returns 1 if string s1 is less than *s2*, 0 otherwise.

Form 2: Compares the string *s1* to **cp2*. Returns 1 if the left side of the expression is less than the right side, 0 otherwise.

Form 3: Compares the string *s1* to **cp2*. Returns 1 if the left side of the expression is less than the right side, 0 otherwise.

string::operator <=

string class

Form 1

```
friend int operator <= ( const string _FAR &s1, const string _FAR &s2 );
```

Form 2

```
friend int operator <= ( const string _FAR &s, const char _FAR *cp );
```

Form 3

```
friend int operator <= ( const char _FAR *cp, const string _FAR &s );
```

Description

Form 1: Compares the string *s1* to string *s2*. Returns 1 if string *s1* is less than or equal to *s2*, 0 otherwise.

Form 2: Compares string *s1* to **cp*. Returns 1 if the left side of the expression is less than or equal to the right side, 0 otherwise.

Form 3: Compares string *s1* to **cp*. Returns 1 if the left side of the expression is less than or equal to the right side, 0 otherwise.

string::operator >

string class

Form 1

```
friend int operator > ( const string _FAR &s1, const string _FAR &s2 );
```

Form 2

```
friend int operator > ( const string _FAR &s, const char _FAR *cp );
```

Form 3

```
friend int operator > ( const char _FAR *cp, const string _FAR &s );
```

Description

Form 1: Compares the string *s1* to string *s2*. Returns 1 if string *s1* is greater than *s2*, 0 otherwise.

Form 2: Compares string *s1* to **cp2*. Returns 1 if the left side of the expression is greater than the right side, 0 otherwise.

Form 3: Compares string *s1* to **cp2*. Returns 1 if the left side of the expression is greater than the right side, 0 otherwise.

string::operator >=

string class

Form 1

```
friend int operator >= ( const string _FAR &s1, const string _FR &s2 );
```

Form 2

```
friend int operator >= ( const string _FAR &s, const char _FAR *cp );
```

Form 3

```
friend int operator >= ( const char _FAR *cp, const string _FAR &s );
```

Description

Form 1: Compares string *s1* to string *s2*. Returns 1 if string *s1* is greater than or equal to *s2*, 0 otherwise.

Form 2: Compares string *s1* to **cp*. Returns 1 if the left side of the expression is greater than or equal to the right side, 0 otherwise.

Form 3: Compares string *s1* to **cp*. Returns 1 if the left side of the expression is greater than or equal to the right side, 0 otherwise.

string::operator >>

string class

Form 1

```
friend ipstream _FAR & operator >> ( ipstream _FAR & is, string _FAR &
  str );
```

Form 2

```
istream _FAR & _Cdecl _FARFUNC operator>>(istream _FAR &is, string _FAR
  &s);
```

Description

Form 1: Extracts string *str* from input stream *is*.

Form 2: Behaves the same as `operator >> (istream&, char *)`, and returns a reference to *is*.

string::operator <<

string class

Form 1

```
ostream _FAR & _Cdecl _FARFUNC operator<<(ostream _FAR &os, const string
  _FAR & s);
```

Form 2

```
opstream _FAR& _Cdecl operator << ( opstream _FAR & os, const string _FAR &
  str );
```

Description

Form 1: Behaves the same as `operator << (ostream&,const char *)` except that it does not terminate when it encounters a null character in the string. Returns a reference to *os*.

Form 2: Inserts string *str* into persistent output stream *os*.

string::operator +

string class

Form 1

```
string _Cdecl _FARFUNC operator + ( const char _FAR *cp, const string _FAR
  & s);
```

Form 2

```
string _Cdecl _FARFUNC operator + ( const string _FAR &s1, const string
  _FAR &s2 );
```

Description

Form 1: Concatenates **cp* and string *s*.

Form 2: Concatenates string *s1* and *s2*.

string::getline

string class

Form 1

```
istream _FAR & _Cdecl getline( istream _FAR &is, string _FAR &s );
```

Form 2

```
istream _FAR & _Cdecl getline( istream _FAR &is, string _FAR &s, char c );
```

Description

Form 1: Behaves the same as `istream::getline(chptr, NPOS)`, except that instead of storing into a *char* array, it stores into a *string*. *getline* returns a reference to *is*.

Form 2: Behaves the same as `istream::getline(cb, NPOS, c)`, except that instead of storing into a *char* array, it stores into a *string*. *getline* returns a reference to *is*.

string::to_lower

string class

Syntax

```
string  _Cdecl _FARFUNC to_lower(const string _FAR &s);
```

Description

Changes string *s* to lowercase.

string::to_upper

string class

Syntax

```
string _Cdecl _FARFUNC to_upper(const string _FAR &s);
```

Description

Changes string *s* to uppercase.

TSubString class

Header File
cstring.h

Syntax
```
class TSubString;
```

Description
The TSubString class allows selected substrings to be addressed.

Public Member Functions
get_at
is_null
length
put_at
start
to_lower
to_upper

Protected Member Functions
assert_element

Operators
=
==
!=
()
[]
!

TSubString::get_at

See Also TSubString class

Syntax

```
char get_at( size_t pos ) const;
```

Description

Returns the character at the specified position. If `pos > length()-1`, an exception is thrown.

See Also

TSubString::put_at

TSubString::is_null

TSubString class

Syntax

```
int is_null() const;
```

Description

Returns 1 if the string is empty, 0 otherwise.

TSubString::length

TSubString class

Syntax

```
size_t length() const;
```

Description

Returns the substring length.

TSubString::put_at

TSubString class

Syntax

```
void put_at( size_t pos, char c );
```

Description

Replaces the character at *pos* with *c*. If `pos == length()`, *put_at* appends *c* to the target string. If `pos > length()` an exception is thrown.

TSubString::start

TSubString class

Syntax

```
int start() const;
```

Description

Returns the index of the starting character.

TSubString::to_lower

TSubString class

Syntax

```
void to_lower();
```

Description

Changes the substring to lowercase.

TSubString::to_upper

TSubString class

Syntax

```
void to_upper();
```

Description

Changes the substring to uppercase.

TSubString::assert_element

TSubString class

Syntax

```
int assert_element(size_t pos) const;
```

Description

Returns 1 if *pos* represents a valid index into the substring, 0 otherwise.

TSubString::operator =

TSubString class

Syntax

```
TSubString _FAR & operator=(const string _FAR &s);
```

Description

Copies *s* into the target substring.

TSubString::operator ==

TSubString class

Form 1

```
int operator==(const char _FAR * cp) const;
```

Form 2

```
int operator==(const string _FAR & s) const;
```

Description

Form 1: Tests for equality between the target substring and **cp*. The two are equal if they have the same length, and if the same location in each string contains the same character. operator **==** returns a 1 to indicate that the strings are equal, and a 0 to indicate that they are not equal.

Form 2: Tests for equality between the target substring and string *s*. Two are equal if they have the same length, and if the same location in each string contains the same character. operator **==** returns a 1 to indicate that the strings are equal, and a 0 to indicate that they are not equal.

TSubString::operator !=

TSubString class

Form 1

```
int operator!=(const char _FAR * cp) const;
```

Form 2

```
int operator!=(const string _FAR & s) const;
```

Description

Form 1: Tests for inequality between the target string and **cp*. Two strings are equal if they have the same length, and if the same location in each string contains the same character. operator **!=** returns a 1 to indicate that the strings are not equal, and a 0 to indicate that they are equal.

Form 2: Tests for inequality between the target string and string *s*. Two strings are equal if they have the same length, and if the same location in each string contains the same character. operator **!=** returns a 1 to indicate that the strings are not equal, and a 0 to indicate that they are equal.

TSubString::operator ()

TSubString class

Form 1

```
char _FAR & operator()(size_t pos);
```

Form 2

```
char operator()(size_t pos) const;
```

Description

Form 1: Returns a reference to the character at position *pos*.

Form 2: Returns the character at position *pos*.

TSubString::operator []

TSubString class

Form 1

```
char _FAR & operator[](size_t pos);
```

Form 2

```
char operator[](size_t pos) const;
```

Description

Form 1: Returns a reference to the character at position *pos*.

Form 2: Returns the character at position *pos*.

TSubString::operator !

TSubString class

Syntax

```
int operator!() const;
```

Description

Detects null substrings. Returns 1 if the substring is not null.

TRegexp class

Header File
regexp.h

Description
This class represents regular expressions. *TRegexp* is a support class used by the string class for string searches.

Regular expressions use these special characters:

. [ ] - ^ * ? + $

General Rules
Characters other than the special characters match themselves. For example `"yardbird"` matches "yardbird".

A backslash (****) followed by a special character, matches the special character itself. For example `"Pardon\?"` matches "Pardon?".

The following escape codes can be used to match control characters:

`\b`	backspace
`\e`	Esc
`\f`	formfeed
`\n`	newline
`\r`	carriage return
`\t`	tab
`\xddd`	the literal hex number 0xddd
`\^x`	where x matches some control-code (for example \^c, \^c)

One-Character Regular Expressions
The **.** special character matches any single character except a newline character. For example `".ive"` would match "jive" or "five".

The **[** and **]** special characters are used to denote one-character regular expressions that will match any of the characters within the brackets. For example, `"[aeiou]"` would match either "a", "e", "i", "o", or "u".

The **-** special character is used within the `[ ]` special characters to denote a range of characters to match. For example, `"[ a-z ]"` would match on any lowercase alphabetic character between a and z.

The **^** special character is used to specify search for any character but those specified. For example, `"[ ^g-v ]"` would match on any lowercase alphabetic character NOT between g and v.

Multiple-Character Regular Expressions
The ***** special character following a one-character regular expression matches zero or more occurrences of that regular expression. For example, `"[ a-z ]*"` matches zero or more occurrences of lowercase alphabetic characters.

The **+** special character following a one-character regular expression matches one or more occurrences of that regular expression. For example, `"[ 0-9 ]+"` matches one or more occurrences of lowercase alphabetic characters.

The **?** special character specifies that the following character is optional. For example "xy?z" matches on "xy" or "xyz".

Regular expressions can be concatentated. For example, `"[ A-Z ][ a-z ]*"` matches

capitalized words.

Matching at the Beginning and End of a Line

If the **^** special character is at the beginning of a regular expression, then a match occurs only if the string is at the beginning of a line. For example, "`^[ A-Z ][ a-z ]*`" matches capitalized words at the beginning of a line.

If the **$** special character is at the end of a regular expression, the then a match occurs only if the string is at the end of a line. For example, "`[ A-Z ][ a-z ]*$`" matches capitalized words at the end of a line.

Type Definitions

StatVal

Public Constructors

TRegexp::TRegexp

Public Member Functions

find

status

Operators

=

TRegexp::StatVal

TRegexp class

Syntax

```
enum StatVal{OK=0, ILLEGAL, TOOLONG};
```

Description

StatVal enumerates the status conditions returned by TRegexp::status, where:

Status	Description
OK	Means the given regular expression is legal
ILLEGAL	Means the pattern was illegal
TOOLONG	Means the pattern exceeded maximum length (128)

TRegexp::TRegexp

TRegexp class

Form1:

```
TRegexp(const char far* cp);
```

Form2:

```
TRegexp(const TRegexp far& r);
```

Description

Form 1: Constructs a regular expression object using the pattern given pointed to by *cp*.

Form 2: Constructs a copy of regular expression object *r*.

TRegexp::find

TRegexp class

Syntax

```
size_t find(const string& s, size_t* len, size_t start=0)
```

Description

Finds the first instance in string *s* that matches this regular expression. The search begins at index *start*, and *len* returns the length of the matching string if found. The return value contains the index of the the beginning of the matching string. If no match is found, *len* is 0, and -1 is returned.

TRegexp::operator =

TRegexp class

Form 1

```
TRegexp& operator=(const TRegexp& r)
```

Form 2

```
TRegexp& operator=(const char* cp)
```

Description

Form 1: Sets this regular expression to a copy of *r*, using value semantics.

Form 2: Sets this regular expression to the pattern given by *cp*.

TRegexp::status

See Also TRegexp class

Syntax

```
StatVal status()
```

Description

Returns the status of this regular expression. Status values are:

Status	Description
OK	means the given regular expression is legal
ILLEGAL	means the pattern was illegal
TOOLONG	means the pattern exceeded maximum length (128)

See Also
TRegexp::StatVal