

Unit Test Documentation

Adrian Bedard

Jonathan Bedard

July 9, 2016

Contents

I	Unit Test Library	2
1	Introduction	3
1.1	Namespace test	3
1.2	Datastructures Testing	3
2	File Index	4
2.1	File List	4
3	File Documentation	6
3.1	DatastructuresTest.cpp File Reference	6
3.1.1	Detailed Description	6
3.2	DatastructuresTest.h File Reference	6
3.2.1	Detailed Description	6
3.3	defaultTestInit.cpp File Reference	7
3.3.1	Detailed Description	7
3.4	libraryTests.cpp File Reference	7
3.4.1	Detailed Description	7
3.5	libraryTests.h File Reference	7
3.5.1	Detailed Description	8
3.6	masterTestHolder.cpp File Reference	8
3.6.1	Detailed Description	8
3.7	masterTestHolder.h File Reference	8
3.7.1	Detailed Description	9
3.8	singleFunctionTest.cpp File Reference	9
3.9	singleFunctionTest.h File Reference	9
3.10	singleTest.cpp File Reference	9
3.10.1	Detailed Description	9
3.11	singleTest.h File Reference	10
3.11.1	Detailed Description	10
3.12	testSuite.cpp File Reference	10
3.12.1	Detailed Description	10
3.13	testSuite.h File Reference	10
3.13.1	Detailed Description	11
3.14	UnitTest.cpp File Reference	11
3.14.1	Detailed Description	11
3.15	UnitTest.h File Reference	11

3.15.1 Detailed Description	12
3.16 UnitTestLog.h File Reference	12
3.17 UnitTestMain.cpp File Reference	13
3.17.1 Detailed Description	13
3.17.2 Function Documentation	13
4 Class Index	14
4.1 Class List	14
5 Namespace Documentation	15
5.1 test Namespace Reference	15
5.1.1 Typedef Documentation	16
5.1.2 Function Documentation	16
5.1.3 Variable Documentation	18
6 Class Documentation	19
6.1 test::generalTestException Class Reference	19
6.1.1 Detailed Description	20
6.1.2 Constructor & Destructor Documentation	20
6.1.3 Member Function Documentation	20
6.1.4 Member Data Documentation	21
6.2 test::libraryTests Class Reference	21
6.2.1 Detailed Description	23
6.2.2 Constructor & Destructor Documentation	23
6.2.3 Member Function Documentation	23
6.2.4 Member Data Documentation	25
6.3 test::masterTestHolder Class Reference	26
6.3.1 Detailed Description	27
6.3.2 Constructor & Destructor Documentation	27
6.3.3 Member Function Documentation	27
6.3.4 Member Data Documentation	29
6.4 test::nullFunctionException Class Reference	29
6.4.1 Detailed Description	29
6.4.2 Constructor & Destructor Documentation	30
6.4.3 Member Function Documentation	30
6.5 test::singleFunctionTest Class Reference	30
6.5.1 Detailed Description	31
6.5.2 Constructor & Destructor Documentation	31
6.5.3 Member Function Documentation	31
6.5.4 Member Data Documentation	32
6.6 test::singleTest Class Reference	32
6.6.1 Detailed Description	32
6.6.2 Constructor & Destructor Documentation	33
6.6.3 Member Function Documentation	33
6.6.4 Member Data Documentation	34
6.7 test::testSuite Class Reference	34
6.7.1 Detailed Description	35
6.7.2 Constructor & Destructor Documentation	35
6.7.3 Member Function Documentation	36
6.7.4 Member Data Documentation	38

6.8	test::unknownException Class Reference	39
6.8.1	Detailed Description	39
6.8.2	Constructor & Destructor Documentation	39
6.8.3	Member Function Documentation	40
II	Datastructures Library	41
7	Introduction	42
7.1	Unit Testing	42
7.2	Namespace os	42
8	File Index	43
8.1	File List	43
9	File Documentation	46
9.1	abstractSorting.h File Reference	46
9.1.1	Detailed Description	46
9.2	arrayStack.h File Reference	47
9.2.1	Detailed Description	47
9.3	basicLock.cpp File Reference	47
9.3.1	Detailed Description	48
9.4	basicLock.h File Reference	48
9.4.1	Detailed Description	48
9.5	basicStructures.h File Reference	48
9.5.1	Detailed Description	49
9.6	constantPrinter.cpp File Reference	49
9.6.1	Detailed Description	49
9.7	constantPrinter.h File Reference	49
9.7.1	Detailed Description	50
9.8	Datastructures.h File Reference	50
9.8.1	Detailed Description	50
9.9	descriptiveException.h File Reference	50
9.9.1	Detailed Description	51
9.10	eventDriver.h File Reference	51
9.10.1	Detailed Description	51
9.11	iterator.h File Reference	51
9.11.1	Detailed Description	52
9.12	linkedQueue.h File Reference	52
9.12.1	Detailed Description	53
9.13	linkedStack.h File Reference	53
9.13.1	Detailed Description	53
9.14	lockable.h File Reference	53
9.14.1	Detailed Description	54
9.15	Locks.h File Reference	54
9.15.1	Detailed Description	54
9.16	macroAVL.h File Reference	54
9.16.1	Detailed Description	55
9.17	macroBoundedQueue.h File Reference	55
9.17.1	Detailed Description	56

9.18	macroDatastructuresInterface.h File Reference	56
9.18.1	Detailed Description	56
9.19	macroHash.h File Reference	56
9.19.1	Detailed Description	57
9.20	macroList.h File Reference	57
9.20.1	Detailed Description	58
9.21	macros.h File Reference	58
9.21.1	Detailed Description	58
9.22	macroVector.h File Reference	58
9.22.1	Detailed Description	59
9.23	matrix.h File Reference	59
9.23.1	Detailed Description	63
9.23.2	Function Documentation	63
9.24	nodeFrame.h File Reference	72
9.24.1	Detailed Description	73
9.25	nodeMacroHeader.h File Reference	73
9.25.1	Detailed Description	73
9.26	osLogger.cpp File Reference	73
9.26.1	Detailed Description	74
9.27	osLogger.h File Reference	74
9.27.1	Detailed Description	74
9.28	readWriteInterface.h File Reference	75
9.29	readWriteLock.cpp File Reference	75
9.29.1	Detailed Description	75
9.30	readWriteLock.h File Reference	75
9.30.1	Detailed Description	76
9.31	simpleHash.h File Reference	76
9.31.1	Detailed Description	76
9.32	smartPointer.h File Reference	76
9.32.1	Detailed Description	80
9.32.2	Function Documentation	80
9.33	specializedStructures.h File Reference	84
9.33.1	Detailed Description	84
9.34	threadCounter.cpp File Reference	84
9.34.1	Detailed Description	85
9.35	threadCounter.h File Reference	85
9.35.1	Detailed Description	85
9.36	threadLock.cpp File Reference	85
9.36.1	Detailed Description	86
9.37	threadLock.h File Reference	86
9.37.1	Detailed Description	86
9.38	vector2d.h File Reference	86
9.38.1	Detailed Description	87
9.39	vector3d.h File Reference	88
9.39.1	Detailed Description	88
10	Class Index	90
10.1	Class List	90

11 Namespace Documentation	93
11.1 os Namespace Reference	93
11.1.1 Typedef Documentation	98
11.1.2 Enumeration Type Documentation	100
11.1.3 Function Documentation	101
11.1.4 Variable Documentation	105
12 Class Documentation	107
12.1 os::ARRAY_STACK< dataType > Class Template Reference	107
12.1.1 Constructor & Destructor Documentation	108
12.1.2 Member Function Documentation	108
12.2 os::AVL_NODE< dataType > Class Template Reference	109
12.2.1 Detailed Description	111
12.2.2 Constructor & Destructor Documentation	111
12.2.3 Member Function Documentation	112
12.2.4 Member Data Documentation	117
12.3 os::AVL_TREE< dataType > Class Template Reference	118
12.3.1 Detailed Description	120
12.3.2 Constructor & Destructor Documentation	120
12.3.3 Member Function Documentation	120
12.3.4 Member Data Documentation	126
12.4 os::basicLock Class Reference	126
12.4.1 Detailed Description	127
12.4.2 Constructor & Destructor Documentation	127
12.4.3 Member Function Documentation	128
12.4.4 Member Data Documentation	129
12.5 os::BOUNDED_QUEUE< dataType > Class Template Reference	129
12.5.1 Detailed Description	131
12.5.2 Constructor & Destructor Documentation	131
12.5.3 Member Function Documentation	131
12.5.4 Member Data Documentation	137
12.6 os::BOUNDED_QUEUE_NODE< dataType > Class Template Reference	138
12.6.1 Detailed Description	139
12.6.2 Constructor & Destructor Documentation	139
12.6.3 Member Function Documentation	140
12.6.4 Member Data Documentation	143
12.7 os::constantPrinter Class Reference	143
12.7.1 Detailed Description	144
12.7.2 Constructor & Destructor Documentation	144
12.7.3 Member Function Documentation	144
12.7.4 Member Data Documentation	147
12.8 os::constIterator< dataType > Class Template Reference	147
12.8.1 Detailed Description	149
12.8.2 Constructor & Destructor Documentation	149
12.8.3 Member Function Documentation	150
12.8.4 Friends And Related Function Documentation	155
12.8.5 Member Data Documentation	155
12.9 os::DATASTRUCTURE< dataType > Class Template Reference	155
12.9.1 Detailed Description	156

12.9.2	Constructor & Destructor Documentation	156
12.9.3	Member Function Documentation	157
12.10	os::descriptiveException Class Reference	161
12.10.1	Detailed Description	161
12.10.2	Constructor & Destructor Documentation	161
12.10.3	Member Function Documentation	162
12.10.4	Member Data Documentation	162
12.11	os::errorPointer Class Reference	163
12.11.1	Detailed Description	163
12.11.2	Constructor & Destructor Documentation	163
12.11.3	Member Function Documentation	164
12.12	os::eventReceiver< senderType > Class Template Reference	165
12.12.1	Detailed Description	165
12.12.2	Constructor & Destructor Documentation	166
12.12.3	Member Function Documentation	166
12.12.4	Friends And Related Function Documentation	167
12.12.5	Member Data Documentation	167
12.13	os::eventSender< receiverType > Class Template Reference	167
12.13.1	Detailed Description	168
12.13.2	Constructor & Destructor Documentation	169
12.13.3	Member Function Documentation	169
12.13.4	Friends And Related Function Documentation	171
12.13.5	Member Data Documentation	171
12.14	os::HASH< dataType > Class Template Reference	171
12.14.1	Detailed Description	173
12.14.2	Constructor & Destructor Documentation	173
12.14.3	Member Function Documentation	173
12.14.4	Member Data Documentation	176
12.15	os::HASH_NODE< dataType > Class Template Reference	177
12.15.1	Detailed Description	178
12.15.2	Constructor & Destructor Documentation	178
12.15.3	Member Function Documentation	178
12.15.4	Member Data Documentation	181
12.16	os::indirectMatrix< dataType > Class Template Reference	181
12.16.1	Detailed Description	182
12.16.2	Constructor & Destructor Documentation	183
12.16.3	Member Function Documentation	184
12.16.4	Friends And Related Function Documentation	187
12.16.5	Member Data Documentation	187
12.17	os::iterator< dataType > Class Template Reference	187
12.17.1	Detailed Description	190
12.17.2	Constructor & Destructor Documentation	190
12.17.3	Member Function Documentation	190
12.17.4	Friends And Related Function Documentation	195
12.17.5	Member Data Documentation	195
12.18	os::iteratorBase< dataType > Class Template Reference	196
12.18.1	Detailed Description	197
12.18.2	Member Function Documentation	197
12.18.3	Friends And Related Function Documentation	200

12.19	os::iteratorBaseInterface< dataType > Class Template Reference	200
12.19.1	Detailed Description	201
12.19.2	Member Function Documentation	201
12.20	os::iteratorSource Class Reference	203
12.20.1	Detailed Description	204
12.20.2	Member Enumeration Documentation	204
12.20.3	Constructor & Destructor Documentation	205
12.20.4	Member Function Documentation	205
12.20.5	Member Data Documentation	206
12.21	os::LINKED_QUEUE< dataType > Class Template Reference	206
12.21.1	Detailed Description	207
12.21.2	Constructor & Destructor Documentation	207
12.21.3	Member Function Documentation	208
12.22	os::LINKED_STACK< dataType > Class Template Reference	209
12.22.1	Detailed Description	210
12.22.2	Constructor & Destructor Documentation	210
12.22.3	Member Function Documentation	210
12.23	os::LIST< dataType > Class Template Reference	211
12.23.1	Detailed Description	213
12.23.2	Constructor & Destructor Documentation	213
12.23.3	Member Function Documentation	214
12.23.4	Member Data Documentation	217
12.24	os::LIST_NODE< dataType > Class Template Reference	217
12.24.1	Detailed Description	219
12.24.2	Constructor & Destructor Documentation	219
12.24.3	Member Function Documentation	219
12.24.4	Member Data Documentation	220
12.25	os::lockable Class Reference	221
12.25.1	Detailed Description	222
12.25.2	Constructor & Destructor Documentation	222
12.25.3	Member Function Documentation	222
12.25.4	Member Data Documentation	224
12.26	os::matrix< dataType > Class Template Reference	224
12.26.1	Detailed Description	225
12.26.2	Constructor & Destructor Documentation	225
12.26.3	Member Function Documentation	227
12.26.4	Friends And Related Function Documentation	229
12.26.5	Member Data Documentation	229
12.27	os::NODE< dataType > Class Template Reference	230
12.27.1	Detailed Description	231
12.27.2	Constructor & Destructor Documentation	231
12.27.3	Member Function Documentation	232
12.27.4	Member Data Documentation	233
12.28	os::nodeFrame< dataType > Class Template Reference	233
12.28.1	Detailed Description	235
12.28.2	Constructor & Destructor Documentation	235
12.28.3	Member Function Documentation	236
12.28.4	Member Data Documentation	241
12.29	os::readWriteInterface Class Reference	241

12.29.1Detailed Description	242
12.29.2Constructor & Destructor Documentation	242
12.29.3Member Function Documentation	243
12.30os::readWriteLock Class Reference	244
12.30.1Detailed Description	246
12.30.2Member Enumeration Documentation	246
12.30.3Constructor & Destructor Documentation	246
12.30.4Member Function Documentation	247
12.30.5Member Data Documentation	249
12.31os::simpleHash< dataType > Class Template Reference	250
12.31.1Detailed Description	251
12.31.2Constructor & Destructor Documentation	251
12.31.3Member Function Documentation	252
12.31.4Member Data Documentation	256
12.32os::smart_ptr< dataType > Class Template Reference	257
12.32.1Detailed Description	259
12.32.2Constructor & Destructor Documentation	260
12.32.3Member Function Documentation	262
12.32.4Member Data Documentation	268
12.33os::SORTED_LIST< dataType > Class Template Reference	269
12.33.1Detailed Description	270
12.33.2Member Function Documentation	270
12.33.3Member Data Documentation	270
12.34os::threadCounter Class Reference	270
12.34.1Detailed Description	271
12.34.2Constructor & Destructor Documentation	271
12.34.3Member Function Documentation	272
12.34.4Member Data Documentation	273
12.35os::threadLock Class Reference	273
12.35.1Detailed Description	274
12.35.2Constructor & Destructor Documentation	274
12.35.3Member Function Documentation	275
12.35.4Member Data Documentation	275
12.36os::UNSORTED_LIST< dataType > Class Template Reference	276
12.36.1Detailed Description	276
12.36.2Member Function Documentation	277
12.36.3Member Data Documentation	277
12.37os::VECTOR< dataType > Class Template Reference	277
12.37.1Detailed Description	279
12.37.2Constructor & Destructor Documentation	279
12.37.3Member Function Documentation	280
12.37.4Member Data Documentation	284
12.38os::vector2d< dataType > Class Template Reference	284
12.38.1Detailed Description	286
12.38.2Constructor & Destructor Documentation	286
12.38.3Member Function Documentation	287
12.38.4Member Data Documentation	295
12.39os::vector3d< dataType > Class Template Reference	295
12.39.1Detailed Description	297

12.39.2	Constructor & Destructor Documentation	297
12.39.3	Member Function Documentation	298
12.39.4	Member Data Documentation	307
12.40	os::VECTOR_NODE< dataType > Class Template Reference	307
12.40.1	Detailed Description	308
12.40.2	Constructor & Destructor Documentation	309
12.40.3	Member Function Documentation	309
12.40.4	Member Data Documentation	312

Part I

Unit Test Library

Chapter 1

Introduction

The UnitTest library contains classes which preform automated unit tests while a project is under development. Utilizing C++ exceptions, the UnitTest library separates its test battery into libraries tested, suites in libraries and tests in suites. The UnitTest library iterates through instantiated libraries running every test suite in the library.

1.1 Namespace test

The test namespace is designed to hold all of the classes and functions related to unit testing. Classes and functions in the test namespace should not be included in the final release application. It is expected that libraries add to this namespace and place their own testing assets here. Note that the test namespace uses elements from the os namespace, all of these elements are defined in the Datastructures library.

1.2 Datastructures Testing

The Datastructures library is rigorously unit tested by the UnitTest library, and the Datastructures unit tests are automatically included in any system unit test unless specifically removed. The Datastructures UnitTests are particularly important because the Datastructures library serves as a base for memory management and data organization. These tests fall broadly into two categories: deterministic and random.

Deterministic tests preform the exact same test every iteration. Deterministic tests are used to ensure that specific functions and operators are returning expected data. Deterministic tests don't merely identify the existence of an error, but usually identify the precise nature of the error as well.

Random tests use a random number generator to preform a unique test with every iteration. This allows unit tests to, over time, catch edge cases with complex data structures. In contrast to deterministic tests, random testing will usually not identify the precise nature of the error.

Note that as a general rule, the implementation of tests is not documented. The location of test suites is documented, through both .h and .cpp files, but the classes and functions which make up these tests are not included.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

DatastructuresTest.cpp	
Datastructures library test implementation	6
DatastructuresTest.h	
Datastructures library test	6
defaultTestInit.cpp	
Default UnitTest initializer function	7
Exceptions.cpp	
Implements the error throwing functions	??
Exceptions.h	
Includes unit test exception headers	??
generalTestException.h	
General unit test exception	??
libraryTests.cpp	
Library tests implementations	7
libraryTests.h	
Library tests class	7
masterTestHolder.cpp	
MasterTestHolder singleton implementations	8
masterTestHolder.h	
MasterTestHolder singleton	8
nullFunctionException.h	??
singleFunctionTest.cpp	9
singleFunctionTest.h	9
singleTest.cpp	
Single test class implementation	9
singleTest.h	
Single test class	10
testSuite.cpp	
Single test class	10
testSuite.h	
Single test class	10

UnitTest.cpp	
Unit Test logging and global functions	11
UnitTest.h	
Unit Test header file	11
UnitTestLog.h	12
UnitTestMain.cpp	
UnitTest entry point	13
unknownException.h	
Unknown test exception	??

Chapter 3

File Documentation

3.1 DatastructuresTest.cpp File Reference

Datastructures library test implementation.

3.1.1 Detailed Description

Datastructures library test implementation.

Author

Jonathan Bedard

Date

7/7/2016

Bug No known bugs.

Implements the Datastructures library test. These tests are designed to guarantee the functionality of each of the elements in the Datastructures library.

3.2 DatastructuresTest.h File Reference

Datastructures library test.

3.2.1 Detailed Description

Datastructures library test.

Author

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

Contains the declaration of the Datastructures library test. Note that this library test is automatically added to all Unit Test executables.

3.3 defaultTestInit.cpp File Reference

Default UnitTest initializer function.

3.3.1 Detailed Description

Default UnitTest initializer function.

Author

Jonathan Bedard

Date

2/12/2016

Bug No known bugs.

By default, this is the implementation of the UnitTest initializer function which binds UnitTest libraries to the test battery. This function should be defined by a file in the library of the main entry point of the application.

3.4 libraryTests.cpp File Reference

Library tests implementations.

3.4.1 Detailed Description

Library tests implementations.

Jonathan Bedard

Date

7/3/2016

Bug No known bugs.

This file contains implementations for the library test base class. Consult **libraryTests.h** (p. 7) for details.

3.5 libraryTests.h File Reference

Library tests class.

Classes

- class **test::libraryTests**

Library test group.

Namespaces

- **test**

3.5.1 Detailed Description

Library tests class.

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

This file contains declarations for the library test base class, which each library should re-implement to provide testing functionality.

3.6 masterTestHolder.cpp File Reference

masterTestHolder singleton implementations

3.6.1 Detailed Description

masterTestHolder singleton implementations

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

This file contains implementations for the **test::masterTestHolder** (p. 26) singleton class. Consult **masterTestHolder.h** (p. 8) for details.

3.7 masterTestHolder.h File Reference

masterTestHolder singleton

Classes

- class **test::masterTestHolder**

Unit Test singleton.

Namespaces

- **test**

3.7.1 Detailed Description

masterTestHolder singleton

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

This file contains declarations for the **test::masterTestHolder** (p. 26) singleton class. This file represents the top level of the Unit Test driver classes.

3.8 singleFunctionTest.cpp File Reference

3.9 singleFunctionTest.h File Reference

Classes

- class **test::singleFunctionTest**
Single unit test from function.

Namespaces

- **test**

Typedefs

- typedef void(* **test::testFunction**) ()
Typedef for single test function.

3.10 singleTest.cpp File Reference

Single test class implementation.

3.10.1 Detailed Description

Single test class implementation.

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

This file contains implementation for a single unit test. Consult singleTest.h for details.

3.11 singleTest.h File Reference

Single test class.

Classes

- class **test::singleTest**
Single unit test class.

Namespaces

- **test**

3.11.1 Detailed Description

Single test class.

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

This file contains declarations for a single unit test. Unit tests can be defined as separate class or a simple test function, as defined by **singleFunctionTest.h** (p. 9).

3.12 testSuite.cpp File Reference

Single test class.

3.12.1 Detailed Description

Single test class.

Jonathan Bedard

Date

7/3/2016

Bug No known bugs.

This file contains declarations for a test suite. Consult **testSuite.h** (p. 10) for details.

3.13 testSuite.h File Reference

Single test class.

Classes

- class **test::testSuite**

Test suite.

Namespaces

- **test**

3.13.1 Detailed Description

Single test class.

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

This file contains declarations for a test suite. Test suites contain lists of unit tests.

3.14 UnitTest.cpp File Reference

Unit Test logging and global functions.

3.14.1 Detailed Description

Unit Test logging and global functions.

Author

Jonathan Bedard

Date

7/3/2016

Bug No known bugs.

Implements logging in the test namespace. Implements a number of global test functions used for initializing and ending a Unit Test battery.

3.15 UnitTest.h File Reference

Unit Test header file.

Namespaces

- **test**

Functions

- void **test::startTests** ()
Print out header for Unit Tests.
- void **test::endTestsError** (os::errorPointer except)
End tests in error.
- void **test::endTestsSuccess** ()
End tests successfully.
- void **test::testInit** (int argc=0, char **argv=NULL)
Test initialization.

3.15.1 Detailed Description

Unit Test header file.

Author

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

Packages all headers required for the UnitTest library and declares a number of global test functions used for initializing and ending a Unit Test battery.

3.16 UnitTestLog.h File Reference

Namespaces

- **test**

Functions

- std::ostream & **test::testout_func** ()
Standard out object for test namespace.
- std::ostream & **test::testerr_func** ()
Standard error object for test namespace.

Variables

- os::smart_ptr< std::ostream > **test::testout_ptr**
Standard out pointer for test namespace.
- os::smart_ptr< std::ostream > **test::testerr_ptr**
Standard error pointer for test namespace.

3.17 UnitTestMain.cpp File Reference

UnitTest entry point.

Functions

- `int main (int argc, char **argv)`

3.17.1 Detailed Description

UnitTest entry point.

Author

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

This file is the entry point to a UnitTestExe application. The application created with this file will initialize and run the test battery. If successful, the application will return 0, else, it will return -1.

3.17.2 Function Documentation

`int main (int argc, char ** argv)`

Chapter 4

Class Index

4.1 Class List

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

test::generalTestException	
Base class for test exceptions	19
test::libraryTests	
Library test group	21
test::masterTestHolder	
Unit Test singleton	26
test::nullFunctionException	
NULL function exception class	29
test::singleFunctionTest	
Single unit test from function	30
test::singleTest	
Single unit test class	32
test::testSuite	
Test suite	34
test::unknownException	
Unknown exception class	39

Chapter 5

Namespace Documentation

5.1 test Namespace Reference

Classes

- class **generalTestException**
Base class for test exceptions.
- class **libraryTests**
Library test group.
- class **masterTestHolder**
Unit Test singleton.
- class **nullFunctionException**
NULL function exception class.
- class **singleFunctionTest**
Single unit test from function.
- class **singleTest**
Single unit test class.
- class **testSuite**
Test suite.
- class **unknownException**
Unknown exception class.

Typedefs

- typedef void(* **testFunction**) ()
Typedef for single test function.

Functions

- void **startTests** ()
Print out header for Unit Tests.
- void **endTestsError** (os::errorPointer except)

End tests in error.

- void **endTestsSuccess** ()

End tests successfully.

- void **testInit** (int argc=0, char **argv=NULL)

Test initialization.

- std::ostream & **testout_func** ()

Standard out object for test namespace.

- std::ostream & **testerr_func** ()

Standard error object for test namespace.

- void **throwGeneralTestException** (const std::string &description, const std::string &location)

Creates and throws a test exception.

- void **throwNullFunctionException** (const std::string &location)

Creates and throws a null function exception.

- void **throwUnknownException** (const std::string &location)

Creates and throws an unknown exception.

Variables

- os::smart_ptr< std::ostream > **testout_ptr**

Standard out pointer for test namespace.

- os::smart_ptr< std::ostream > **testerr_ptr**

Standard error pointer for test namespace.

5.1.1 Typedef Documentation

```
typedef void(* test::testFunction) ()
```

Typedef for single test function.

This typedef defines what a single test function looks like. For simplicity, a single unit test can be defined by a function of this type instead of inheriting from **test::singleTest** (p. 32).

Returns

void

5.1.2 Function Documentation

```
void test::endTestsError ( os::errorPointer except )
```

End tests in error.

Prints out a global division block line of '=' characters, then the information provided in the exception passed to the function then another global division block

Parameters

in	except	Exception which caused the error
----	--------	----------------------------------

Returns

void

void test::endTestsSuccess ()

End tests successfully.

Prints out a global division block line of '=' characters, then the test results data provided by the **test::masterTestHolder** (p. 26) then another global division block

Returns

void

void test::startTests ()

Print out header for Unit Tests.

Prints out a global division block line of '=' characters, then 'Unit Test Battery' and then another global division block.

Returns

void

std::ostream& test::testerr_func ()

Standard error object for test namespace.

#define statements allow the user to call this function with "test::testerr." Logging is achieved by using "test::testerr" as one would use "std::cerr."

void test::testInit (int argc = 0, char ** argv = NULL)

Test initialization.

This function is re-implemented by each executable which uses the UnitTest library. This function is used to bind all of the library tests, except the Datastructures library test.

Returns

void

std::ostream& test::testout_func ()

Standard out object for test namespace.

#define statements allow the user to call this function with "test::testout." Logging is achieved by using "test::testout" as one would use "std::cout."

void test::throwGeneralTestException (const std::string & description, const std::string & location)

Creates and throws a test exception.

Parameters

in	<i>description</i>	Error description
in	<i>location</i>	Source of the error, file and function

Returns

void

void test::throwNullFunctionException (const std::string & location)

Creates and throws a null function exception.

Parameters

in	<i>location</i>	Source of the error, file and function
----	-----------------	--

Returns

void

void test::throwUnknownException (const std::string & location)

Creates and throws an unknown exception.

Parameters

in	<i>location</i>	Source of the error, file and function
----	-----------------	--

Returns

void

5.1.3 Variable Documentation

os::smart_ptr<std::ostream> test::testerr_ptr

Standard error pointer for test namespace.

This std::ostream is used as standard error for the test namespace. This pointer can be swapped out to programmatically redirect standard error for the test namespace.

os::smart_ptr<std::ostream> test::testout_ptr

Standard out pointer for test namespace.

This std::ostream is used as standard out for the test namespace. This pointer can be swapped out to programmatically redirect standard out for the test namespace.

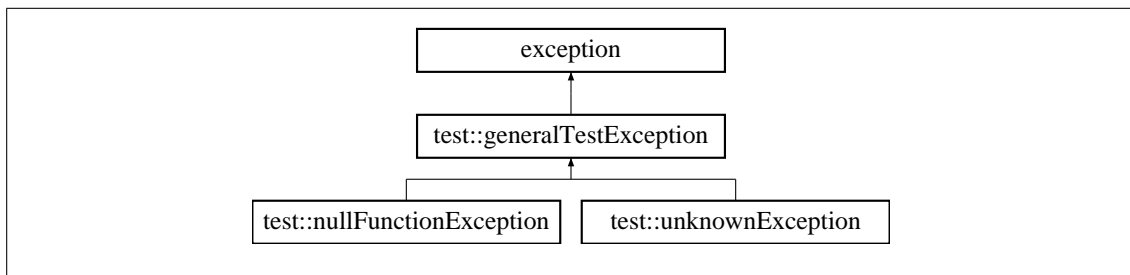
Chapter 6

Class Documentation

6.1 test::generalTestException Class Reference

Base class for test exceptions.

Inheritance diagram for test::generalTestException:



Public Member Functions

- **generalTestException** (const std::string &err, const std::string &loc)
Construct exception with error and location.
- virtual ~**generalTestException** () throw ()
Virtual destructor.
- const char * **what** () const final throw ()
std::exception overload
- const std::string & **getLocation** () const
Location description.
- const std::string & **getString** () const
Error description.

Static Public Member Functions

- static void **throwException** (const std::string &description, const std::string &location)
Creates and throws a test exception.

Private Attributes

- `std::string location`
The location where the error came from.
- `std::string _error`
A description of the error.
- `std::string total_error`
Combination of the error and location.

6.1.1 Detailed Description

Base class for test exceptions.

This class defines an exception which has a location. Because this class holds multiple `std::string` objects, the error description can be dynamically set.

6.1.2 Constructor & Destructor Documentation

```
test::generalTestException::generalTestException ( const std::string & err, const std::string & loc )  
[inline]
```

Construct exception with error and location.

Constructs the exception with an error string and a location string. Also builds the **test::generalTestException::total_error** (p. 21) string for use by the "what()" function.

Parameters

in	<i>err</i>	Error string
in	<i>loc</i>	Location string

```
virtual test::generalTestException::~~generalTestException ( ) throw ( ) [inline], [virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

6.1.3 Member Function Documentation

```
const std::string& test::generalTestException::getLocation ( ) const [inline]
```

Location description.

Returns

test::generalTestException::location (p. 21)

```
const std::string& test::generalTestException::getString ( ) const [inline]
```

Error description.

Returns

test::generalTestException::_error (p. 21)

```
static void test::generalTestException::throwException ( const std::string & description, const
std::string & location ) [static]
```

Creates and throws a test exception.

Parameters

in	<i>description</i>	Error description
in	<i>location</i>	Source of the error, file and function

Returns

void

```
const char* test::generalTestException::what ( ) const throw ) [inline], [final]
```

std::exception overload

Overloaded from std::exception. This function outputs the complete description, which contains both the error description and location description.

Returns

character pointer to the complete description

6.1.4 Member Data Documentation

```
std::string test::generalTestException::_error [private]
```

A description of the error.

```
std::string test::generalTestException::location [private]
```

The location where the error came from.

```
std::string test::generalTestException::total_error [private]
```

Combination of the error and location.

This string is constructed in the constructor so that "what()" can refer to a location in memory. This std::string is a combination of **test::generalTestException::_error** (p. 21) and **test::generalTestException::location** (p. 21).

6.2 test::libraryTests Class Reference

Library test group.

Public Member Functions

- **libraryTests** (std::string ln)
Library test constructor.
- virtual **~libraryTests** ()
Virtual destructor.
- void **runTests** () throw (os::errorPointer)
Runs all of the test suites.
- virtual void **onSetup** ()
Runs on shutdown of the group.
- virtual void **onTeardown** ()
Runs on teardown of the group.
- void **logBegin** ()
Logs the beginning of a library test.
- bool **logEnd** (os::errorPointer except=NULL)
Logs the end of a library test.
- int **getNumSuites** () const
Number of suites in the set.
- int **getNumSuccess** () const
Number of suites successfully completed.
- int **getNumRun** () const
Number of suites attempted to run.
- void **pushSuite** (os::smart_ptr< **testSuite** > suite)
Add suite to the set.
- void **removeSuite** (os::smart_ptr< **testSuite** > suite)
Remove suite from the set.
- int **compare** (const **libraryTests** &cmp) const
Compares two library test suites.
- **operator size_t** () const
size_t cast for the library

Private Attributes

- std::string **libName**
Name of library to be tested.
- os::pointerAVLTree< **testSuite** > **suiteList**
Set of test suites.
- int **suitesCompleted**
Number of suites successfully completed.
- int **suitesRun**
Number of suites attempted to run.

6.2.1 Detailed Description

Library test group.

This class contains a set of test suites which are designed to a specific library. Each library must define it's own version of this class in-order to be tested.

6.2.2 Constructor & Destructor Documentation

```
test::libraryTests::libraryTests ( std::string ln )
```

Library test constructor.

This constructor initializes the number of suites completed and number of suites run to 0, along with sets the name of library being tested.

Parameters

in	<i>ln</i>	Name of library to be tested
----	-----------	------------------------------

```
virtual test::libraryTests::~~libraryTests ( ) [inline], [virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

6.2.3 Member Function Documentation

```
int test::libraryTests::compare ( const libraryTests & cmp ) const [inline]
```

Compares two library test suites.

Parameters

in	<i>cmp</i>	Element to compare
----	------------	--------------------

Returns

0 if equal, 1 if greater than, -1 if less than

```
int test::libraryTests::getNumRun ( ) const [inline]
```

Number of suites attempted to run.

Returns

test::libraryTests::suitesRun (p. 26)

```
int test::libraryTests::getNumSuccess ( ) const [inline]
```

Number of suites successfully completed.

Returns

test::libraryTests::suitesCompleted (p. 26)

```
int test::libraryTests::getNumSuites ( ) const [inline]
```

Number of suites in the set.

Returns

```
test::libraryTests::suiteList.size()
```

```
void test::libraryTests::logBegin ( )
```

Logs the beginning of a library test.

Outputs the name of the library to be tested along with a line break made of '+' characters.

Returns

```
void
```

```
bool test::libraryTests::logEnd ( os::errorPointer except = NULL )
```

Logs the end of a library test.

Outputs the number of suites run and how many of these suites were both successful and how many of these suites failed.

Returns

```
True if all suites successful, else false
```

```
virtual void test::libraryTests::onSetup ( ) [inline], [virtual]
```

Runs on shutdown of the group.

Each library group calls this function as it starts up, allowing groups to define actions performed to setup the group.

Returns

```
void
```

```
virtual void test::libraryTests::onTeardown ( ) [inline], [virtual]
```

Runs on teardown of the group.

Guaranteed to run even if the group itself fails. A custom tear-down for the group can re-implement this class.

Returns

```
void
```

test::libraryTests::operator size_t () const

size_t cast for the library

Returns

size_t hash of the library

void test::libraryTests::pushSuite (os::smart_ptr< **testSuite** > suite) [inline]

Add suite to the set.

Adds a **test::testSuite** (p. 34) to the set of suites to be tested.

Parameters

in	<i>suite</i>	Test suite to be added to set
----	--------------	-------------------------------

Returns

void

void test::libraryTests::removeSuite (os::smart_ptr< **testSuite** > suite) [inline]

Remove suite from the set.

Removes a **test::testSuite** (p. 34) from the set of suites to be tested.

Parameters

in	<i>suite</i>	Test suite to be removed from the set
----	--------------	---------------------------------------

Returns

void

void test::libraryTests::runTests () throw os::errorPointer)

Runs all of the test suites.

Runs all test suites bound to this class. Each suite should manage its own errors, but it is possible that this function will throw an error of type os::errorPointer.

Returns

void

6.2.4 Member Data Documentation

std::string test::libraryTests::libName [private]

Name of library to be tested.

os::pointerAVLTree<**testSuite**> test::libraryTests::suiteList [private]

Set of test suites.

int test::libraryTests::suitesCompleted [private]

Number of suites successfully completed.

int test::libraryTests::suitesRun [private]

Number of suites attempted to run.

6.3 test::masterTestHolder Class Reference

Unit Test singleton.

Public Member Functions

- virtual ~**masterTestHolder** ()
Virtual destructor.
- bool **runTests** () throw (os::errorPointer)
Runs all of the library tests.
- int **getNumLibs** () const
Number of libraries in the set.
- int **getNumSuccess** () const
Number of libraries successfully completed.
- int **getNumRun** () const
Number of libraries attempted to run.
- void **pushLibrary** (os::smart_ptr< **libraryTests** > lib)
Add library to the set.
- void **removeLibrary** (os::smart_ptr< **libraryTests** > lib)
Remove library from the set.

Static Public Member Functions

- static **masterTestHolder** & **singleton** ()
Singleton access.

Private Member Functions

- **masterTestHolder** ()
Private constructor.

Private Attributes

- `os::pointerAVLTree< libraryTests > libraryList`
Set of library tests.
- `int libsCompleted`
Number of libraries successfully completed.
- `int libsRun`
Number of libraries attempted to run.

6.3.1 Detailed Description

Unit Test singleton.

This class contains a set of library tests. Every library test must add itself to this class in-order to be tested. The `test::masterTestHolder::runTests()` (p. 28) function runs all of the library tests.

6.3.2 Constructor & Destructor Documentation

```
test::masterTestHolder::masterTestHolder ( ) [private]
```

Private constructor.

The `test::masterTestHolder` (p. 26) class is a singleton class. This constructor initializes the number of libraries completed and number of libraries run to 0.

```
virtual test::masterTestHolder::~~masterTestHolder ( ) [inline], [virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

6.3.3 Member Function Documentation

```
int test::masterTestHolder::getNumLibs ( ) const [inline]
```

Number of libraries in the set.

Returns

```
test::masterTestHolder::libraryList.size()
```

```
int test::masterTestHolder::getNumRun ( ) const [inline]
```

Number of libraries attempted to run.

Returns

```
test::masterTestHolder::libsRun (p. 29)
```

int test::masterTestHolder::getNumSuccess () const [inline]

Number of libraries successfully completed.

Returns

test::masterTestHolder::libsCompleted (p. 29)

void test::masterTestHolder::pushLibrary (os::smart_ptr< **libraryTests** > lib) [inline]

Add library to the set.

Adds a **test::libraryTests** (p. 21) to the set of library tests to be tested.

Parameters

in	<i>lib</i>	Library test to be added to set
----	------------	---------------------------------

Returns

void

void test::masterTestHolder::removeLibrary (os::smart_ptr< **libraryTests** > lib) [inline]

Remove library from the set.

Removes a **test::libraryTests** (p. 21) from the set of library tests to be tested.

Parameters

in	<i>lib</i>	Library test to be removed from the set
----	------------	---

Returns

void

bool test::masterTestHolder::runTests () throw os::errorPointer)

Runs all of the library tests.

Runs all library tests bound to this class. Each library should manage its own errors, but it is possible that this function will throw an error of type os::errorPointer.

Returns

True if all the tests were successful, else, false

static **masterTestHolder**& test::masterTestHolder::singleton () [static]

Singleton access.

This function constructs the single reference to the **test::masterTestHolder** (p. 26) class if needed. Then, it returns a pointer to this single reference.

Returns

Singleton reference to **test::masterTestHolder** (p. 26)

6.3.4 Member Data Documentation

os::pointerAVLTree<**libraryTests**> test::masterTestHolder::libraryList [private]

Set of library tests.

int test::masterTestHolder::libsCompleted [private]

Number of libraries successfully completed.

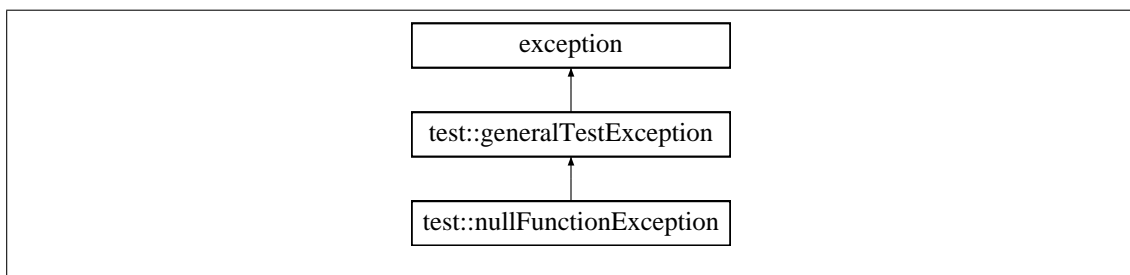
int test::masterTestHolder::libsRun [private]

Number of libraries attempted to run.

6.4 test::nullFunctionException Class Reference

NULL function exception class.

Inheritance diagram for test::nullFunctionException:



Public Member Functions

- **nullFunctionException** (const std::string &loc)
Construct exception with location.
- **~nullFunctionException** () final throw ()
Virtual destructor.

Static Public Member Functions

- static void **throwException** (const std::string &location)
Creates and throws a null function exception.

6.4.1 Detailed Description

NULL function exception class.

This class defines the common exception case where a NULL function pointer is received.

6.4.2 Constructor & Destructor Documentation

`test::nullFunctionException::nullFunctionException (const std::string & loc) [inline]`

Construct exception with location.

Constructs a **test::generalTestException** (p. 19) with the provided location and the static string for a NULL function exception.

Parameters

in	<i>loc</i>	Location string
-----------	------------	-----------------

`test::nullFunctionException::~~nullFunctionException () throw) [inline], [final]`

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

6.4.3 Member Function Documentation

`static void test::nullFunctionException::throwException (const std::string & location) [static]`

Creates and throws a null function exception.

Parameters

in	<i>location</i>	Source of the error, file and function
-----------	-----------------	--

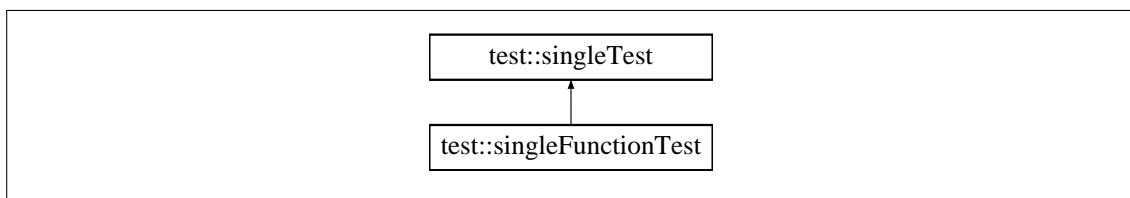
Returns

void

6.5 test::singleFunctionTest Class Reference

Single unit test from function.

Inheritance diagram for test::singleFunctionTest:



Public Member Functions

- **singleFunctionTest** (std::string tn, **testFunction** f)

Single unit test constructor.

- **~singleFunctionTest** () final

Virtual destructor.

- void **test** () final throw (os::errorPointer)

Call unit test function.

Private Attributes

- **testFunction func**

Reference to unit test function.

6.5.1 Detailed Description

Single unit test from function.

This class allows a **test::singleTest** (p. 32) to be defined by a single test function.

6.5.2 Constructor & Destructor Documentation

test::singleFunctionTest::singleFunctionTest (std::string tn, **testFunction** f)

Single unit test constructor.

Parameters

in	<i>tn</i>	Name of unit test
in	<i>f</i>	Function which defines test

test::singleFunctionTest::~~singleFunctionTest () [inline], [final]

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

6.5.3 Member Function Documentation

void test::singleFunctionTest::test () throw os::errorPointer) [final], [virtual]

Call unit test function.

Calls the function bound to this class in the constructor pointed to by **test::singleFunctionTest::func** (p. 32). If the function pointed to by the function pointer throws an exception, this function will throw the same exception.

Returns

void

Reimplemented from **test::singleTest** (p. 34).

6.5.4 Member Data Documentation

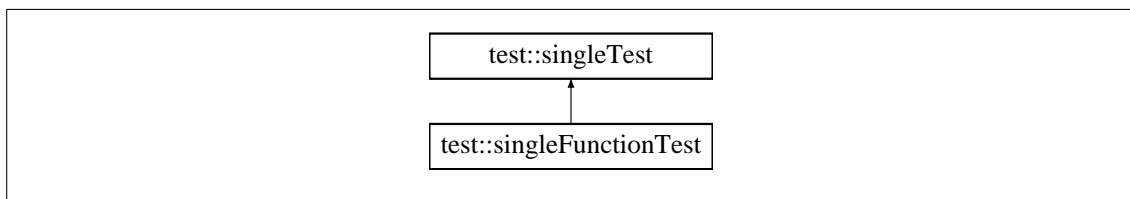
testFunction test::singleFunctionTest::func [private]

Reference to unit test function.

6.6 test::singleTest Class Reference

Single unit test class.

Inheritance diagram for test::singleTest:



Public Member Functions

- **singleTest** (std::string tn)
Single unit test constructor.
- virtual ~**singleTest** ()
Virtual destructor.
- virtual void **setupTest** ()
Preforms any test set-up.
- virtual void **test** ()
Preforms core unit-test.
- virtual void **teardownTest** ()
Preforms any test tear-down.
- void **logBegin** ()
Prints out the name of the test.
- bool **logEnd** (os::errorPointer except=NULL)
Logs errors for test.

Private Attributes

- std::string **testName**
Name of unit test.

6.6.1 Detailed Description

Single unit test class.

This class acts as the base class for all unit tests. It inherits from the `os::ptrComp` class to allow it to be inserted into abstract data-structures.

6.6.2 Constructor & Destructor Documentation

`test::singleTest::singleTest (std::string tn)`

Single unit test constructor.

Parameters

in	<i>tn</i>	Name of unit test
----	-----------	-------------------

`virtual test::singleTest::~~singleTest () [inline], [virtual]`

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

6.6.3 Member Function Documentation

`void test::singleTest::logBegin ()`

Prints out the name of the test.

Returns

void

`bool test::singleTest::logEnd (os::errorPointer except = NULL)`

Logs errors for test.

If the passed exception is NULL, no logging is preformed. Otherwise, the "what()" function of the exception is printed. This function return true if NULL is passed as the exception.

Parameters

in	<i>except</i>	Exception to be printed, NULL by default
----	---------------	--

Returns

True if except is NULL

`virtual void test::singleTest::setupTest () [inline], [virtual]`

Preforms any test set-up.

This function is designed to preform any set-up a test requires. This is especially useful if a class of tests require the same set-up routine. This function assumes that the **test::testSuite** (p. 34) will catch exceptions in this function if they are thrown.

Returns

void

virtual void test::singleTest::teardownTest () [inline], [virtual]

Preforms any test tear-down.

This function is designed to preform any tear-down a test requires. This is especially useful if a class of tests require the same tear-down routine. This function assumes that the **test::testSuite** (p. 34) will catch exceptions in this function if they are thrown.

Returns

void

virtual void test::singleTest::test () [virtual]

Preforms core unit-test.

This function is designed to preform the actual unit test. This function assumes that the **test::testSuite** (p. 34) will catch exceptions in this function if they are thrown.

Returns

void

Reimplemented in **test::singleFunctionTest** (p. 31).

6.6.4 Member Data Documentation

std::string test::singleTest::testName [private]

Name of unit test.

6.7 test::testSuite Class Reference

Test suite.

Public Member Functions

- **testSuite** (std::string sn)
Test suite constructor.
- virtual ~**testSuite** ()
Virtual destructor.
- void **runTests** () throw (os::errorPointer)
Runs all of the tests.
- virtual void **onSetup** ()
Runs on shutdown.
- virtual void **onTeardown** ()
Runs on teardown of the suite.
- void **logBegin** ()
Logs the beginning of a suite test.
- bool **logEnd** (os::errorPointer except=NULL)
Logs the end of a suite test.

- int **getNumTests** () const
Number of tests in the set.
- int **getNumSuccess** () const
Number of tests successfully completed.
- int **getNumRun** () const
Number of tests attempted to run.
- void **pushTest** (os::smart_ptr< **singleTest** > tst)
Add test to the set.
- void **removeTest** (os::smart_ptr< **singleTest** > tst)
Remove test to the set.
- virtual void **pushTest** (std::string str, **testFunction** tst)
Add test to the set.
- int **compare** (const **testSuite** &cmp) const
Compares two library test suites.
- **operator size_t** () const
size_t cast for the library

Private Attributes

- std::string **suiteName**
Name of test suite.
- os::pointerUnsortedList< **singleTest** > **testList**
Set of tests.
- int **testsCompleted**
Number of tests successfully completed.
- int **testsRun**
Number of tests attempted to run.

6.7.1 Detailed Description

Test suite.

Defines a named test suite which has a selection of tests to run.

6.7.2 Constructor & Destructor Documentation

test::testSuite::testSuite (std::string sn)

Test suite constructor.

This constructor initializes the number of tests completed and number of tests run to 0, along with sets the name of suite being tested.

Parameters

in	sn	Name of suite to be tested
----	----	----------------------------

```
virtual test::testSuite::~~testSuite ( ) [inline], [virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

6.7.3 Member Function Documentation

```
int test::testSuite::compare ( const testSuite & cmp ) const [inline]
```

Compares two library test suites.

Parameters

in	<i>cmp</i>	Element to compare
----	------------	--------------------

Returns

0 if equal, 1 if greater than, -1 if less than

```
int test::testSuite::getNumRun ( ) const [inline]
```

Number of tests attempted to run.

Returns

test::testSuite::testsRun (p. 39)

```
int test::testSuite::getNumSuccess ( ) const [inline]
```

Number of tests successfully completed.

Returns

test::testSuite::testsCompleted (p. 38)

```
int test::testSuite::getNumTests ( ) const [inline]
```

Number of tests in the set.

Returns

test::testSuite::testList.size()

```
void test::testSuite::logBegin ( )
```

Logs the beginning of a suite test.

Outputs the name of the suite to be tested along with a line break made of '-' characters.

Returns

void

`bool test::testSuite::logEnd (os::errorPointer except = NULL)`

Logs the end of a suite test.

Outputs the number of tests run and how many of these tests were both successful and how many of these tests failed.

Returns

True if all tests successful, else false

`virtual void test::testSuite::onSetup () [inline], [virtual]`

Runs on shutdown.

Each suite calls this function as it starts up, allowing suites to define actions performed to setup the suite.

Returns

void

`virtual void test::testSuite::onTeardown () [inline], [virtual]`

Runs on teardown of the suite.

Guaranteed to run even if the suite itself fails. A custom tear-down for the suite can re-implement this class.

Returns

void

`test::testSuite::operator size_t () const`

size_t cast for the library

Returns

size_t hash of the library

`void test::testSuite::pushTest (os::smart_ptr< singleTest > tst) [inline]`

Add test to the set.

Adds a **test::singleTest** (p. 32) to the set of tests to be tested.

Parameters

in	<i>tst</i>	Test to be added to set
----	------------	-------------------------

Returns

void

`virtual void test::testSuite::pushTest (std::string str, testFunction tst) [inline], [virtual]`

Add test to the set.

Adds a **test::testFunction** (p. 16) to the set of tests to be tested. Constructs a **test::singleTest** (p. 32) from a function and a test name

Parameters

in	<i>str</i>	Test name
in	<i>tst</i>	Function which defines test

Returns

void

```
void test::testSuite::removeTest ( os::smart_ptr< singleTest > tst ) [inline]
```

Remove test to the set.

Removes a **test::singleTest** (p. 32) from the set of tests to be tested.

Parameters

in	<i>tst</i>	Test to be removed from the set
----	------------	---------------------------------

Returns

void

```
void test::testSuite::runTests ( ) throw os::errorPointer)
```

Runs all of the tests.

Runs all tests bound to this class. This function catches exceptions thrown by **test::singleTest** (p. 32) and logs the results.

Returns

void

6.7.4 Member Data Documentation

```
std::string test::testSuite::suiteName [private]
```

Name of test suite.

```
os::pointerUnsortedList<singleTest> test::testSuite::testList [private]
```

Set of tests.

```
int test::testSuite::testsCompleted [private]
```

Number of tests successfully completed.

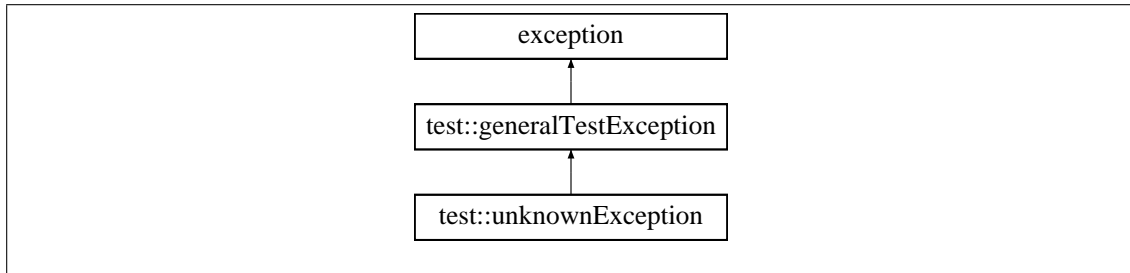
int test::testSuite::testsRun [private]

Number of tests attempted to run.

6.8 test::unknownException Class Reference

Unknown exception class.

Inheritance diagram for test::unknownException:



Public Member Functions

- **unknownException** (const std::string &loc)
Construct exception with location.
- **~unknownException** () final throw ()
Virtual destructor.

Static Public Member Functions

- static void **throwException** (const std::string &location)
Creates and throws an unknown exception.

6.8.1 Detailed Description

Unknown exception class.

This class defines the common exception case where the precise nature of the exception is unknown.

6.8.2 Constructor & Destructor Documentation

test::unknownException::unknownException (const std::string & loc) [inline]

Construct exception with location.

Constructs a **test::generalTestException** (p. 19) with the provided location and the static string for an unknown exception.

Parameters

in	loc	Location string
----	-----	-----------------

`test::unknownException::~~unknownException () throw) [inline], [final]`

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

6.8.3 Member Function Documentation

`static void test::unknownException::throwException (const std::string & location) [static]`

Creates and throws an unknown exception.

Parameters

in	<i>location</i>	Source of the error, file and function
----	-----------------	--

Returns

void

Part II

Datastructures Library

Chapter 7

Introduction

The Datastructures library contains a series of utility classes and template classes used for the organization and management of data. Most notably, this library allow dynamic memory management through the `smart_ptr` class and provides a flexible runtime data container in the `ads` (Abstract Data Structure) template and its children.

7.1 Unit Testing

The testing of the Datastructures library is contained within the `UnitTest` library. Since the `UnitTest` library uses the functionality of the Datastructures library, the Datastructures library cannot be dependent on the `UnitTest` library as the `UnitTest` library is already dependent on the Datastructures library

7.2 Namespace `os`

Datastructures extends the `os` namespace. The `os` namespace is designed for tools, algorithms and data-structures used in programs of all types. Structures in this library do not implement operating system specific interfaces such as sockets and file I/O. The `osMechanics` library also extends the `os` namespace.

Chapter 8

File Index

8.1 File List

Here is a list of all files with brief descriptions:

abstractSorting.h	Template for sorting arrays	46
ads.h	Abstract datastructure interface	??
adsFrame.h	Include headers with macro management	??
arrayStack.h	Defines a stack with an array	47
asyncAVL.h	Asynchronous AVL tree	??
AVL.h	AVL tree	??
basicLock.cpp	Implementation basicLock	47
basicLock.h	Non-recursive lock	48
basicStructures.h	Include structure headers with macro management	48
constantPrinter.cpp	Constant printing support, implementation	49
constantPrinter.h	Constant printing support	49
Datastructures.h	Master Datastructures header file	50
descriptiveException.h	Basic descriptive exception	50
eventDriver.h	Event sender and receiver	51
iterator.h	Defines an iterator	51

linkedQueue.h	Defines a queue with a linked list	52
linkedStack.h	Defines a stack with a linked list	53
list.h	Doubly Linked List	??
lockable.h	Defines the lockable interface	53
Locks.h	Constant locking support	54
macroAVL.h	Defines a generalized AVL tree	54
macroBoundedQueue.h	Defines a bounded queue	55
macroDatastructuresInterface.h	Defines the interface used by datastructures	56
macroHash.h	Defines an iterable hash table	56
macroList.h	Defines a basic linked list	57
macros.h	Support macro declarations	58
macroVector.h	Defines an expandable array	58
matrix.h	Matrix templates	59
nodeFrame.h	Declares a framework for nodes	72
nodeMacroHeader.h	Node macro header	73
osLogger.cpp	Logging for os namespace, implementation	73
osLogger.h	Logging for os namespace	74
readWriteInterface.h	75
readWriteLock.cpp	Implements the read/write lock	75
readWriteLock.h	Declaration of the read/write lock	75
set.h	Smart Set	??
simpleHash.h	Constant locking support	76
smartPointer.h	Template declaration of os::smart_ptr (p. 257)	76
specializedStructures.h	Include specialized structure headers with macro management	84
threadCounter.cpp	Implementation of thread counter	84

threadCounter.h	
Thread counter declaration	85
threadLock.cpp	
Implementation of thread locks	85
threadLock.h	
Recursive lock	86
vector2d.h	
Vector templates	86
vector3d.h	
Vector templates	88

Chapter 9

File Documentation

9.1 abstractSorting.h File Reference

Template for sorting arrays.

Namespaces

- **os**

Functions

- `template<class dataType >`
`int os::defaultCompare (const dataType &v1, const dataType &v2)`
Basic compare.
- `template<class dataType >`
`int os::pointerCompare (const smart_ptr< dataType > &ptr1, const smart_ptr< dataType > &ptr2) throw ()`
Pointer compare.
- `template<class dataType >`
`void os::quicksort (dataType *arr, unsigned length, int(*sort_comparision)(const dataType &, const dataType &)=&defaultCompare)`
Template quick-sort.
- `template<class dataType >`
`void os::pointerQuicksort (smart_ptr< smart_ptr< dataType > > arr, unsigned length, int(*sort_comparision)(const smart_ptr< dataType > &, const smart_ptr< dataType > &)=&pointerCompare)`
Template for quick-sort, pointer version.

9.1.1 Detailed Description

Template for sorting arrays.

Author

Jonathan Bedard

Date

7/7/2016

Bug No known bugs.

This file contains a template class definition of an AVL tree and its nodes. This tree has insertion, search and deletion of $O(\log(n))$ where n is the number of nodes in the tree. This tree is thread safe.

9.2 arrayStack.h File Reference

Defines a stack with an array.

Classes

- class **os::ARRAY_STACK**< **dataType** >

Namespaces

- **os**

9.2.1 Detailed Description

Defines a stack with an array.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This header should only be included from **specializedStructures.h** (p. 84). It defines a number of different types of stacks to be used.

9.3 basicLock.cpp File Reference

Implementation basicLock.

9.3.1 Detailed Description

Implementation `basicLock`.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Implements the non-recursive lock defined in **`basicLock.h`** (p. 48).

9.4 `basicLock.h` File Reference

Non-recursive lock.

Classes

- class **`os::basicLock`**
Basic lock Wraps the `std::mutex` class allowing it to be accessed in a constant way.

Namespaces

- **`os`**

9.4.1 Detailed Description

Non-recursive lock.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Defines a simple, non-recursive, mutex.

9.5 `basicStructures.h` File Reference

Include structure headers with macro management.

9.5.1 Detailed Description

Include structure headers with macro management.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Includes a series of headers defining specific containers with different macro declarations defined. Each macro type defines the structures in different ways.

9.6 constantPrinter.cpp File Reference

Constant printing support, implementation.

9.6.1 Detailed Description

Constant printing support, implementation.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This file implements **os::constantPrinter** (p. 143). Consult staticConstantPrinter.h for detailed documentation.

9.7 constantPrinter.h File Reference

Constant printing support.

Classes

- class **os::constantPrinter**
Prints constant arrays to files.

Namespaces

- **os**

9.7.1 Detailed Description

Constant printing support.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This file contains a class which helps facilitate printing massive tables of constants. It outputs .h and .cpp files with configured arrays of constants.

9.8 Datastructures.h File Reference

Master Datastructures header file.

9.8.1 Detailed Description

Master Datastructures header file.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

All of the headers in the Datastructures library are held in this file. When using the Datastructures library, it is expected that this header is included instead of the individual required headers.

9.9 descriptiveException.h File Reference

Basic descriptive exception.

Classes

- class **os::descriptiveException**
Basic exception with description.

Namespaces

- **os**

9.9.1 Detailed Description

Basic descriptive exception.

Author

Jonathan Bedard

Date

6/3/2016

Bug No known bugs.

Defines a class which allows for a simple string to describe an exception

9.10 eventDriver.h File Reference

Event sender and receiver.

Classes

- class **os::eventSender**< **receiverType** >
Class which enables event sending.
- class **os::eventReceiver**< **senderType** >
Class which enables event receiving.

Namespaces

- **os**

9.10.1 Detailed Description

Event sender and receiver.

Author

Jonathan Bedard

Date

7/7/2016

Bug No known bugs.

Both **os::eventReceiver** (p. 165) and **os::eventSender** (p. 167) are experimental classes and have not been tested or utilized.

9.11 iterator.h File Reference

Defines an iterator.

Classes

- class **os::iteratorSource**
Base class for iterator source.
- class **os::iteratorBaseInterface< dataType >**
Iterator base interface.
- class **os::iteratorBase< dataType >**
Iterator source.
- class **os::iterator< dataType >**
Generalized iterator.
- class **os::constIterator< dataType >**
Generalized constant iterator.

Namespaces

- **os**

9.11.1 Detailed Description

Defines an iterator.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Defines an iterator which wraps a node for intuitive usage

9.12 linkedQueue.h File Reference

Defines a queue with a linked list.

Classes

- class **os::LINKED_QUEUE< dataType >**
Queue built on-top of a list.

Namespaces

- **os**

9.12.1 Detailed Description

Defines a queue with a linked list.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This header should only be included from **specializedStructures.h** (p. 84). It defines a number of different types of queues to be used.

9.13 linkedStack.h File Reference

Defines a stack with a linked list.

Classes

- class **os::LINKED_STACK**< **dataType** >
Stack built on-top of a list.

Namespaces

- **os**

9.13.1 Detailed Description

Defines a stack with a linked list.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This header should only be included from **specializedStructures.h** (p. 84). It defines a number of different types of stacks to be used.

9.14 lockable.h File Reference

Defines the lockable interface.

Classes

- class **os::lockable**

An interface defining a class which can be locked.

Namespaces

- **os**

9.14.1 Detailed Description

Defines the lockable interface.

Author

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

All class which can be locked should utilize the interface defined in this header.

9.15 Locks.h File Reference

Constant locking support.

9.15.1 Detailed Description

Constant locking support.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Includes the headers for all locking structures.

9.16 macroAVL.h File Reference

Defines a generalized AVL tree.

Classes

- class **os::AVL_NODE**< **dataType** >
AVL tree node Node used in an AVL tree. This node allows both iteration and random access.
- class **os::AVL_TREE**< **dataType** >
Template AVL Tree definition.

Namespaces

- **os**

9.16.1 Detailed Description

Defines a generalized AVL tree.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This header should only be included from **basicStructures.h** (p.48). It defines a number of different types of AVL trees to be used.

9.17 macroBoundedQueue.h File Reference

Defines a bounded queue.

Classes

- class **os::BOUNDED_QUEUE**< **dataType** >
*Template **BOUNDED_QUEUE** (p. 129) definition.*
- class **os::BOUNDED_QUEUE_NODE**< **dataType** >
***BOUNDED_QUEUE** (p. 129) node.*

Namespaces

- **os**

9.17.1 Detailed Description

Defines a bounded queue.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This header should only be included from **specializedStructures.h** (p. 84). It defines a number of different types of bounded queues to be used.

9.18 macroDatastructuresInterface.h File Reference

Defines the interface used by datastructures.

Classes

- class **os::DATASTRUCTURE**< **dataType** >
Object node definition.

Namespaces

- **os**

9.18.1 Detailed Description

Defines the interface used by datastructures.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This header should only be included from the **adsFrame.h** (p. ??). It defines a number of different forms of the generalized data-structure template

9.19 macroHash.h File Reference

Defines an iterable hash table.

Classes

- class **os::HASH< dataType >**
Iterable hash table.
- class **os::HASH_NODE< dataType >**
Hash node.

Namespaces

- **os**

9.19.1 Detailed Description

Defines an iterable hash table.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This header should only be included from **specializedStructures.h** (p. 84). It defines a number of different types of hashes to be used.

9.20 macroList.h File Reference

Defines a basic linked list.

Classes

- class **os::LIST_NODE< dataType >**
List node Node used in linked lists to store each element.
- class **os::LIST< dataType >**
List framework Template for a linked list, insertion is fully defined in subsequent extensions of this class.
- class **os::UNSORTED_LIST< dataType >**
Unsorted list A basic list which remains unsorted unless the sort function is called on it.
- class **os::SORTED_LIST< dataType >**
Sorted list A basic list which remains unsorted unless the sort function is called on it.

Namespaces

- **os**

9.20.1 Detailed Description

Defines a basic linked list.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This header should only be included from **basicStructures.h** (p.48). It defines a number of different types of vectors to be used.

9.21 macros.h File Reference

Support macro declarations.

9.21.1 Detailed Description

Support macro declarations.

Author

Jonathan Bedard

Date

5/16/2016

Bug No known bugs.

Declares a number of macros for general use. Most notable are the `POINTER_COMPARE` macro and the `COMPARE_OPERATORS` macro. Both of these macros rely off of `CURRENT_CLASS` being defined, and both aid in the definition of comparison operators.

9.22 macroVector.h File Reference

Defines an expandable array.

Classes

- class **os::VECTOR< dataType >**
Template vector definition.
- class **os::VECTOR_NODE< dataType >**
Vector node.

Namespaces

- **os**

9.22.1 Detailed Description

Defines an expandable array.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This header should only be included from **basicStructures.h** (p.48). It defines a number of different types of vectors to be used.

9.23 matrix.h File Reference

Matrix templates.

Classes

- class **os::matrix**< **dataType** >
Raw matrix.
- class **os::indirectMatrix**< **dataType** >
Indirect matrix.

Namespaces

- **os**

Functions

- template<class dataType >
bool **os::compareSize** (const matrix< dataType > &m1, const matrix< dataType > &m2) throw
()
Compares the size of two matrices.
- template<class dataType >
bool **os::compareSize** (const indirectMatrix< dataType > &m1, const matrix< dataType >
&m2) throw ()
Compares the size of two matrices.
- template<class dataType >
bool **os::compareSize** (const matrix< dataType > &m1, const indirectMatrix< dataType >
&m2) throw ()

Compares the size of two matrices.

- `template<class dataType >`
`bool os::compareSize (const indirectMatrix< dataType > &m1, const indirectMatrix< dataType > &m2) throw ()`

Compares the size of two matrices.

- `template<class dataType >`
`bool os::testCross (const matrix< dataType > &m1, const matrix< dataType > &m2) throw ()`

Tests if the cross-product is a legal operation.

- `template<class dataType >`
`bool os::testCross (const indirectMatrix< dataType > &m1, const matrix< dataType > &m2) throw ()`

Tests if the cross-product is a legal operation.

- `template<class dataType >`
`bool os::testCross (const matrix< dataType > &m1, const indirectMatrix< dataType > &m2) throw ()`

Tests if the cross-product is a legal operation.

- `template<class dataType >`
`bool os::testCross (const indirectMatrix< dataType > &m1, const indirectMatrix< dataType > &m2) throw ()`

Tests if the cross-product is a legal operation.

- `template<class dataType >`
`bool operator== (const os::matrix< dataType > &m1, const os::matrix< dataType > &m2) throw ()`

Test for equality.

- `template<class dataType >`
`bool operator== (const os::indirectMatrix< dataType > &m1, const os::matrix< dataType > &m2) throw ()`

Test for equality.

- `template<class dataType >`
`bool operator== (const os::matrix< dataType > &m1, const os::indirectMatrix< dataType > &m2) throw ()`

Test for equality.

- `template<class dataType >`
`bool operator== (const os::indirectMatrix< dataType > &m1, const os::indirectMatrix< dataType > &m2) throw ()`

Test for equality.

- `template<class dataType >`
`bool operator!= (const os::matrix< dataType > &m1, const os::matrix< dataType > &m2) throw ()`

Test for inequality.

- `template<class dataType >`
`bool operator!= (const os::indirectMatrix< dataType > &m1, const os::matrix< dataType > &m2) throw ()`

Test for inequality.

- `template<class dataType >`
`bool operator!= (const os::matrix< dataType > &m1, const os::indirectMatrix< dataType >`
`&m2) throw ()`
Test for inequality.
- `template<class dataType >`
`bool operator!= (const os::indirectMatrix< dataType > &m1, const os::indirectMatrix< data↵`
`Type > &m2) throw ()`
Test for inequality.
- `template<class dataType >`
`os::matrix< dataType > operator+ (const os::matrix< dataType > &m1, const os::matrix<`
`dataType > &m2) throw ()`
Addition.
- `template<class dataType >`
`os::matrix< dataType > operator+ (const os::indirectMatrix< dataType > &m1, const os↵`
`::matrix< dataType > &m2) throw ()`
Addition.
- `template<class dataType >`
`os::matrix< dataType > operator+ (const os::matrix< dataType > &m1, const os::indirect↵`
`Matrix< dataType > &m2) throw ()`
Addition.
- `template<class dataType >`
`os::indirectMatrix< dataType > operator+ (const os::indirectMatrix< dataType > &m1, const`
`os::indirectMatrix< dataType > &m2) throw ()`
Addition.
- `template<class dataType >`
`os::matrix< dataType > operator- (const os::matrix< dataType > &m1, const os::matrix<`
`dataType > &m2) throw ()`
Subtraction.
- `template<class dataType >`
`os::matrix< dataType > operator- (const os::indirectMatrix< dataType > &m1, const os↵`
`::matrix< dataType > &m2) throw ()`
Subtraction.
- `template<class dataType >`
`os::matrix< dataType > operator- (const os::matrix< dataType > &m1, const os::indirect↵`
`Matrix< dataType > &m2) throw ()`
Subtraction.
- `template<class dataType >`
`os::indirectMatrix< dataType > operator- (const os::indirectMatrix< dataType > &m1, const`
`os::indirectMatrix< dataType > &m2) throw ()`
Subtraction.
- `template<class dataType >`
`os::matrix< dataType > operator* (const os::matrix< dataType > &m1, const os::matrix<`
`dataType > &m2) throw ()`
Cross-product.

- `template<class dataType >`
os::matrix< dataType > **operator*** (const **os::indirectMatrix**< dataType > &m1, const **os::matrix**< dataType > &m2) throw ()
Cross-product.
- `template<class dataType >`
os::matrix< dataType > **operator*** (const **os::matrix**< dataType > &m1, const **os::indirectMatrix**< dataType > &m2) throw ()
Cross-product.
- `template<class dataType >`
os::indirectMatrix< dataType > **operator*** (const **os::indirectMatrix**< dataType > &m1, const **os::indirectMatrix**< dataType > &m2) throw ()
Cross-product.
- `template<class dataType >`
os::matrix< dataType > **operator*** (const dataType &d1, const **os::matrix**< dataType > &m1) throw ()
Scalar multiplication.
- `template<class dataType >`
os::matrix< dataType > **operator*** (const **os::matrix**< dataType > &m1, const dataType &d1) throw ()
Scalar multiplication.
- `template<class dataType >`
os::matrix< dataType > **operator/** (const **os::matrix**< dataType > &m1, const dataType &d1) throw ()
Scalar division.
- `template<class dataType >`
os::indirectMatrix< dataType > **operator*** (const dataType &d1, const **os::indirectMatrix**< dataType > &m1) throw ()
Scalar multiplication.
- `template<class dataType >`
os::indirectMatrix< dataType > **operator*** (const **os::indirectMatrix**< dataType > &m1, const dataType &d1) throw ()
Scalar multiplication.
- `template<class dataType >`
os::indirectMatrix< dataType > **operator/** (const **os::indirectMatrix**< dataType > &m1, const dataType &d1) throw ()
Scalar division.
- `template<class dataType >`
std::ostream & **operator**<< (std::ostream &os, const **os::matrix**< dataType > &dt) throw ()
Prints out a matrix.
- `template<class dataType >`
std::ostream & **operator**<< (std::ostream &os, const **os::indirectMatrix**< dataType > &dt) throw ()
Prints out a matrix.

9.23.1 Detailed Description

Matrix templates.

Author

Jonathan Bedard

Date

5/15/2016

Bug No known bugs.

This file contains two template class definitions for matrices. One of these is an "indirect" matrix, meaning that the is an array of pointers, and the other is a direct matrix, meaning the matrix is an array of values.

9.23.2 Function Documentation

```
template<class dataType > bool operator!= ( const os::matrix< dataType > & m1, const os::matrix< dataType > & m2 ) throw )
```

Test for inequality.

Calls '==' and then inverts the result. Depends on the '!=' operator of dataType.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>m2</i>	Raw matrix reference

Returns

False if exactly equivalent

```
template<class dataType > bool operator!= ( const os::indirectMatrix< dataType > & m1, const os::matrix< dataType > & m2 ) throw )
```

Test for inequality.

Calls '==' and then inverts the result. Depends on the '!=' operator of dataType.

Parameters

in	<i>m1</i>	Indirect matrix reference
in	<i>m2</i>	Raw matrix reference

Returns

False if exactly equivalent


```
template<class dataType > bool operator!= ( const os::matrix< dataType > & m1, const os::indirectMatrix< dataType > & m2 ) throw )
```

Test for inequality.

Calls '==' and then inverts the result. Depends on the '!=' operator of dataType.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>m2</i>	Indirect matrix reference

Returns

False if exactly equivalent

```
template<class dataType > bool operator!= ( const os::indirectMatrix< dataType > & m1, const os::indirectMatrix< dataType > & m2 ) throw )
```

Test for inequality.

Calls '==' and then inverts the result. Depends on the '!=' operator of dataType.

Parameters

in	<i>m1</i>	Indirect matrix reference
in	<i>m2</i>	Indirect matrix reference

Returns

False if exactly equivalent

```
template<class dataType > os::matrix<dataType> operator* ( const os::matrix< dataType > & m1, const os::matrix< dataType > & m2 ) throw )
```

Cross-product.

Performs the cross-product. The cross-product is undefined if the `_width` of m1 does not equal the `_height` of m2. If the cross-product is undefined, a matrix of size (0,0) will be returned. Depends on the '*' and '+=' operator of the dataType.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>m2</i>	Raw matrix reference

Returns

m1 x m2 (raw matrix)

```
template<class dataType > os::matrix<dataType> operator* ( const os::indirectMatrix<
dataType > & m1, const os::matrix< dataType > & m2 ) throw )
```

Cross-product.

Preforms the cross-product. The cross- product is undefined if the `_width` of m1 does not equal the `_height` of m2. If the cross-product is undefined, a matrix of size (0,0) will be returned. Depends on the '*' and '+=' operator of the dataType.

Parameters

in	<i>m1</i>	Indirect matrix reference
in	<i>m2</i>	Raw matrix reference

Returns

m1 x m2 (raw matrix)

```
template<class dataType > os::matrix<dataType> operator* ( const os::matrix< dataType > &
m1, const os::indirectMatrix< dataType > & m2 ) throw )
```

Cross-product.

Preforms the cross-product. The cross- product is undefined if the `_width` of m1 does not equal the `_height` of m2. If the cross-product is undefined, a matrix of size (0,0) will be returned. Depends on the '*' and '+=' operator of the dataType.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>m2</i>	Indirect matrix reference

Returns

m1 x m2 (raw matrix)

```
template<class dataType > os::indirectMatrix<dataType> operator* ( const os::indirectMatrix<
dataType > & m1, const os::indirectMatrix< dataType > & m2 ) throw )
```

Cross-product.

Preforms the cross-product. The cross- product is undefined if the `_width` of m1 does not equal the `_height` of m2. If the cross-product is undefined, a matrix of size (0,0) will be returned. Depends on the '*' and '+=' operator of the dataType.

Parameters

in	<i>m1</i>	Indirect matrix reference
in	<i>m2</i>	Indirect matrix reference

Returns

$m1 \times m2$ (indirect matrix)

```
template<class dataType > os::matrix<dataType> operator* ( const dataType & d1, const os::matrix< dataType > & m1 ) throw )
```

Scalar multiplication.

Multiplies a matrix by a constant. This function depends on the '*' operator of the dataType.

Parameters

in	<i>d1</i>	Scalar data type
in	<i>m1</i>	Raw matrix reference

Returns

$d1 * m1$ (raw matrix)

```
template<class dataType > os::matrix<dataType> operator* ( const os::matrix< dataType > & m1, const dataType & d1 ) throw )
```

Scalar multiplication.

Multiplies a matrix by a constant. This function depends on the '*' operator of the dataType.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>d1</i>	Scalar data type

Returns

$d1 * m1$ (raw matrix)

```
template<class dataType > os::indirectMatrix<dataType> operator* ( const dataType & d1, const os::indirectMatrix< dataType > & m1 ) throw )
```

Scalar multiplication.

Multiplies an indirect matrix by a constant. This function depends on the '*' operator of the dataType.

Parameters

in	<i>d1</i>	Scalar data type
in	<i>m1</i>	Indirect matrix reference

Returns

$d1 * m1$ (indirect matrix)

```
template<class dataType > os::indirectMatrix<dataType> operator* ( const os::indirectMatrix<
dataType > & m1, const dataType & d1 ) throw )
```

Scalar multiplication.

Multiplies an indirect matrix by a constant. This function depends on the '*' operator of the data↵
Type.

Parameters

in	<i>m1</i>	Indirect matrix reference
in	<i>d1</i>	Scalar data type

Returns

$d1 * m1$ (indirect matrix)

```
template<class dataType > os::matrix<dataType> operator+ ( const os::matrix< dataType > &
m1, const os::matrix< dataType > & m2 ) throw )
```

Addition.

Preforms matrix addition. Matrix addition is undefined if the two matrices are of different size.
If the operation is undefined, a matrix of size (0,0) will be returned. Depends on the '+' operator of
dataType.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>m2</i>	Raw matrix reference

Returns

$m1 + m2$ (raw matrix)

```
template<class dataType > os::matrix<dataType> operator+ ( const os::indirectMatrix<
dataType > & m1, const os::matrix< dataType > & m2 ) throw )
```

Addition.

Preforms matrix addition. Matrix addition is undefined if the two matrices are of different size.
If the operation is undefined, a matrix of size (0,0) will be returned. Depends on the '+' operator of
dataType.

Parameters

in	<i>m1</i>	Indirect matrix reference
in	<i>m2</i>	Raw matrix reference

Returns

$m1 + m2$ (raw matrix)

```
template<class dataType > os::matrix<dataType> operator+ ( const os::matrix< dataType > & m1, const os::indirectMatrix< dataType > & m2 ) throw )
```

Addition.

Preforms matrix addition. Matrix addition is undefined if the two matrices are of different size. If the operation is undefined, a matrix of size (0,0) will be returned. Depends on the '+' operator of dataType.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>m2</i>	Indirect matrix reference

Returns

$m1 + m2$ (raw matrix)

```
template<class dataType > os::indirectMatrix<dataType> operator+ ( const os::indirectMatrix< dataType > & m1, const os::indirectMatrix< dataType > & m2 ) throw )
```

Addition.

Preforms matrix addition. Matrix addition is undefined if the two matrices are of different size. If the operation is undefined, a matrix of size (0,0) will be returned. Depends on the '+' operator of dataType.

Parameters

in	<i>m1</i>	Indirect matrix reference
in	<i>m2</i>	Indirect matrix reference

Returns

$m1 + m2$ (indirect matrix)

```
template<class dataType > os::matrix<dataType> operator- ( const os::matrix< dataType > & m1, const os::matrix< dataType > & m2 ) throw )
```

Subtraction.

Preforms matrix subtraction. Matrix subtraction is undefined if the two matrices are of different size. If the operation is undefined, a matrix of size (0,0) will be returned. Depends on the '-' operator of dataType.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>m2</i>	Raw matrix reference

Returns

$m1 - m2$ (raw matrix)

```
template<class dataType > os::matrix<dataType> operator- ( const os::indirectMatrix< dataType  
> & m1, const os::matrix< dataType > & m2 ) throw )
```

Subtraction.

Preforms matrix subtraction. Matrix subtraction is undefined if the two matrices are of different size. If the operation is undefined, a matrix of size (0,0) will be returned. Depends on the '-' operator of dataType.

Parameters

in	<i>m1</i>	Indirect matrix reference
in	<i>m2</i>	Raw matrix reference

Returns

$m1 - m2$ (raw matrix)

```
template<class dataType > os::matrix<dataType> operator- ( const os::matrix< dataType > &  
m1, const os::indirectMatrix< dataType > & m2 ) throw )
```

Subtraction.

Preforms matrix subtraction. Matrix subtraction is undefined if the two matrices are of different size. If the operation is undefined, a matrix of size (0,0) will be returned. Depends on the '-' operator of dataType.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>m2</i>	Indirect matrix reference

Returns

$m1 - m2$ (raw matrix)

```
template<class dataType > os::indirectMatrix<dataType> operator- ( const os::indirectMatrix<  
dataType > & m1, const os::indirectMatrix< dataType > & m2 ) throw )
```

Subtraction.

Preforms matrix subtraction. Matrix subtraction is undefined if the two matrices are of different size. If the operation is undefined, a matrix of size (0,0) will be returned. Depends on the '-' operator of dataType.

Parameters

in	<i>m1</i>	Indirect matrix reference
in	<i>m2</i>	Indirect matrix reference

Returns

$m1 - m2$ (indirect matrix)

```
template<class dataType > os::matrix<dataType> operator/ ( const os::matrix< dataType > &
m1, const dataType & d1 ) throw )
```

Scalar division.

Divides a matrix by a constant. This function depends on the '/' operator of the dataType. No zero check, as the dataType is not defined.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>d1</i>	Scalar data type

Returns

$m1/d$ (raw matrix)

```
template<class dataType > os::indirectMatrix<dataType> operator/ ( const os::indirectMatrix<
dataType > & m1, const dataType & d1 ) throw )
```

Scalar division.

Divides an indirect matrix by a constant. This function depends on the '/' operator of the dataType. No zero check, as the dataType is not defined.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>d1</i>	Scalar data type

Returns

$m1/d$ (raw matrix)

```
template<class dataType > std::ostream& operator<< ( std::ostream & os, const os::matrix<
dataType > & dt ) throw )
```

Prints out a matrix.

Prints out the entire matrix in the provided output stream. This matrix will be printed out in text form and requires the dataType of the matrix to define an ostream operator.

Parameters

	<i>[in/out]</i>	os std::ostream reference
in	<i>dt</i>	Raw matrix reference

Returns

`std::ostream os`

```
template<class dataType > std::ostream& operator<< ( std::ostream & os, const  
os::indirectMatrix< dataType > & dt ) throw )
```

Prints out a matrix.

Prints out the entire matrix in the provided output stream. This matrix will be printed out in text form and requires the dataType of the matrix to define an ostream operator.

Parameters

	<i>[in/out]</i>	os std::ostream reference
in	<i>dt</i>	Indirect matrix reference

Returns

`std::ostream os`

```
template<class dataType > bool operator== ( const os::matrix< dataType > & m1, const  
os::matrix< dataType > & m2 ) throw )
```

Test for equality.

Tests the two matrices for equal size and then tests each matrix element for equality as well. This function is dependent on the '!=' definition of the dataType.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>m2</i>	Raw matrix reference

Returns

True if exactly equivalent

```
template<class dataType > bool operator== ( const os::indirectMatrix< dataType > & m1, const  
os::matrix< dataType > & m2 ) throw )
```

Test for equality.

Tests the two matrices for equal size and then tests each matrix element for equality as well. This function is dependent on the '!=' definition of the dataType.

Parameters

in	<i>m1</i>	Indirect matrix reference
in	<i>m2</i>	Raw matrix reference

Returns

True if exactly equivalent

```
template<class dataType > bool operator==( const os::matrix< dataType > & m1, const os::indirectMatrix< dataType > & m2 ) throw )
```

Test for equality.

Tests the two matrices for equal size and then tests each matrix element for equality as well. This function is dependent on the '!=' definition of the dataType.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>m2</i>	Indirect matrix reference

Returns

True if exactly equivalent

```
template<class dataType > bool operator==( const os::indirectMatrix< dataType > & m1, const os::indirectMatrix< dataType > & m2 ) throw )
```

Test for equality.

Tests the two matrices for equal size and then tests each matrix element for equality as well. This function is dependent on the '!=' definition of the dataType.

Parameters

in	<i>m1</i>	Indirect matrix reference
in	<i>m2</i>	Indirect matrix reference

Returns

True if exactly equivalent

9.24 nodeFrame.h File Reference

Declares a framework for nodes.

Classes

- class **os::nodeFrame**< **dataType** >
nodeFrame (p. 233) interface

Namespaces

- **os**

9.24.1 Detailed Description

Declares a framework for nodes.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Defines a node interface to be used by data structures to create iterators.

9.25 nodeMacroHeader.h File Reference

Node macro header.

Classes

- class **os::NODE**< **dataType** >
Object node definition.

Namespaces

- **os**

9.25.1 Detailed Description

Node macro header.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Defines three different types of nodes to be used. One node defines a version of the object defined by an object, and the other two by a pointer to an object.

9.26 osLogger.cpp File Reference

Logging for os namespace, implementation.

9.26.1 Detailed Description

Logging for os namespace, implementation.

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

This file contains global functions and variables used for logging in the os namespace.

9.27 osLogger.h File Reference

Logging for os namespace.

Namespaces

- **os**

Functions

- `std::ostream & os::osout_func ()`
Standard out object for os namespace.
- `std::ostream & os::oserr_func ()`
Standard error object for os namespace.

Variables

- `smart_ptr< std::ostream > os::osout_ptr`
Standard out pointer for os namespace.
- `smart_ptr< std::ostream > os::oserr_ptr`
Standard error pointer for os namespace.

9.27.1 Detailed Description

Logging for os namespace.

Jonathan Bedard

Date

1/30/2016

Bug No known bugs.

This file contains declarations which are used for logging within the os namespace.

9.28 readWriteInterface.h File Reference

Classes

- class **os::readWriteInterface**
Read/write interface.

Namespaces

- **os**

9.29 readWriteLock.cpp File Reference

Implements the read/write lock.

9.29.1 Detailed Description

Implements the read/write lock.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Implements the read/write lock as defined in **readWriteLock.h** (p. 75).

9.30 readWriteLock.h File Reference

Declaration of the read/write lock.

Classes

- class **os::readWriteLock**
Read/write lock.

Namespaces

- **os**

9.30.1 Detailed Description

Declaration of the read/write lock.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This header file defines a lock which can be incremented for reading and locked for writing.

9.31 simpleHash.h File Reference

Constant locking support.

Classes

- class **os::simpleHash**< **dataType** >
Basic hash function.

Namespaces

- **os**

9.31.1 Detailed Description

Constant locking support.

Author

Jonathan Bedard

Date

7/4/2016

Bug No known bugs.

Contains a basic hash storage structure. Note that so long as an object can be cast to an integer, this hash table will work.

9.32 smartPointer.h File Reference

Template declaration of **os::smart_ptr** (p. 257).

Classes

- class **os::smart_ptr< dataType >**
Reference counted pointer.
- class **os::errorPointer**
Error pointer class.

Namespaces

- **os**

Typedefs

- typedef void(* **os::void_rec**) (void *)
Deletion function typedef.

Enumerations

- enum **os::smart_pointer_type** {
 os::null_type =0, **os::raw_type**, **os::shared_type**, **os::shared_type_array**,
 os::shared_type_dynamic_delete }
*Enumeration for types of **os::smart_ptr** (p. 257).*

Functions

- template<class targ , class src >
 smart_ptr< targ > **os::cast** (const **os::smart_ptr**< src > &conv) throw ()
 os::smart_ptr (p. 257) *cast function*
- template<class dataType >
 bool **operator==** (const **os::smart_ptr**< dataType > &c1, const **os::smart_ptr**< dataType >
 &c2) throw ()
- template<class dataType >
 bool **operator==** (const **os::smart_ptr**< dataType > &c1, const dataType *c2) throw ()
- template<class dataType >
 bool **operator==** (const dataType *c1, const **os::smart_ptr**< dataType > &c2) throw ()
- template<class dataType >
 bool **operator==** (const **os::smart_ptr**< dataType > &c1, const void *c2) throw ()
- template<class dataType >
 bool **operator==** (const void *c1, const **os::smart_ptr**< dataType > &c2) throw ()
- template<class dataType >
 bool **operator==** (const **os::smart_ptr**< dataType > &c1, const int c2) throw ()
- template<class dataType >
 bool **operator==** (const int c1, const **os::smart_ptr**< dataType > &c2) throw ()
- template<class dataType >
 bool **operator==** (const **os::smart_ptr**< dataType > &c1, const long c2) throw ()
- template<class dataType >
 bool **operator==** (const long c1, const **os::smart_ptr**< dataType > &c2) throw ()

- [illegible]

- [illegible]

- `template<class dataType >`
`bool operator>= (const os::smart_ptr< dataType > &c1, const os::smart_ptr< dataType >`
`&c2) throw ()`
- `template<class dataType >`
`bool operator>= (const os::smart_ptr< dataType > &c1, const dataType *&c2) throw ()`
- `template<class dataType >`
`bool operator>= (const dataType *&c1, const os::smart_ptr< dataType > &c2) throw ()`
- `template<class dataType >`
`bool operator>= (const os::smart_ptr< dataType > &c1, const void *&c2) throw ()`
- `template<class dataType >`
`bool operator>= (const void *&c1, const os::smart_ptr< dataType > &c2) throw ()`
- `template<class dataType >`
`bool operator>= (const os::smart_ptr< dataType > &c1, const int c2) throw ()`
- `template<class dataType >`
`bool operator>= (const int c1, const os::smart_ptr< dataType > &c2) throw ()`
- `template<class dataType >`
`bool operator>= (const os::smart_ptr< dataType > &c1, const long c2) throw ()`
- `template<class dataType >`
`bool operator>= (const long c1, const os::smart_ptr< dataType > &c2) throw ()`
- `template<class dataType >`
`bool operator>= (const os::smart_ptr< dataType > &c1, const unsigned long c2) throw ()`
- `template<class dataType >`
`bool operator>= (const unsigned long c1, const os::smart_ptr< dataType > &c2) throw ()`

9.32.1 Detailed Description

Template declaration of `os::smart_ptr` (p. 257).

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This file contains a template declaration of `os::smart_ptr` (p. 257) and supporting constants and functions. Note that because `os::smart_ptr` (p. 257) is a template class, the implementation of `os::smart_ptr` (p. 257) occurs here as well.

9.32.2 Function Documentation

```
template<class dataType > bool operator!= ( const os::smart_ptr< dataType > & c1, const
os::smart_ptr< dataType > & c2 ) throw ) [inline]
```

```
template<class dataType > bool operator!= ( const os::smart_ptr< dataType > & c1, const
dataType * c2 ) throw ) [inline]
```

```

template<class dataType > bool operator!= ( const dataType * c1, const os::smart_ptr< dataType
> & c2 ) throw ) [inline]

template<class dataType > bool operator!= ( const os::smart_ptr< dataType > & c1, const void *
c2 ) throw ) [inline]

template<class dataType > bool operator!= ( const void * c1, const os::smart_ptr< dataType > &
c2 ) throw ) [inline]

template<class dataType > bool operator!= ( const os::smart_ptr< dataType > & c1, const int c2
) throw ) [inline]

template<class dataType > bool operator!= ( const int c1, const os::smart_ptr< dataType > & c2
) throw ) [inline]

template<class dataType > bool operator!= ( const os::smart_ptr< dataType > & c1, const long
c2 ) throw ) [inline]

template<class dataType > bool operator!= ( const long c1, const os::smart_ptr< dataType > &
c2 ) throw ) [inline]

template<class dataType > bool operator!= ( const os::smart_ptr< dataType > & c1, const
unsigned long c2 ) throw ) [inline]

template<class dataType > bool operator!= ( const unsigned long c1, const os::smart_ptr<
dataType > & c2 ) throw ) [inline]

template<class dataType > bool operator< ( const os::smart_ptr< dataType > & c1, const
os::smart_ptr< dataType > & c2 ) throw ) [inline]

template<class dataType > bool operator< ( const os::smart_ptr< dataType > & c1, const
dataType * c2 ) throw ) [inline]

template<class dataType > bool operator< ( const dataType * c1, const os::smart_ptr< dataType
> & c2 ) throw ) [inline]

template<class dataType > bool operator< ( const os::smart_ptr< dataType > & c1, const void *
c2 ) throw ) [inline]

template<class dataType > bool operator< ( const void * c1, const os::smart_ptr< dataType > &
c2 ) throw ) [inline]

template<class dataType > bool operator< ( const os::smart_ptr< dataType > & c1, const int c2 )
throw ) [inline]

template<class dataType > bool operator< ( const int c1, const os::smart_ptr< dataType > & c2 )
throw ) [inline]

template<class dataType > bool operator< ( const os::smart_ptr< dataType > & c1, const long c2
) throw ) [inline]

template<class dataType > bool operator< ( const long c1, const os::smart_ptr< dataType > & c2
) throw ) [inline]

```

```

template<class dataType > bool operator< ( const os::smart_ptr< dataType > & c1, const
unsigned long c2 ) throw ) [inline]

template<class dataType > bool operator< ( const unsigned long c1, const os::smart_ptr<
dataType > & c2 ) throw ) [inline]

template<class dataType > bool operator<= ( const os::smart_ptr< dataType > & c1, const
os::smart_ptr< dataType > & c2 ) throw ) [inline]

template<class dataType > bool operator<= ( const os::smart_ptr< dataType > & c1, const
dataType * c2 ) throw ) [inline]

template<class dataType > bool operator<= ( const dataType * c1, const os::smart_ptr<
dataType > & c2 ) throw ) [inline]

template<class dataType > bool operator<= ( const os::smart_ptr< dataType > & c1, const void *
c2 ) throw ) [inline]

template<class dataType > bool operator<= ( const void * c1, const os::smart_ptr< dataType > &
c2 ) throw ) [inline]

template<class dataType > bool operator<= ( const os::smart_ptr< dataType > & c1, const int c2
) throw ) [inline]

template<class dataType > bool operator<= ( const int c1, const os::smart_ptr< dataType > & c2
) throw ) [inline]

template<class dataType > bool operator<= ( const os::smart_ptr< dataType > & c1, const long
c2 ) throw ) [inline]

template<class dataType > bool operator<= ( const long c1, const os::smart_ptr< dataType > &
c2 ) throw ) [inline]

template<class dataType > bool operator<= ( const os::smart_ptr< dataType > & c1, const
unsigned long c2 ) throw ) [inline]

template<class dataType > bool operator<= ( const unsigned long c1, const os::smart_ptr<
dataType > & c2 ) throw ) [inline]

template<class dataType > bool operator== ( const os::smart_ptr< dataType > & c1, const
os::smart_ptr< dataType > & c2 ) throw ) [inline]

template<class dataType > bool operator== ( const os::smart_ptr< dataType > & c1, const
dataType * c2 ) throw ) [inline]

template<class dataType > bool operator== ( const dataType * c1, const os::smart_ptr<
dataType > & c2 ) throw ) [inline]

template<class dataType > bool operator== ( const os::smart_ptr< dataType > & c1, const void *
c2 ) throw ) [inline]

template<class dataType > bool operator== ( const void * c1, const os::smart_ptr< dataType > &
c2 ) throw ) [inline]

```

```

template<class dataType > bool operator== ( const os::smart_ptr< dataType > & c1, const int c2
) throw ) [inline]

template<class dataType > bool operator== ( const int c1, const os::smart_ptr< dataType > & c2
) throw ) [inline]

template<class dataType > bool operator== ( const os::smart_ptr< dataType > & c1, const long
c2 ) throw ) [inline]

template<class dataType > bool operator== ( const long c1, const os::smart_ptr< dataType > &
c2 ) throw ) [inline]

template<class dataType > bool operator== ( const os::smart_ptr< dataType > & c1, const
unsigned long c2 ) throw ) [inline]

template<class dataType > bool operator== ( const unsigned long c1, const os::smart_ptr<
dataType > & c2 ) throw ) [inline]

template<class dataType > bool operator> ( const os::smart_ptr< dataType > & c1, const
os::smart_ptr< dataType > & c2 ) throw ) [inline]

template<class dataType > bool operator> ( const os::smart_ptr< dataType > & c1, const
dataType *& c2 ) throw ) [inline]

template<class dataType > bool operator> ( const dataType *& c1, const os::smart_ptr<
dataType > & c2 ) throw ) [inline]

template<class dataType > bool operator> ( const os::smart_ptr< dataType > & c1, const void *
c2 ) throw ) [inline]

template<class dataType > bool operator> ( const void * c1, const os::smart_ptr< dataType > &
c2 ) throw ) [inline]

template<class dataType > bool operator> ( const os::smart_ptr< dataType > & c1, const int c2 )
throw ) [inline]

template<class dataType > bool operator> ( const int c1, const os::smart_ptr< dataType > & c2 )
throw ) [inline]

template<class dataType > bool operator> ( const os::smart_ptr< dataType > & c1, const long c2
) throw ) [inline]

template<class dataType > bool operator> ( const long c1, const os::smart_ptr< dataType > & c2
) throw ) [inline]

template<class dataType > bool operator> ( const os::smart_ptr< dataType > & c1, const
unsigned long c2 ) throw ) [inline]

template<class dataType > bool operator> ( const unsigned long c1, const os::smart_ptr<
dataType > & c2 ) throw ) [inline]

template<class dataType > bool operator== ( const os::smart_ptr< dataType > & c1, const
os::smart_ptr< dataType > & c2 ) throw ) [inline]

```

```

template<class dataType > bool operator>= ( const os::smart_ptr< dataType > & c1, const
dataType *& c2 ) throw ) [inline]

template<class dataType > bool operator>= ( const dataType *& c1, const os::smart_ptr<
dataType > & c2 ) throw ) [inline]

template<class dataType > bool operator>= ( const os::smart_ptr< dataType > & c1, const void *
c2 ) throw ) [inline]

template<class dataType > bool operator>= ( const void * c1, const os::smart_ptr< dataType > &
c2 ) throw ) [inline]

template<class dataType > bool operator>= ( const os::smart_ptr< dataType > & c1, const int c2
) throw ) [inline]

template<class dataType > bool operator>= ( const int c1, const os::smart_ptr< dataType > & c2
) throw ) [inline]

template<class dataType > bool operator>= ( const os::smart_ptr< dataType > & c1, const long
c2 ) throw ) [inline]

template<class dataType > bool operator>= ( const long c1, const os::smart_ptr< dataType > &
c2 ) throw ) [inline]

template<class dataType > bool operator>= ( const os::smart_ptr< dataType > & c1, const
unsigned long c2 ) throw ) [inline]

template<class dataType > bool operator>= ( const unsigned long c1, const os::smart_ptr<
dataType > & c2 ) throw ) [inline]

```

9.33 specializedStructures.h File Reference

Include specialized structure headers with macro management.

9.33.1 Detailed Description

Include specialized structure headers with macro management.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Includes a series of headers extending containers with different macro declarations defined. Note that these containers extend basic ones, allowing for more specific access rules.

9.34 threadCounter.cpp File Reference

Implementation of thread counter.

9.34.1 Detailed Description

Implementation of thread counter.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Implementation of the threadCounter class defined in **threadCounter.h** (p. 85).

9.35 threadCounter.h File Reference

Thread counter declaration.

Classes

- class **os::threadCounter**
Thread counter.

Namespaces

- **os**

9.35.1 Detailed Description

Thread counter declaration.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Defines a class used to attach a counter to a thread.

9.36 threadLock.cpp File Reference

Implementation of thread locks.

9.36.1 Detailed Description

Implementation of thread locks.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Implements the recursive thread lock.

9.37 threadLock.h File Reference

Recursive lock.

Classes

- class **os::threadLock**

Thread lock Wraps the std::recursive_mutex class allowing it to be accessed in a constant way. This thread will allow a lock to be called multiple times in a single thread.

Namespaces

- **os**

9.37.1 Detailed Description

Recursive lock.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Defines a recursive lock which can be locked multiple times in the same thread.

9.38 vector2d.h File Reference

Vector templates.

Classes

- class **os::vector2d**< **dataType** >
2-dimensional vector

Namespaces

- **os**

Typedefs

- typedef vector2d< int8_t > **os::vector2d_8**
8 bit 2-d vector
- typedef vector2d< uint8_t > **os::vector2d_u8**
unsigned 8 bit 2-d vector
- typedef vector2d< int16_t > **os::vector2d_16**
16 bit 2-d vector
- typedef vector2d< uint16_t > **os::vector2d_u16**
unsigned 16 bit 2-d vector
- typedef vector2d< int32_t > **os::vector2d_32**
32 bit 2-d vector
- typedef vector2d< uint32_t > **os::vector2d_u32**
unsigned 32 bit 2-d vector
- typedef vector2d< int64_t > **os::vector2d_64**
64 bit 2-d vector
- typedef vector2d< uint64_t > **os::vector2d_u64**
unsigned 64 bit 2-d vector
- typedef vector2d< float > **os::vector2d_f**
float 2-d vector
- typedef vector2d< double > **os::vector2d_d**
double 2-d vector

9.38.1 Detailed Description

Vector templates.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This file contains a template classes defining a 2-d vector object. Vectors can, in a broad sense, be used for any class which defines general mathematical operations. This particular file offers vector type definitions for all of the basic integer and floating point types.

9.39 vector3d.h File Reference

Vector templates.

Classes

- class **os::vector3d**< **dataType** >
3-dimensional vector

Namespaces

- **os**

Typedefs

- typedef vector3d< int8_t > **os::vector3d_8**
8 bit 3-d vector
- typedef vector3d< uint8_t > **os::vector3d_u8**
unsigned 8 bit 3-d vector
- typedef vector3d< int16_t > **os::vector3d_16**
16 bit 3-d vector
- typedef vector3d< uint16_t > **os::vector3d_u16**
unsigned 16 bit 3-d vector
- typedef vector3d< int32_t > **os::vector3d_32**
32 bit 3-d vector
- typedef vector3d< uint32_t > **os::vector3d_u32**
unsigned 32 bit 3-d vector
- typedef vector3d< int64_t > **os::vector3d_64**
64 bit 3-d vector
- typedef vector3d< uint64_t > **os::vector3d_u64**
unsigned 64 bit 3-d vector
- typedef vector3d< float > **os::vector3d_f**
float 3-d vector
- typedef vector3d< double > **os::vector3d_d**
double 3-d vector

9.39.1 Detailed Description

Vector templates.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This file contains a template classes defining a 3-d vector object. Vectors can, in a broad sense, be used for any class which defines general mathematical operations. This particular file offers vector type definitions for all of the basic integer and floating point types.

Chapter 10

Class Index

10.1 Class List

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

os::adnode< dataType >	Abstract data-node	??
os::ads< dataType >	Abstract datastructure	??
os::ARRAY_STACK< dataType >	107
os::asyncAVLNode< dataType >	Node for usage in an asynchronous AVL tree	??
os::asyncAVLTree< dataType >	Asynchronous balanced binary search tree	??
os::AVL_NODE< dataType >	AVL tree node Node used in an AVL tree. This node allows both iteration and random access	109
os::AVL_TREE< dataType >	Template AVL Tree definition	118
os::AVLNode< dataType >	Node for usage in an AVL tree	??
os::AVLTree< dataType >	Balanced binary search tree	??
os::basicLock	Basic lock Wraps the std::mutex class allowing it to be accessed in a constant way	126
os::BOUNDED_QUEUE< dataType >	Template BOUNDED_QUEUE (p. 129) definition	129
os::BOUNDED_QUEUE_NODE< dataType >	BOUNDED_QUEUE (p. 129) node	138
os::constantPrinter	Prints constant arrays to files	143
os::constIterator< dataType >	Generalized constant iterator	147
os::DATASTRUCTURE< dataType >	Object node definition	155

os::descriptiveException	Basic exception with description	161
os::errorPointer	Error pointer class	163
os::eventReceiver< senderType >	Class which enables event receiving	165
os::eventSender< receiverType >	Class which enables event sending	167
os::HASH< dataType >	Iterable hash table	171
os::HASH_NODE< dataType >	Hash node	177
os::indirectMatrix< dataType >	Indirect matrix	181
os::iterator< dataType >	Generalized iterator	187
os::iteratorBase< dataType >	Iterator source	196
os::iteratorBaseInterface< dataType >	Iterator base interface	200
os::iteratorSource	Base class for iterator source	203
os::LINKED_QUEUE< dataType >	Queue built on-top of a list	206
os::LINKED_STACK< dataType >	Stack built on-top of a list	209
os::LIST< dataType >	List framework Template for a linked list, insertion is fully defined in subsequent extensions of this class	211
os::LIST_NODE< dataType >	List node Node used in linked lists to store each element	217
os::lockable	An interface defining a class which can be locked	221
os::matrix< dataType >	Raw matrix	224
os::NODE< dataType >	Object node definition	230
os::nodeFrame< dataType >	NodeFrame interface	233
os::ptrComp	Pointer compare interface	??
os::readWriteInterface	Read/write interface	241
os::readWriteLock	Read/write lock	244
os::simpleHash< dataType >	Basic hash function	250
os::smart_ptr< dataType >	Reference counted pointer	257

os::smartSet< dataType >	
Smart set abstract data-structures	??
os::SORTED_LIST< dataType >	
Sorted list A basic list which remains unsorted unless the sort function is called on it	269
os::threadCounter	
Thread counter	270
os::threadLock	
Thread lock Wraps the std::recursive_mutex class allowing it to be accessed in a constant way. This thread will allow a lock to be called multiple times in a single thread	273
os::UNSORTED_LIST< dataType >	
Unsorted list A basic list which remains unsorted unless the sort function is called on it	276
os::unsortedList< dataType >	
Unsorted linked list	??
os::unsortedListNode< dataType >	
Node for usage in a linked list	??
os::VECTOR< dataType >	
Template vector definition	277
os::vector2d< dataType >	
2-dimensional vector	284
os::vector3d< dataType >	
3-dimensional vector	295
os::VECTOR_NODE< dataType >	
Vector node	307

Chapter 11

Namespace Documentation

11.1 os Namespace Reference

Classes

- class **adnode**
Abstract data-node.
- class **ads**
Abstract datastructure.
- class **ARRAY_STACK**
- class **asyncAVLNode**
Node for usage in an asynchronous AVL tree.
- class **asyncAVLTree**
Asynchronous balanced binary search tree.
- class **AVL_NODE**
AVL tree node Node used in an AVL tree. This node allows both iteration and random access.
- class **AVL_TREE**
Template AVL Tree definition.
- class **AVLNode**
Node for usage in an AVL tree.
- class **AVLTree**
Balanced binary search tree.
- class **basicLock**
Basic lock Wraps the std::mutex class allowing it to be accessed in a constant way.
- class **BOUNDED_QUEUE**
*Template **BOUNDED_QUEUE** (p. 129) definition.*
- class **BOUNDED_QUEUE_NODE**
***BOUNDED_QUEUE** (p. 129) node.*
- class **constantPrinter**
Prints constant arrays to files.
- class **constIterator**

- Generalized constant iterator.*
- class **DATASTRUCTURE**
 - Object node definition.*
- class **descriptiveException**
 - Basic exception with description.*
- class **errorPointer**
 - Error pointer class.*
- class **eventReceiver**
 - Class which enables event receiving.*
- class **eventSender**
 - Class which enables event sending.*
- class **HASH**
 - Iterable hash table.*
- class **HASH_NODE**
 - Hash node.*
- class **indirectMatrix**
 - Indirect matrix.*
- class **iterator**
 - Generalized iterator.*
- class **iteratorBase**
 - Iterator source.*
- class **iteratorBaseInterface**
 - Iterator base interface.*
- class **iteratorSource**
 - Base class for iterator source.*
- class **LINKED_QUEUE**
 - Queue built on-top of a list.*
- class **LINKED_STACK**
 - Stack built on-top of a list.*
- class **LIST**
 - List framework Template for a linked list, insertion is fully defined in subsequent extensions of this class.*
- class **LIST_NODE**
 - List node Node used in linked lists to store each element.*
- class **lockable**
 - An interface defining a class which can be locked.*
- class **matrix**
 - Raw matrix.*
- class **NODE**
 - Object node definition.*
- class **nodeFrame**
 - nodeFrame** (p. 233) interface*
- class **ptrComp**

- Pointer compare interface.*
- class **readWriteInterface**
 - Read/write interface.*
- class **readWriteLock**
 - Read/write lock.*
- class **simpleHash**
 - Basic hash function.*
- class **smart_ptr**
 - Reference counted pointer.*
- class **smartSet**
 - Smart set abstract data-structures.*
- class **SORTED_LIST**
 - Sorted list A basic list which remains unsorted unless the sort function is called on it.*
- class **threadCounter**
 - Thread counter.*
- class **threadLock**
 - Thread lock Wraps the std::recursive_mutex class allowing it to be accessed in a constant way. This thread will allow a lock to be called multiple times in a single thread.*
- class **UNSORTED_LIST**
 - Unsorted list A basic list which remains unsorted unless the sort function is called on it.*
- class **unsortedList**
 - Unsorted linked list.*
- class **unsortedListNode**
 - Node for usage in a linked list.*
- class **VECTOR**
 - Template vector definition.*
- class **vector2d**
 - 2-dimensional vector*
- class **vector3d**
 - 3-dimensional vector*
- class **VECTOR_NODE**
 - Vector node.*

Typedefs

- typedef void(* **void_rec**) (void *)
 - Deletion function typedef.*
- typedef **vector2d**< int8_t > **vector2d_8**
 - 8 bit 2-d vector*
- typedef **vector2d**< uint8_t > **vector2d_u8**
 - unsigned 8 bit 2-d vector*
- typedef **vector2d**< int16_t > **vector2d_16**
 - 16 bit 2-d vector*

- typedef **vector2d**< uint16_t > **vector2d_u16**
unsigned 16 bit 2-d vector
- typedef **vector2d**< int32_t > **vector2d_32**
32 bit 2-d vector
- typedef **vector2d**< uint32_t > **vector2d_u32**
unsigned 32 bit 2-d vector
- typedef **vector2d**< int64_t > **vector2d_64**
64 bit 2-d vector
- typedef **vector2d**< uint64_t > **vector2d_u64**
unsigned 64 bit 2-d vector
- typedef **vector2d**< float > **vector2d_f**
float 2-d vector
- typedef **vector2d**< double > **vector2d_d**
double 2-d vector
- typedef **vector3d**< int8_t > **vector3d_8**
8 bit 3-d vector
- typedef **vector3d**< uint8_t > **vector3d_u8**
unsigned 8 bit 3-d vector
- typedef **vector3d**< int16_t > **vector3d_16**
16 bit 3-d vector
- typedef **vector3d**< uint16_t > **vector3d_u16**
unsigned 16 bit 3-d vector
- typedef **vector3d**< int32_t > **vector3d_32**
32 bit 3-d vector
- typedef **vector3d**< uint32_t > **vector3d_u32**
unsigned 32 bit 3-d vector
- typedef **vector3d**< int64_t > **vector3d_64**
64 bit 3-d vector
- typedef **vector3d**< uint64_t > **vector3d_u64**
unsigned 64 bit 3-d vector
- typedef **vector3d**< float > **vector3d_f**
float 3-d vector
- typedef **vector3d**< double > **vector3d_d**
double 3-d vector

Enumerations

- enum **setTypes** { **def_set** =0, **small_set**, **sorted_set** }
Index of abstract data-structures.
- enum **smart_pointer_type** {
 null_type =0, **raw_type**, **shared_type**, **shared_type_array**,
 shared_type_dynamic_delete }
*Enumeration for types of **os::smart_ptr** (p. 257).*

Functions

- `template<class dataType >`
`int defaultCompare (const dataType &v1, const dataType &v2)`
Basic compare.
- `template<class dataType >`
`int pointerCompare (const smart_ptr< dataType > &ptr1, const smart_ptr< dataType > &ptr2) throw ()`
Pointer compare.
- `template<class dataType >`
`void quicksort (dataType *arr, unsigned length, int(*sort_comparision)(const dataType &, const dataType &)=&defaultCompare)`
Template quick-sort.
- `template<class dataType >`
`void pointerQuicksort (smart_ptr< smart_ptr< dataType > > arr, unsigned length, int(*sort_comparision)(const smart_ptr< dataType > &, const smart_ptr< dataType > &)=&pointerCompare)`
Template for quick-sort, pointer version.
- `template<class dataType >`
`bool compareSize (const matrix< dataType > &m1, const matrix< dataType > &m2) throw ()`
Compares the size of two matrices.
- `template<class dataType >`
`bool compareSize (const indirectMatrix< dataType > &m1, const matrix< dataType > &m2) throw ()`
Compares the size of two matrices.
- `template<class dataType >`
`bool compareSize (const matrix< dataType > &m1, const indirectMatrix< dataType > &m2) throw ()`
Compares the size of two matrices.
- `template<class dataType >`
`bool compareSize (const indirectMatrix< dataType > &m1, const indirectMatrix< dataType > &m2) throw ()`
Compares the size of two matrices.
- `template<class dataType >`
`bool testCross (const matrix< dataType > &m1, const matrix< dataType > &m2) throw ()`
Tests if the cross-product is a legal operation.
- `template<class dataType >`
`bool testCross (const indirectMatrix< dataType > &m1, const matrix< dataType > &m2) throw ()`
Tests if the cross-product is a legal operation.
- `template<class dataType >`
`bool testCross (const matrix< dataType > &m1, const indirectMatrix< dataType > &m2) throw ()`
Tests if the cross-product is a legal operation.
- `template<class dataType >`
`bool testCross (const indirectMatrix< dataType > &m1, const indirectMatrix< dataType > &m2) throw ()`

Tests if the cross-product is a legal operation.

- `std::ostream & osout_func ()`

Standard out object for os namespace.

- `std::ostream & oserr_func ()`

Standard error object for os namespace.

- `template<class targ , class src >`

`smart_ptr< targ > cast (const os::smart_ptr< src > &conv) throw ()`

***os::smart_ptr** (p. 257) cast function*

Variables

- `smart_ptr< std::ostream > osout_ptr`

Standard out pointer for os namespace.

- `smart_ptr< std::ostream > oserr_ptr`

Standard error pointer for os namespace.

11.1.1 Typedef Documentation

typedef `vector2d<int16_t> os::vector2d_16`

16 bit 2-d vector

typedef `vector2d<int32_t> os::vector2d_32`

32 bit 2-d vector

typedef `vector2d<int64_t> os::vector2d_64`

64 bit 2-d vector

typedef `vector2d<int8_t> os::vector2d_8`

8 bit 2-d vector

typedef `vector2d<double> os::vector2d_d`

double 2-d vector

typedef `vector2d<float> os::vector2d_f`

float 2-d vector

typedef `vector2d<uint16_t> os::vector2d_u16`

unsigned 16 bit 2-d vector

typedef **vector2d**<uint32_t> **os::vector2d_u32**

unsigned 32 bit 2-d vector

typedef **vector2d**<uint64_t> **os::vector2d_u64**

unsigned 64 bit 2-d vector

typedef **vector2d**<uint8_t> **os::vector2d_u8**

unsigned 8 bit 2-d vector

typedef **vector3d**<int16_t> **os::vector3d_16**

16 bit 3-d vector

typedef **vector3d**<int32_t> **os::vector3d_32**

32 bit 3-d vector

typedef **vector3d**<int64_t> **os::vector3d_64**

64 bit 3-d vector

typedef **vector3d**<int8_t> **os::vector3d_8**

8 bit 3-d vector

typedef **vector3d**<double> **os::vector3d_d**

double 3-d vector

typedef **vector3d**<float> **os::vector3d_f**

float 3-d vector

typedef **vector3d**<uint16_t> **os::vector3d_u16**

unsigned 16 bit 3-d vector

typedef **vector3d**<uint32_t> **os::vector3d_u32**

unsigned 32 bit 3-d vector

typedef **vector3d**<uint64_t> **os::vector3d_u64**

unsigned 64 bit 3-d vector

typedef **vector3d**<uint8_t> **os::vector3d_u8**

unsigned 8 bit 3-d vector

typedef void(* os::void_rec) (void *)

Deletion function typedef.

The **os::void_rec** (p. 100) function pointer typedef is used by **os::smart_ptr** (p. 257) when it is of type **os::shared_type_dynamic_delete** (p. 101) to destroy non-standard pointers, usually when interfacing with C code.

Parameters

in,out	void*	designed for non-standard deletion.
--------	-------	-------------------------------------

Returns

void

11.1.2 Enumeration Type Documentation

enum **os::setTypes**

Index of abstract data-structures.

This enumeration contains a numbered reference to all of the available abstract data-structures.

Enumerator

def_set Default set enumeration. Currently defaults to a small set.

small_set Small memory burden set. The small set uses an unsorted linked list to store data.

sorted_set Sorted set. The sorted set uses an AVL tree to store data.

enum **os::smart_pointer_type**

Enumeration for types of **os::smart_ptr** (p. 257).

Defines types of **os::smart_ptr** (p. 257). These types are used to define the deletion behaviour of the pointer.

Enumerator

null_type No type. **os::null_type** (p. 100) pointers are the default type of **os::smart_ptr** (p. 257). Any **os::smart_ptr** (p. 257) of type **os::null_type** (p. 100) can be guaranteed to hold a NULL pointer.

raw_type Raw pointer. **os::raw_type** (p. 100) pointers are the default type of **os::smart_ptr** (p. 257) when instantiated with a standard pointer. Any **os::smart_ptr** (p. 257) of type **os::raw_type** (p. 100) is not responsible for the deletion of it's pointer and makes no guarantees as to the availability of it's pointer.

shared_type Reference counted pointer. **os::shared_type** (p. 100) pointers must be instantiated from an **os::smart_ptr** (p. 257) of this type or explicitly through **os::smart_ptr** (p. 257) constructor arguments. **os::shared_type** (p. 100) pointers will automatically delete the pointer contained within the object when the reference count of the **os::smart_ptr** (p. 257) reaches 0.

shared_type_array Reference counted array. Similar in usage and instantiation to **os::raw_type** (p. 100). **os::smart_ptr** (p. 257) of type **os::shared_type_array** (p. 101) are designed to be used with array and will run `delete []` when the reference count reaches 0 instead of `delete`.

shared_type_dynamic_delete Reference pointer with non-standard deletion. Similar in usage and instantiation to **os::raw_type** (p. 100). **os::smart_ptr** (p. 257) of type **os::shared_type_dynamic_delete** (p. 101) are used when the deletion of a pointer is not contained within the object destructor. This is specifically designed for interface with C code not using "new" and "delete."

11.1.3 Function Documentation

```
template<class targ , class src > smart_ptr<targ> os::cast ( const os::smart_ptr< src > & conv )
throw ) [inline]
```

os::smart_ptr (p. 257) cast function

Casts an **os::smart_ptr**<src> to and **os::smart_ptr**<targ>. This function is a template function, targ and src are the templates respectively. Note that the is an explicit cast and is not guranteed to be safe.

Parameters

in	conv	Reference to os::smart_ptr <src> to be converted
----	------	---

Returns

New **os::smart_ptr**<targ> constructed from the received **os::smart_ptr** (p. 257)

```
template<class dataType > bool os::compareSize ( const matrix< dataType > & m1, const
matrix< dataType > & m2 ) throw )
```

Compares the size of two matrices.

Compares the size of two raw matrices. If both have the same `_width` and the same `_height`, they are considered to be the same size.

Parameters

in	m1	Raw matrix reference
in	m2	Raw matrix reference

Returns

True if the matrices are the same size

```
template<class dataType > bool os::compareSize ( const indirectMatrix< dataType > & m1, const
matrix< dataType > & m2 ) throw )
```

Compares the size of two matrices.

Compares the size of an indirect matrix and a raw matrix in that order. If both have the same `_width` and the same `_height`, they are considered to be the same size.

Parameters

<code>in</code>	<code>m1</code>	Indirect matrix reference
<code>in</code>	<code>m2</code>	Raw matrix reference

Returns

True if the matrices are the same size

```
template<class dataType > bool os::compareSize ( const matrix< dataType > & m1, const indirectMatrix< dataType > & m2 ) throw )
```

Compares the size of two matrices.

Compares the size of a raw matrix and an indirect matrix in that order. If both have the same `_width` and the same `_height`, they are considered to be the same size.

Parameters

<code>in</code>	<code>m1</code>	Raw matrix reference
<code>in</code>	<code>m2</code>	Indirect matrix reference

Returns

True if the matrices are the same size

```
template<class dataType > bool os::compareSize ( const indirectMatrix< dataType > & m1, const indirectMatrix< dataType > & m2 ) throw )
```

Compares the size of two matrices.

Compares the size of two indirect matrices. If both have the same `_width` and the same `_height`, they are considered to be the same size.

Parameters

<code>in</code>	<code>m1</code>	Indirect matrix reference
<code>in</code>	<code>m2</code>	Indirect matrix reference

Returns

True if the matrices are the same size

```
template<class dataType > int os::defaultCompare ( const dataType & v1, const dataType & v2 )
```

Basic compare.

Acts as a default comparison function for sorting. This function compares the data as if it is in integer form.

Parameters

in	v1	Reference 1 to compare
in	v2	Reference 2 to compare

Returns

1 if greater than, -1 if less than, 0 if equal to

```
std::ostream& os::oserr_func ( )
```

Standard error object for os namespace.

#define statements allow the user to call this function with "os::oserr." Logging is achieved by using "os::oserr" as one would use "std::cerr."

```
std::ostream& os::osout_func ( )
```

Standard out object for os namespace.

#define statements allow the user to call this function with "os::osout." Logging is achieved by using "os::osout" as one would use "std::cout."

```
template<class dataType > int os::pointerCompare ( const smart_ptr< dataType > & ptr1, const smart_ptr< dataType > & ptr2 ) throw ( )
```

Pointer compare.

Acts as a default comparison function for pointer sorting. Compares the raw pointer values of the two arguments and returns the result.

Parameters

in	ptr1	Pointer 1 to compare
in	ptr2	Pointer 2 to compare

Returns

1 if greater than, -1 if less than, 0 if equal to

```
template<class dataType > void os::pointerQuicksort ( smart_ptr< smart_ptr< dataType > > arr, unsigned length, int(*) (const smart_ptr< dataType > &, const smart_ptr< dataType > &) sort_comparison = &pointerCompare )
```

Template for quick-sort, pointer version.

Performs quick sort on the provided array of the given length where the array is of pointers to the data type instead of the data type.

Parameters

	<i>[in/out]</i>	array Set of data to be sorted
in	<i>length</i>	Length of array to be sorted
in	<i>sort_comparison</i>	Comparison function definition

Returns

void

```
template<class dataType > void os::quicksort ( dataType * arr, unsigned length, int(*)(const
dataType &, const dataType &) sort_comparison = &defaultCompare )
```

Template quick-sort.

Performs quick sort on the provided array of the given length with the given comparison function. The default comparison function is one which uses the comparison operators

Parameters

	<i>[in/out]</i>	array Set of data to be sorted
in	<i>length</i>	Length of array to be sorted
in	<i>sort_comparison</i>	Comparison function definition

Returns

void

```
template<class dataType > bool os::testCross ( const matrix< dataType > & m1, const matrix<
dataType > & m2 ) throw )
```

Tests if the cross-product is a legal operation.

Compares the `_width` of the first matrix versus the `_height` of the second. If the two are equal, the cross-product is defined.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>m2</i>	Raw matrix reference

Returns

True if the cross-product is defined

```
template<class dataType > bool os::testCross ( const indirectMatrix< dataType > & m1, const
matrix< dataType > & m2 ) throw )
```

Tests if the cross-product is a legal operation.

Compares the `_width` of the first matrix versus the `_height` of the second. If the two are equal, the cross-product is defined.

Parameters

in	<i>m1</i>	Indirect matrix reference
in	<i>m2</i>	Raw matrix reference

Returns

True if the cross-product is defined

```
template<class dataType > bool os::testCross ( const matrix< dataType > & m1, const indirectMatrix< dataType > & m2 ) throw )
```

Tests if the cross-product is a legal operation.

Compares the `_width` of the first matrix versus the `_height` of the second. If the two are equal, the cross-product is defined.

Parameters

in	<i>m1</i>	Raw matrix reference
in	<i>m2</i>	Indirect matrix reference

Returns

True if the cross-product is defined

```
template<class dataType > bool os::testCross ( const indirectMatrix< dataType > & m1, const indirectMatrix< dataType > & m2 ) throw )
```

Tests if the cross-product is a legal operation.

Compares the `_width` of the first matrix versus the `_height` of the second. If the two are equal, the cross-product is defined.

Parameters

in	<i>m1</i>	Indirect matrix reference
in	<i>m2</i>	Indirect matrix reference

Returns

True if the cross-product is defined

11.1.4 Variable Documentation

smart_ptr<std::ostream> os::oserr_ptr

Standard error pointer for os namespace.

This `std::ostream` is used as standard error for the `os` namespace. This pointer can be swapped out to programmatically redirect standard error for the `os` namespace.

smart_ptr<std::ostream> `os::osout_ptr`

Standard out pointer for `os` namespace.

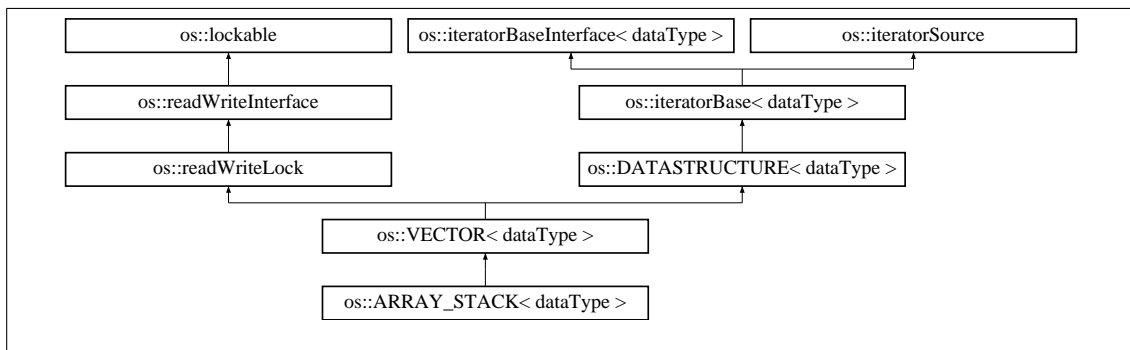
This `std::ostream` is used as standard out for the `os` namespace. This pointer can be swapped out to programmatically redirect standard out for the `os` namespace.

Chapter 12

Class Documentation

12.1 os::ARRAY_STACK< dataType > Class Template Reference

Inheritance diagram for os::ARRAY_STACK< dataType >:



Public Member Functions

- **ARRAY_STACK ()**
Default constructor Simple constructor which implicitly call the constructor of it's base.
- **virtual ~ARRAY_STACK ()**
Destroys the stack Uses the destructor function defined by the unsorted list.
- **void pop ()**
Remove top element.
- **bool push (const dataType &x)**
Add element.
- **const dataType & constTop () const**
Const access top element.
- **dataType & top ()**
Access top element.
- **const dataType & top () const**

Const access top element.

- **bool** **push** (**smart_ptr**< dataType > x)
- **const smart_ptr**< dataType > **constTop** () **const**
- **smart_ptr**< dataType > **top** ()
- **const smart_ptr**< dataType > **top** () **const**

Additional Inherited Members

12.1.1 Constructor & Destructor Documentation

```
template<class dataType > os::ARRAY_STACK< dataType >::ARRAY_STACK ( ) [inline]
```

Default constructor Simple constructor which implicitly call the constructor of it's base.

```
template<class dataType > virtual os::ARRAY_STACK< dataType >::~ARRAY_STACK ( )  
[inline], [virtual]
```

Destroys the stack Uses the destructor function defined by the unsorted list.

12.1.2 Member Function Documentation

```
template<class dataType > const dataType& os::ARRAY_STACK< dataType >::constTop ( )  
const [inline]
```

Const access top element.

Returns the top element of the stack. Note that this operation will throw an exception if the stack is empty.

Returns

Immutable top element

```
template<class dataType > const smart_ptr<dataType> os::ARRAY_STACK< dataType  
>::constTop ( ) const [inline]
```

```
template<class dataType > void os::ARRAY_STACK< dataType >::pop ( ) [inline]
```

Remove top element.

Attempts to remove the top element. Throws an exception if the stack is empty.

Returns

void

```
template<class dataType > bool os::ARRAY_STACK< dataType >::push ( const dataType & x )  
[inline]
```

Add element.

Inserts an element at the top of the stack

Parameters

in	x	Element to be inserted
----	---	------------------------

Returns

true

```
template<class dataType > bool os::ARRAY_STACK< dataType >::push ( smart_ptr< dataType > x ) [inline]
```

```
template<class dataType > dataType& os::ARRAY_STACK< dataType >::top ( ) [inline]
```

Access top element.

Returns the top element of the stack. Note that this operation will throw an exception if the stack is empty.

Returns

Mutable top element

```
template<class dataType > const dataType& os::ARRAY_STACK< dataType >::top ( ) const [inline]
```

Const access top element.

Returns the top element of the stack. Note that this operation will throw an exception if the stack is empty.

Returns

Immutable top element

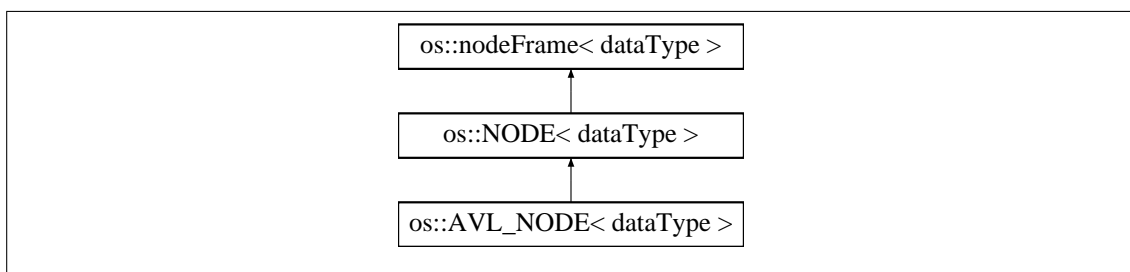
```
template<class dataType > smart_ptr<dataType> os::ARRAY_STACK< dataType >::top ( ) [inline]
```

```
template<class dataType > const smart_ptr<dataType> os::ARRAY_STACK< dataType >::top ( ) const [inline]
```

12.2 os::AVL_NODE< dataType > Class Template Reference

AVL tree node Node used in an AVL tree. This node allows both iteration and random access.

Inheritance diagram for os::AVL_NODE< dataType >:



Public Member Functions

- **smart_ptr< AVL_NODE< dataType > > parent ()**
Returns the mutable parent node.
- **const smart_ptr< AVL_NODE< dataType > > parent () const**
Returns an immutable parent node.
- **smart_ptr< AVL_NODE< dataType > > child (int x)**
Returns a mutable child by index.
- **const smart_ptr< AVL_NODE< dataType > > child (int x) const**
Returns an immutable child by index.
- **size_t height () const**
Returns the height of the sub-tree.
- **size_t numChildren () const**
Returns the number of children.
- **void remove () final throw (descriptiveException)**
Remove this node from the tree.
- **bool iterable () const final**
Returns if the node is iterable.
- **bool randomAccess () const final**
Returns if the node can be accessed randomly.
- **smart_ptr< nodeFrame< dataType > > getNext () final throw (descriptiveException)**
Find the next node.
- **smart_ptr< nodeFrame< dataType > > getPrev () final throw (descriptiveException)**
Find the previous node.
- **const smart_ptr< nodeFrame< dataType > > getNextConst () const final throw (descriptiveException)**
Find the next node.
- **const smart_ptr< nodeFrame< dataType > > getPrevConst () const final throw (descriptiveException)**
Find the previous node.
- **smart_ptr< nodeFrame< dataType > > access (long offset) final throw (descriptiveException)**
Access node by index.
- **const smart_ptr< nodeFrame< dataType > > constAccess (long offset) const final throw (descriptiveException)**
Access node by index.

Static Public Attributes

- **static const bool ITERABLE = true**
AVL nodes are iterable.
- **static const bool RANDOM_ACCESS = true**
AVL nodes allow random-access.

Private Member Functions

- void **propagateUp** (int isFirst=0)
Processes both height and children count This function is designed to move recursively up the tree.
- void **setChild** (**smart_ptr**< **AVL_NODE**< dataType > > c)
Add a child to this node.
- void **setParent** (**smart_ptr**< **AVL_NODE**< dataType > > p, **smart_ptr**< **AVL_NODE**< dataType > > self_pointer)
Sets the parent node.
- void **removeChild** (**smart_ptr**< **AVL_NODE**< dataType > > c)
Remove a child from this node.
- void **removeChild** (int pos)
Remove a child from this node.
- void **removeParent** ()
Remove the parent node.
- void **purge** ()
Remove all children and parents.
- **AVL_NODE** (**AVL_TREE**< dataType > *src, const dataType &dt)
Private constructor.

Private Attributes

- **smart_ptr**< **AVL_NODE**< dataType > > **_parent**
Parent node one level up in the tree.
- **smart_ptr**< **AVL_NODE**< dataType > > **_child** [2]
Children nodes.
- size_t **_height**
The height of the tree.
- size_t **_numChildren**
Number of children.

Additional Inherited Members

12.2.1 Detailed Description

```
template<class dataType>
class os::AVL_NODE< dataType >
```

AVL tree node Node used in an AVL tree. This node allows both iteration and random access.

12.2.2 Constructor & Destructor Documentation

```
template<class dataType > os::AVL_NODE< dataType >::AVL_NODE ( AVL_TREE< dataType >
* src, const dataType & dt ) [inline], [private]
```

Private constructor.

Only AVL trees can access the node constructor.

Parameters

in	src	Source structure
in	dt	Data to be bound to node

12.2.3 Member Function Documentation

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::AVL_NODE< dataType  
>::access ( long offset ) throw descriptiveException  [inline], [final], [virtual]
```

Access node by index.

Access a node offset from the current node by some value. If a node cannot be randomly accessed, an exception will be thrown.

Returns

Offset node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 236).

```
template<class dataType > smart_ptr<AVL_NODE<dataType> > os::AVL_NODE< dataType  
>::child ( int x ) [inline]
```

Returns a mutable child by index.

Returns child node by index. 0 indicates the left child, **AVL_NODE**<dataType>::child1. 1 indicates the right child, **AVL_NODE**<dataType>::child2. All other indices will return NULL.

Returns

Modifiable **os::AVL_NODE**<dataType>::child1 for x==0, **AVL_NODE**<dataType>::child2 for x==1

```
template<class dataType > const smart_ptr<AVL_NODE<dataType> > os::AVL_NODE<  
dataType >::child ( int x ) const [inline]
```

Returns an immutable child by index.

Returns child node by index. 0 indicates the left child, **AVL_NODE**<dataType>::child1. 1 indicates the right child, **AVL_NODE**<dataType>::child2. All other indices will return NULL.

Returns

Constant **os::AVL_NODE**<dataType>::child1 for x==0, **AVL_NODE**<dataType>::child2 for x==1

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::AVL_NODE<  
dataType >::constAccess ( long offset ) const throw descriptiveException  [inline], [final],  
[virtual]
```

Access node by index.

Access a node offset from the current node by some value. If a node cannot be randomly accessed, an exception will be thrown.

Returns

Offset node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 236).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::AVL_NODE< dataType  
>::getNext ( ) throw descriptiveException) [inline], [final], [virtual]
```

Find the next node.

This functions attempts to search for the next node in the structure. Note that this will prevent writing to the structure while it is in progress.

Returns

Pointer to the next node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 237).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::AVL_NODE<  
dataType >::getNextConst ( ) const throw descriptiveException) [inline], [final],  
[virtual]
```

Find the next node.

This functions attempts to search for the next node in the structure. Note that this will prevent writing to the structure while it is in progress.

Returns

Pointer to the next node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 238).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::AVL_NODE< dataType  
>::getPrev ( ) throw descriptiveException) [inline], [final], [virtual]
```

Find the previous node.

This functions attempts to search for the previous node in the structure. Note that this will prevent writing to the structure while it is in progress.

Returns

Pointer to the previous node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 238).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::AVL_NODE<  
dataType >::getPrevConst ( ) const throw descriptiveException) [inline], [final],  
[virtual]
```

Find the previous node.

This functions attempts to search for the previous node in the structure. Note that this will prevent writing to the structure while it is in progress.

Returns

Pointer to the previous node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 238).

template<class dataType > size_t **os::AVL_NODE**< dataType >::height () const [inline]

Returns the height of the sub-tree.

Returns

os::AVL_NODE<dataType>::_height (p. 117)

template<class dataType > bool **os::AVL_NODE**< dataType >::iterable () const [inline],
[final], [virtual]

Returns if the node is iterable.

Returns

object::ITERABLE

Reimplemented from **os::nodeFrame**< dataType > (p. 239).

template<class dataType > size_t **os::AVL_NODE**< dataType >::numChildren () const
[inline]

Returns the number of children.

Returns

os::AVL_NODE<dataType>::_numChildren (p. 117)

template<class dataType > smart_ptr<AVL_NODE<dataType> > **os::AVL_NODE**< dataType
>::parent () [inline]

Returns the mutable parent node.

Returns

Modifiable **os::AVL_NODE**<dataType>::_parent (p. 117)

template<class dataType > const smart_ptr<AVL_NODE<dataType> > **os::AVL_NODE**<
dataType >::parent () const [inline]

Returns an immutable parent node.

Returns

Constant **os::AVL_NODE**<dataType>::_parent (p. 117)

template<class dataType > void **os::AVL_NODE**< dataType >::propagateUp (int isFirst = 0)
[inline], [private]

Processes both height and children count This function is designed to move recursively up the tree.

Parameters

in	<i>isFirst</i>	0 by default, increments each time
----	----------------	------------------------------------

Returns

void

```
template<class dataType > void os::AVL_NODE< dataType >::purge ( ) [inline], [private]
```

Remove all children and parents.

This function is important because nodes are of type **os::smart_ptr** (p.257), since there are co-dependencies, failure to run this function on deletion of the tree will cause a memory leak.

Returns

void

```
template<class dataType > bool os::AVL_NODE< dataType >::randomAccess ( ) const  
[inline], [final], [virtual]
```

Returns if the node can be accessed randomly.

Returns

object::RANDOM_ACCESS

Reimplemented from **os::nodeFrame**< **dataType** > (p. 240).

```
template<class dataType > void os::AVL_NODE< dataType >::remove ( ) throw  
descriptiveException) [final], [virtual]
```

Remove this node from the tree.

Returns

void

Reimplemented from **os::nodeFrame**< **dataType** > (p. 240).

```
template<class dataType > void os::AVL_NODE< dataType >::removeChild ( smart_ptr<  
AVL_NODE< dataType > > c ) [inline], [private]
```

Remove a child from this node.

Checks **os::AVL_NODE**<dataType>::child1 and **os::AVL_NODE**<dataType>::child2 for equality with the the node received as a parameter.

Parameters

in	c	Node to be removed
----	---	--------------------

Returns

void

```
template<class dataType > void os::AVL_NODE< dataType >::removeChild ( int pos )  
[inline], [private]
```

Remove a child from this node.

Remove `os::AVL_NODE<dataType>::child1` if position is 0 and `os::AVL_NODE<dataType>::child2` if position is 1.

Parameters

in	pos	Node index to be removed
----	-----	--------------------------

Returns

void

```
template<class dataType > void os::AVL_NODE< dataType >::removeParent ( ) [inline],  
[private]
```

Remove the parent node.

Returns

void

```
template<class dataType > void os::AVL_NODE< dataType >::setChild ( smart_ptr<  
AVL_NODE< dataType > > c ) [inline], [private]
```

Add a child to this node.

Set `os::AVL_NODE<dataType>::child1` or `os::AVL_NODE<dataType>::child2` based on the comparison of the node to be inserted with the current node.

Parameters

in	c	Node to be inserted
----	---	---------------------

Returns

void

```
template<class dataType > void os::AVL_NODE< dataType >::setParent ( smart_ptr<  
AVL_NODE< dataType > > p, smart_ptr< AVL_NODE< dataType > > self_pointer ) [inline],  
[private]
```

Sets the parent node.

Sets the parent node of the current node. This function requires a pointer to the current node for memory management.

Parameters

in	<i>p</i>	Parent node
in	<i>self_pointer</i>	Pointer to self, with memory management

Returns

void

12.2.4 Member Data Documentation

```
template<class dataType > smart_ptr<AVL_NODE<dataType> > os::AVL_NODE< dataType  
>::_child[2] [private]
```

Children nodes.

```
template<class dataType > size_t os::AVL_NODE< dataType >::_height [private]
```

The height of the tree.

This variable is kept to reduce computation time. It is dependent on the height of a node's children nodes. The **AVL_NODE**<dataType>::_propagateUp() (p. 114) resets the height based on the height of the node's children.

```
template<class dataType > size_t os::AVL_NODE< dataType >::_numChildren [private]
```

Number of children.

This variable is kept to reduce computation time. It is dependent on the number of children both children have.

```
template<class dataType > smart_ptr<AVL_NODE<dataType> > os::AVL_NODE< dataType  
>::_parent [private]
```

Parent node one level up in the tree.

```
template<class dataType > const bool os::AVL_NODE< dataType >::ITERABLE = true [static]
```

AVL nodes are iterable.

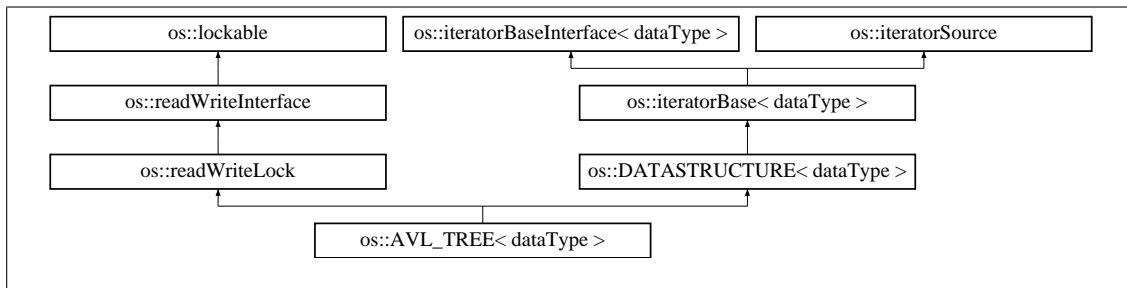
```
template<class dataType > const bool os::AVL_NODE< dataType >::RANDOM_ACCESS = true  
[static]
```

AVL nodes allow random-access.

12.3 os::AVL_TREE< dataType > Class Template Reference

Template AVL Tree definition.

Inheritance diagram for os::AVL_TREE< dataType >:



Public Member Functions

- **AVL_TREE ()**
Default constructor Initializes an empty AVL tree.
- **~AVL_TREE ()**
- **size_t size ()** const final
Access size of the tree.
- **bool iterable ()** const final
Returns if the relevant node type is iterable.
- **bool randomAccess ()** const final
Returns if the relevant node type can be accessed randomly.
- **bool insert (const dataType &x)** final
Insert item into the tree Inserts an item into the tree and procedes to re-balance the tree.
- **bool remove (const dataType &x)** final
Remove item from the data-structure.
- **bool find (const dataType &x)** const final
Searches for an item.
- **dataType & access (const dataType &x)** final
Mutable item access.
- **const dataType & access (const dataType &x)** const final
Immutable item access.
- **dataType & at (size_t i)** final throw (descriptiveException)
Access the tree by index.
- **const dataType & at (size_t i)** const final throw (descriptiveException)
Constant access the tree by index.

Static Public Attributes

- static const bool **ITERABLE** = true
AVL trees are iterable.
- static const bool **RANDOM_ACCESS** = true
AVL trees allow random access.

Protected Member Functions

- **smart_ptr< nodeFrame< dataType > > getFirstNode ()** final
Access to first node.
- **smart_ptr< nodeFrame< dataType > > getLastNode ()** final
Access to last node.
- const **smart_ptr< nodeFrame< dataType > > getFirstNodeConst ()** const final
Constant access to first node.
- const **smart_ptr< nodeFrame< dataType > > getLastNodeConst ()** const final
Constant access to last node.
- **smart_ptr< nodeFrame< dataType > > searchNode (const smart_ptr< dataType > dt)** final
Search for a node.
- const **smart_ptr< nodeFrame< dataType > > searchNodeConst (const smart_ptr< dataType > dt)** const final
Const search for a node.

Private Member Functions

- bool **checkBalance (smart_ptr< AVL_NODE< dataType > > x)**
Checks if a sub-tree is balanced.
- bool **singleRotation (smart_ptr< AVL_NODE< dataType > > r, int dir)**
Rotates a node.
- bool **doubleRotation (smart_ptr< AVL_NODE< dataType > > r, int dir)**
Double-rotate a node.
- **smart_ptr< AVL_NODE< dataType > > findBottom (smart_ptr< AVL_NODE< dataType > > x, int dir)**
Find first or last node in a tree.
- bool **balance (smart_ptr< AVL_NODE< dataType > > x)**
Balances a single node.
- void **balanceUp (smart_ptr< AVL_NODE< dataType > > x)**
Balances this node and ancestor nodes.
- bool **balanceDelete (smart_ptr< AVL_NODE< dataType > > x)**
Removes a node and balances the tree.

Private Attributes

- **smart_ptr< AVL_NODE< dataType > > _root**
Root node of the tree.

Additional Inherited Members

12.3.1 Detailed Description

```
template<class dataType>
class os::AVL_TREE< dataType >
```

Template AVL Tree definition.

Note that there are 6 different versions of this class defined, allowing for multiple pointer and thread-safety definitions.

12.3.2 Constructor & Destructor Documentation

```
template<class dataType> os::AVL_TREE< dataType >::AVL_TREE ( ) [inline]
```

Default constructor Initializes an empty AVL tree.

```
template<class dataType> os::AVL_TREE< dataType >::~AVL_TREE ( ) [inline]
```

AVL tree destructor Destroys all objects held in the tree. Note that in the case the tree is holding references, the pointers will be destroyed instead of the objects.

12.3.3 Member Function Documentation

```
template<class dataType> dataType& os::AVL_TREE< dataType >::access ( const dataType & x
) [inline], [final], [virtual]
```

Mutable item access.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Returns

Mutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 157).

```
template<class dataType> const dataType& os::AVL_TREE< dataType >::access ( const
dataType & x ) const [inline], [final], [virtual]
```

Immutable item access.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Returns

Immutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 157).

```
template<class dataType> dataType& os::AVL_TREE< dataType >::at ( size_t i ) throw
descriptiveException) [inline], [final], [virtual]
```

Access the tree by index.

Parameters

in	<i>i</i>	Index of the element
-----------	----------	----------------------

Returns

Reference to ith element of the tree

Reimplemented from **os::DATASTRUCTURE**< **dataType** > (p. 158).

```
template<class dataType> const dataType& os::AVL_TREE< dataType >::at ( size_t i ) const
throw descriptiveException) [inline], [final], [virtual]
```

Constant access the tree by index.

Parameters

in	<i>i</i>	Index of the element
-----------	----------	----------------------

Returns

Immutable reference to ith element of the tree

Reimplemented from **os::DATASTRUCTURE**< **dataType** > (p. 158).

```
template<class dataType> bool os::AVL_TREE< dataType >::balance ( smart_ptr< AVL_NODE<
dataType > > x ) [inline], [private]
```

Balances a single node.

Parameters

in	<i>x</i>	Node to be balanced
-----------	----------	---------------------

Returns

true if the node is already balanced, else, false

```
template<class dataType> bool os::AVL_TREE< dataType >::balanceDelete ( smart_ptr<
AVL_NODE< dataType > > x ) [inline], [private]
```

Removes a node and balances the tree.

Must receive as an argument a node in the tree. This function removes the node from the tree and re-balances the tree.

Parameters

in	x	Node to be deleted
----	---	--------------------

Returns

true if successful, false if failed

```
template<class dataType> void os::AVL_TREE< dataType >::balanceUp ( smart_ptr<  
AVL_NODE< dataType > > x ) [inline], [private]
```

Balances this node and ancestor nodes.

Balances the current node then orders it's parent node to be balanced as well. This process continues until a node has no parent (indicating the node is the root)

Parameters

in	x	Node to be balanced
----	---	---------------------

Returns

void

```
template<class dataType> bool os::AVL_TREE< dataType >::checkBalance ( smart_ptr<  
AVL_NODE< dataType > > x ) [inline], [private]
```

Checks if a sub-tree is balanced.

Checks if the received node is balanced. This operation is inexpensive as it merely involves comparing the heights of the children nodes.

Parameters

in	x	Node to be checked
----	---	--------------------

Returns

true if balanced, false if not

```
template<class dataType> bool os::AVL_TREE< dataType >::doubleRotation ( smart_ptr<  
AVL_NODE< dataType > > r, int dir ) [inline], [private]
```

Double-rotate a node.

Double-rotates a node based on the dir argument provided. Note that 0 and 1 are the only valid directions.

Parameters

in	x	Node to be rotated
in	dir	Direction node is to be rotated

Returns

true if successful, else, false

```
template<class dataType> bool os::AVL_TREE< dataType >::find ( const dataType & x ) const  
[inline], [final], [virtual]
```

Searches for an item.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references.

Returns

True if found, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 158).

```
template<class dataType> smart_ptr<AVL_NODE<dataType> > os::AVL_TREE< dataType  
>::findBottom ( smart_ptr< AVL_NODE< dataType > > x, int dir ) [inline], [private]
```

Find first or last node in a tree.

Finds the first or last node based on the dir argument provided. Note that 0 and 1 are the only valid directions.

Parameters

in	<i>x</i>	Starting node
in	<i>dir</i>	Direction node to search in

Returns

First or last node in sub-tree

```
template<class dataType> smart_ptr<nodeFrame<dataType> > os::AVL_TREE< dataType  
>::getFirstNode ( ) [inline], [final], [protected], [virtual]
```

Access to first node.

Returns

First node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 198).

```
template<class dataType> const smart_ptr<nodeFrame<dataType> > os::AVL_TREE< dataType  
>::getFirstNodeConst ( ) const [inline], [final], [protected], [virtual]
```

Constant access to first node.

Returns

Immutable first node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 198).

```
template<class dataType> smart_ptr<nodeFrame<dataType> > os::AVL_TREE< dataType
>::getLastNode ( ) [inline], [final], [protected], [virtual]
```

Access to last node.

Returns

Last node in the structure

Implements **os::iteratorBase< dataType >** (p. 198).

```
template<class dataType> const smart_ptr<nodeFrame<dataType> > os::AVL_TREE< dataType
>::getLastNodeConst ( ) const [inline], [final], [protected], [virtual]
```

Constant access to last node.

Returns

Immutable last node in the structure

Implements **os::iteratorBase< dataType >** (p. 199).

```
template<class dataType> bool os::AVL_TREE< dataType >::insert ( const dataType & x )
[inline], [final], [virtual]
```

Insert item into the tree Inserts an item into the tree and procedes to re-balance the tree.

Parameters

in	x	Data to be bound
-----------	----------	------------------

Returns

True if successful, else, false

Implements **os::DATASTRUCTURE< dataType >** (p. 159).

```
template<class dataType> bool os::AVL_TREE< dataType >::iterable ( ) const [inline],
[final], [virtual]
```

Returns if the relevant node type is iterable.

Returns

true

Reimplemented from **os::iteratorSource** (p. 205).

```
template<class dataType> bool os::AVL_TREE< dataType >::randomAccess ( ) const
[inline], [final], [virtual]
```

Returns if the relevant node type can be accessed randomly.

Returns

true

Reimplemented from **os::iteratorSource** (p. 205).

```
template<class dataType> bool os::AVL_TREE< dataType >::remove ( const dataType & x )  
[inline], [final], [virtual]
```

Remove item from the data-structure.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references.

Returns

True if removed, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 160).

```
template<class dataType> smart_ptr<nodeFrame<dataType> > os::AVL_TREE< dataType  
>::searchNode ( const smart_ptr< dataType > dt ) [inline], [final], [protected],  
[virtual]
```

Search for a node.

Parameters

in	dt	Pointer to search for
----	----	-----------------------

Returns

Mutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 199).

```
template<class dataType> const smart_ptr<nodeFrame<dataType> > os::AVL_TREE<  
dataType >::searchNodeConst ( const smart_ptr< dataType > dt ) const [inline], [final],  
[protected], [virtual]
```

Const search for a node.

Parameters

in	dt	Pointer to search for
----	----	-----------------------

Returns

Immutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 200).

```
template<class dataType> bool os::AVL_TREE< dataType >::singleRotation ( smart_ptr<  
AVL_NODE< dataType > > r, int dir ) [inline], [private]
```

Rotates a node.

Rotates a node based on the dir argument provided. Note that 0 and 1 are the only valid directions.

Parameters

in	<i>x</i>	Node to be rotated
in	<i>dir</i>	Direction node is to be rotated

Returns

true if successful, else, false

```
template<class dataType> size_t os::AVL_TREE< dataType >::size ( ) const [inline],  
[final], [virtual]
```

Access size of the tree.

Returns

Number of elements in the tree

Implements **os::DATASTRUCTURE**< **dataType** > (p. 160).

12.3.4 Member Data Documentation

```
template<class dataType> smart_ptr<AVL_NODE< dataType > > os::AVL_TREE< dataType  
>::_root [private]
```

Root node of the tree.

```
template<class dataType> const bool os::AVL_TREE< dataType >::ITERABLE = true [static]
```

AVL trees are iterable.

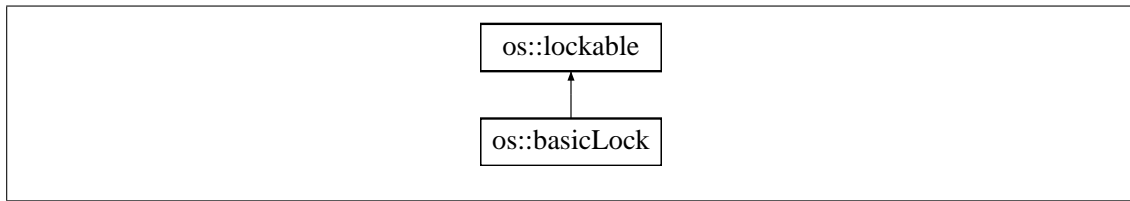
```
template<class dataType> const bool os::AVL_TREE< dataType >::RANDOM_ACCESS = true  
[static]
```

AVL trees allow random access.

12.4 os::basicLock Class Reference

Basic lock Wraps the std::mutex class allowing it to be accessed in a constant way.

Inheritance diagram for os::basicLock:



Public Member Functions

- **basicLock** (bool dMode=**NO_LOCK_CHECK**) throw ()
Default constructor.
- virtual ~**basicLock** () throw (descriptiveException)
Virtual destructor.
- void **lock** () const final throw ()
Locks the std::mutex.
- void **unlock** () const final throw (descriptiveException)
Unlocks the std::mutex.
- bool **locked** () const final throw ()
Checks if the lock is locked.
- bool **try_lock** () const final throw ()
Attempt to lock the object.

Private Member Functions

- **basicLock** (const **basicLock** &cpy)
Undefined copy-constructor.

Private Attributes

- std::mutex **_mtx**
Mutable mutex.
- bool **_lockedStatus**
Locked flag.

Additional Inherited Members

12.4.1 Detailed Description

Basic lock Wraps the std::mutex class allowing it to be accessed in a constant way.

12.4.2 Constructor & Destructor Documentation

os::basicLock::basicLock (const **basicLock** &cpy) [inline], [private]

Undefined copy-constructor.

os::basicLock::basicLock (bool dMode = **NO_LOCK_CHECK**) throw)

Default constructor.

virtual os::basicLock::~~basicLock () throw **descriptiveException**) [virtual]

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.4.3 Member Function Documentation

void os::basicLock::lock () const throw) [final], [virtual]

Locks the std::mutex.

Returns

void

Implements **os::lockable** (p. 223).

bool os::basicLock::locked () const throw) [inline], [final], [virtual]

Checks if the lock is locked.

Returns

True if locked, else, false

Implements **os::lockable** (p. 223).

bool os::basicLock::try_lock () const throw) [final], [virtual]

Attempt to lock the object.

Locks the object if possible, otherwise, returns false.

Returns

True if lock successful

Implements **os::lockable** (p. 223).

void os::basicLock::unlock () const throw **descriptiveException**) [final], [virtual]

Unlocks the std::mutex.

Returns

void

Implements **os::lockable** (p. 223).

12.4.4 Member Data Documentation

`bool os::basicLock::_lockedStatus` [mutable], [private]

Locked flag.

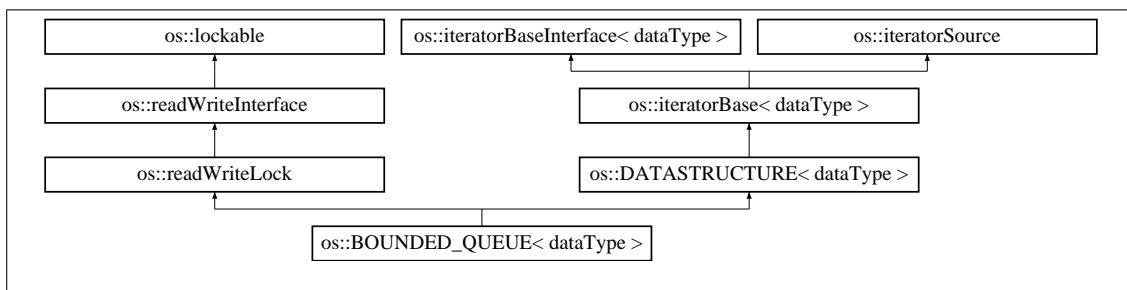
`std::mutex os::basicLock::_mtx` [mutable], [private]

Mutable mutex.

12.5 `os::BOUNDED_QUEUE< dataType >` Class Template Reference

Template **BOUNDED_QUEUE** (p. 129) definition.

Inheritance diagram for `os::BOUNDED_QUEUE< dataType >`:



Public Member Functions

- **BOUNDED_QUEUE** (size_t sz=128)

Default constructor.

- virtual **~BOUNDED_QUEUE** ()

BOUNDED_QUEUE (p. 129) destructor Destroys all objects held in the array. Note that in the case the **BOUNDED_QUEUE** (p. 129) is holding references, the pointers will be destroyed instead of the objects.

- size_t **boundSize** () const

Array bound size.

- void **pop** ()

Remove top element.

- void **setBound** (size_t sz)

Sets the size of the bounding array.

- bool **insert** (const dataType &x) final

*Insert item into the **BOUNDED_QUEUE** (p. 129).*

- bool **remove** (const dataType &x) final

Remove item from the data-structure.

- bool **find** (const dataType &x) const final

Searches for an item.

- **dataType & access** (const dataType &x) final
Mutable item access.
- **const dataType & access** (const dataType &x) const final
Immutable item access.
- **dataType & at** (size_t i) final throw (descriptiveException)
*Access the **BOUNDED_QUEUE** (p. 129) by index.*
- **const dataType & at** (size_t i) const final throw (descriptiveException)
*Access the **BOUNDED_QUEUE** (p. 129) by index.*
- **bool push** (const dataType &x)
Add element.
- **const dataType & constTop** () const
Const access top element.
- **dataType & top** ()
Access top element.
- **const dataType & top** () const
Const access top element.
- **size_t size** () const final
*Access size of the **BOUNDED_QUEUE** (p. 129).*
- **bool iterable** () const final
Returns if the relevant node type is iterable.
- **bool randomAccess** () const final
Returns if the relevant node type can be accessed randomly.

Static Public Attributes

- static const bool **ITERABLE** = true
BOUNDED_QUEUEs are iterable.
- static const bool **RANDOM_ACCESS** = true
BOUNDED_QUEUEs allow random access.

Protected Member Functions

- **smart_ptr< nodeFrame< dataType > > getFirstNode** () final
Access to first node.
- **smart_ptr< nodeFrame< dataType > > getLastNode** () final
Access to last node.
- **const smart_ptr< nodeFrame< dataType > > getFirstNodeConst** () const final
Constant access to first node.
- **const smart_ptr< nodeFrame< dataType > > getLastNodeConst** () const final
Constant access to last node.
- **smart_ptr< nodeFrame< dataType > > searchNode** (const smart_ptr< dataType > dt) final
Search for a node.
- **const smart_ptr< nodeFrame< dataType > > searchNodeConst** (const smart_ptr< dataType > dt) const final
Const search for a node.

Private Attributes

- **dataType * _array**
Pointer to array of data.
- **size_t _arraySize**
Size of the available memory.
- **size_t numElms**
*Number of elements in the **BOUNDED_QUEUE** (p. 129).*
- **size_t pos**
Starting position.

Additional Inherited Members

12.5.1 Detailed Description

```
template<class dataType>
class os::BOUNDED_QUEUE< dataType >
```

Template **BOUNDED_QUEUE** (p. 129) definition.

Note that there are 6 different versions of this class defined, allowing for multiple pointer and thread-safety definitions.

12.5.2 Constructor & Destructor Documentation

```
template<class dataType> os::BOUNDED_QUEUE< dataType >::BOUNDED_QUEUE ( size_t sz
= 128 ) [inline]
```

Default constructor.

This constructor builds the vecotr with a certain size. Note that the **BOUNDED_QUEUE** (p. 129) will always be of, at least, size 32.

Parameters

in	sz	Target size of BOUNDED_QUEUE (p. 129)
----	----	--

```
template<class dataType> virtual os::BOUNDED_QUEUE< dataType >::~~BOUNDED_QUEUE (
) [inline], [virtual]
```

BOUNDED_QUEUE (p. 129) destructor Destroys all objects held in the array. Note that in the case the **BOUNDED_QUEUE** (p. 129) is holding references, the pointers will be destroyed instead of the objects.

12.5.3 Member Function Documentation

```
template<class dataType> dataType& os::BOUNDED_QUEUE< dataType >::access ( const
dataType & x ) [inline], [final], [virtual]
```

Mutable item access.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Returns

Mutable item equal to x

Implements **os::DATASTRUCTURE< dataType >** (p. 157).

```
template<class dataType> const dataType& os::BOUNDED_QUEUE< dataType >::access ( const  
dataType & x ) const [inline], [final], [virtual]
```

Immutable item access.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Returns

Immutable item equal to x

Implements **os::DATASTRUCTURE< dataType >** (p. 157).

```
template<class dataType> dataType& os::BOUNDED_QUEUE< dataType >::at ( size_t i ) throw  
descriptiveException) [inline], [final], [virtual]
```

Access the **BOUNDED_QUEUE** (p. 129) by index.

Parameters

in	<i>i</i>	Index of the array
-----------	----------	--------------------

Returns

Reference to ith element of the **BOUNDED_QUEUE** (p. 129)

Reimplemented from **os::DATASTRUCTURE< dataType >** (p. 158).

```
template<class dataType> const dataType& os::BOUNDED_QUEUE< dataType >::at ( size_t i )  
const throw descriptiveException) [inline], [final], [virtual]
```

Access the **BOUNDED_QUEUE** (p. 129) by index.

Parameters

in	<i>i</i>	Index of the array
-----------	----------	--------------------

Returns

Immutable reference to ith element of the **BOUNDED_QUEUE** (p. 129)

Reimplemented from **os::DATASTRUCTURE< dataType >** (p. 158).

```
template<class dataType> size_t os::BOUNDED_QUEUE< dataType >::boundSize ( ) const  
[inline]
```

Array bound size.

Returns

os::BOUNDED_QUEUE<dataType>::_arraySize (p. 137)

```
template<class dataType> const dataType& os::BOUNDED_QUEUE< dataType >::constTop ( )  
const [inline]
```

Const access top element.

Returns the top element of the queue. Note that this operation will throw an exception if the queue is empty.

Returns

Immutable top element

```
template<class dataType> bool os::BOUNDED_QUEUE< dataType >::find ( const dataType & x )  
const [inline], [final], [virtual]
```

Searches for an item.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references.

Returns

True if found, else, false

Implements **os::DATASTRUCTURE< dataType >** (p. 158).

```
template<class dataType> smart_ptr<nodeFrame<dataType> > os::BOUNDED_QUEUE<  
dataType >::getFirstNode ( ) [final], [protected], [virtual]
```

Access to first node.

Returns

First node in the structure

Implements **os::iteratorBase< dataType >** (p. 198).

```
template<class dataType> const smart_ptr<nodeFrame<dataType>> > os::BOUNDED_QUEUE<
dataType>::getFirstNodeConst ( ) const [final], [protected], [virtual]
```

Constant access to first node.

Returns

Immutable first node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 198).

```
template<class dataType> smart_ptr<nodeFrame<dataType>> > os::BOUNDED_QUEUE<
dataType>::getLastNode ( ) [final], [protected], [virtual]
```

Access to last node.

Returns

Last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 198).

```
template<class dataType> const smart_ptr<nodeFrame<dataType>> > os::BOUNDED_QUEUE<
dataType>::getLastNodeConst ( ) const [final], [protected], [virtual]
```

Constant access to last node.

Returns

Immutable last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 199).

```
template<class dataType> bool os::BOUNDED_QUEUE< dataType >::insert ( const dataType & x
) [inline], [final], [virtual]
```

Insert item into the **BOUNDED_QUEUE** (p. 129).

Inserts an item at the end of the **BOUNDED_QUEUE** (p. 129). Note that different forms of the **BOUNDED_QUEUE** (p. 129) accept pointers instead of object references.

Returns

True

Implements **os::DATASTRUCTURE**< **dataType** > (p. 159).

```
template<class dataType> bool os::BOUNDED_QUEUE< dataType >::iterable ( ) const
[inline], [final], [virtual]
```

Returns if the relevant node type is iterable.

Returns

true

Reimplemented from **os::iteratorSource** (p. 205).

```
template<class dataType> void os::BOUNDED_QUEUE< dataType >::pop ( ) [inline]
```

Remove top element.

Attempts to remove the top element. Throws an exception if the stack is empty.

Returns

void

```
template<class dataType> bool os::BOUNDED_QUEUE< dataType >::push ( const dataType & x ) [inline]
```

Add element.

Inserts an element at the front of the queue

Parameters

in	x	Element to be inserted
----	---	------------------------

Returns

true

```
template<class dataType> bool os::BOUNDED_QUEUE< dataType >::randomAccess ( ) const [inline], [final], [virtual]
```

Returns if the relevant node type can be accessed randomly.

Returns

true

Reimplemented from **os::iteratorSource** (p. 205).

```
template<class dataType> bool os::BOUNDED_QUEUE< dataType >::remove ( const dataType & x ) [inline], [final], [virtual]
```

Remove item from the data-structure.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references.

Returns

True if removed, else, false

Implements **os::DATASTRUCTURE**< dataType > (p. 160).

```
template<class dataType> smart_ptr<nodeFrame<dataType> > os::BOUNDED_QUEUE< dataType >::searchNode ( const smart_ptr< dataType > dt ) [final], [protected], [virtual]
```

Search for a node.

Parameters

in	dt	Pointer to search for
----	----	-----------------------

Returns

Mutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 199).

```
template<class dataType> const smart_ptr<nodeFrame<dataType> > os::BOUNDED_QUEUE<
dataType >::searchNodeConst ( const smart_ptr< dataType > dt ) const  [final], [protected],
[virtual]
```

Const search for a node.

Parameters

in	dt	Pointer to search for
----	----	-----------------------

Returns

Immutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 200).

```
template<class dataType> void os::BOUNDED_QUEUE< dataType >::setBound ( size_t sz )
[inline]
```

Sets the size of the bounding array.

Parameters

in	sz	Bound size
----	----	------------

Returns

void

```
template<class dataType> size_t os::BOUNDED_QUEUE< dataType >::size ( ) const
[inline], [final], [virtual]
```

Access size of the **BOUNDED_QUEUE** (p. 129).

Returns

Number of elements in the **BOUNDED_QUEUE** (p. 129)

Implements **os::DATASTRUCTURE**< **dataType** > (p. 160).

```
template<class dataType> dataType& os::BOUNDED_QUEUE< dataType >::top ( ) [inline]
```

Access top element.

Returns the top element of the queue. Note that this operation will throw an exception if the queue is empty.

Returns

Mutable top element

```
template<class dataType> const dataType& os::BOUNDED_QUEUE< dataType >::top ( ) const  
[inline]
```

Const access top element.

Returns the top element of the queue. Note that this operation will throw an exception if the queue is empty.

Returns

Immutable top element

12.5.4 Member Data Documentation

```
template<class dataType> dataType* os::BOUNDED_QUEUE< dataType >::_array [private]
```

Pointer to array of data.

```
template<class dataType> size_t os::BOUNDED_QUEUE< dataType >::_arraySize [private]
```

Size of the available memory.

```
template<class dataType> const bool os::BOUNDED_QUEUE< dataType >::ITERABLE = true  
[static]
```

BOUNDED_QUEUEs are iterable.

```
template<class dataType> size_t os::BOUNDED_QUEUE< dataType >::numElms [private]
```

Number of elements in the **BOUNDED_QUEUE** (p. 129).

```
template<class dataType> size_t os::BOUNDED_QUEUE< dataType >::pos [private]
```

Starting position.

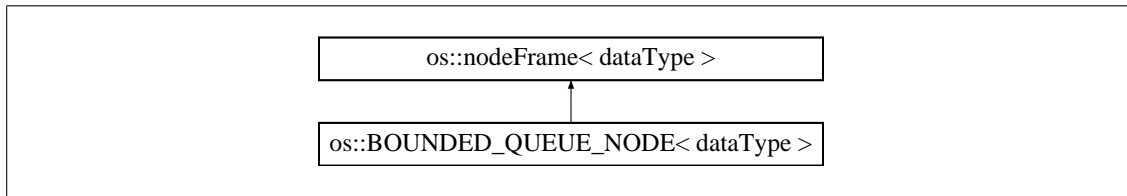
```
template<class dataType> const bool os::BOUNDED_QUEUE< dataType >::RANDOM_ACCESS  
= true [static]
```

BOUNDED_QUEUEs allow random access.

12.6 os::BOUNDED_QUEUE_NODE< dataType > Class Template Reference

BOUNDED_QUEUE (p. 129) node.

Inheritance diagram for os::BOUNDED_QUEUE_NODE< dataType >:



Public Member Functions

- **~BOUNDED_QUEUE_NODE** ()
Destructor Class is not designed to be inherited from.
- **smart_ptr< dataType > get** () final throw ()
Returns a pointer.
- **const smart_ptr< dataType > constGet** () const final throw ()
Returns a const pointer.
- **dataType & operator*** () final throw (descriptiveException)
De-reference.
- **const dataType & operator*** () const final throw (descriptiveException)
De-reference.
- **bool valid** () const final
Valid data query Checks if the provided data is valid.
- **bool iterable** () const final
Returns if the node is iterable.
- **bool randomAccess** () const final
Returns if the node can be accessed randomly.
- **smart_ptr< nodeFrame< dataType > > getNext** () final throw (descriptiveException)
*Returns the next **BOUNDED_QUEUE** (p. 129) frame.*
- **const smart_ptr< nodeFrame< dataType > > getNextConst** () const final throw (descriptiveException)
*Returns the next **BOUNDED_QUEUE** (p. 129) frame.*
- **smart_ptr< nodeFrame< dataType > > getPrev** () final throw (descriptiveException)
*Returns the previous **BOUNDED_QUEUE** (p. 129) frame.*
- **const smart_ptr< nodeFrame< dataType > > getPrevConst** () const final throw (descriptiveException)
*Returns the previous **BOUNDED_QUEUE** (p. 129) frame.*
- **smart_ptr< nodeFrame< dataType > > access** (long offset) final throw (descriptiveException)
Access node by index.

- const **smart_ptr< nodeFrame< dataType > > constAccess** (long offset) const final throw (descriptiveException)
Access node by index.
- void **remove** () final throw (descriptiveException)
Remove this node from the **BOUNDED_QUEUE** (p. 129).

Static Public Attributes

- static const bool **ITERABLE** = true
BOUNDED_QUEUE (p. 129) frames are iterable.
- static const bool **RANDOM_ACCESS** = true
BOUNDED_QUEUE (p. 129) frames allow random-access.

Private Member Functions

- **BOUNDED_QUEUE_NODE** (**BOUNDED_QUEUE**< dataType > *src, size_t pos)
Private constructor This constructor is not designed to be accessed by anything other than the **BOUNDED_QUEUE** (p. 129) class derivatives.

Private Attributes

- size_t **position**
Current position of the **BOUNDED_QUEUE** (p. 129) iterator.

12.6.1 Detailed Description

```
template<class dataType>
class os::BOUNDED_QUEUE_NODE< dataType >
```

BOUNDED_QUEUE (p. 129) node.

Used by the iterator to iterate through a **BOUNDED_QUEUE** (p. 129).

12.6.2 Constructor & Destructor Documentation

```
template<class dataType > os::BOUNDED_QUEUE_NODE< dataType >::BOUNDED_QUEUE_NODE (
    BOUNDED_QUEUE< dataType > * src, size_t pos ) [inline],
[private]
```

Private constructor This constructor is not designed to be accessed by anything other than the **BOUNDED_QUEUE** (p. 129) class derivatives.

```
template<class dataType > os::BOUNDED_QUEUE_NODE< dataType >::~BOUNDED_QUEUE_NODE ( ) [inline]
```

Destructor Class is not designed to be inherited from.

12.6.3 Member Function Documentation

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::BOUNDED_QUEUE_NODE< dataType >::access ( long offset ) throw descriptiveException ) [inline], [final], [virtual]
```

Access node by index.

Access a node offset from the current node by some value. If a node cannot be randomly accessed, an exception will be thrown.

Returns

Offset node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 236).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::BOUNDED_QUEUE_NODE< dataType >::constAccess ( long offset ) const throw descriptiveException ) [inline], [final], [virtual]
```

Access node by index.

Access a node offset from the current node by some value. If a node cannot be randomly accessed, an exception will be thrown.

Returns

Offset node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 236).

```
template<class dataType > const smart_ptr<dataType> os::BOUNDED_QUEUE_NODE< dataType >::constGet ( ) const throw ) [inline], [final], [virtual]
```

Returns a const pointer.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

Implements **os::nodeFrame**< **dataType** > (p. 236).

```
template<class dataType > smart_ptr<dataType> os::BOUNDED_QUEUE_NODE< dataType >::get ( ) throw ) [inline], [final], [virtual]
```

Returns a pointer.

Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

Implements **os::nodeFrame**< **dataType** > (p. 237).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::BOUNDED_QUEUE_  
UE_NODE< dataType >::getNext ( ) throw descriptiveException) [inline], [final],  
[virtual]
```

Returns the next **BOUNDED_QUEUE** (p. 129) frame.

Returns

Next node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 237).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::BOUNDED_QUEUE_  
UE_NODE< dataType >::getNextConst ( ) const throw descriptiveException) [inline],  
[final], [virtual]
```

Returns the next **BOUNDED_QUEUE** (p. 129) frame.

Returns

Next node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 238).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::BOUNDED_QUEUE_  
UE_NODE< dataType >::getPrev ( ) throw descriptiveException) [inline], [final],  
[virtual]
```

Returns the previous **BOUNDED_QUEUE** (p. 129) frame.

Returns

Previous node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 238).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::BOUNDED_QUEUE_  
UE_NODE< dataType >::getPrevConst ( ) const throw descriptiveException) [inline],  
[final], [virtual]
```

Returns the previous **BOUNDED_QUEUE** (p. 129) frame.

Returns

Previous node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 238).

```
template<class dataType > bool os::BOUNDED_QUEUE_NODE< dataType >::iterable ( ) const  
[inline], [final], [virtual]
```

Returns if the node is iterable.

Returns

object::ITERABLE

Reimplemented from **os::nodeFrame**< **dataType** > (p. 239).

```
template<class dataType > dataType& os::BOUNDED_QUEUE_NODE< dataType >::operator* (
) throw descriptiveException) [inline], [final], [virtual]
```

De-reference.

Returns a reference to the contained object, the reference can be modified.

Returns

Contained object

Implements **os::nodeFrame**< **dataType** > (p. 239).

```
template<class dataType > const dataType& os::BOUNDED_QUEUE_NODE< dataType
>::operator* ( ) const throw descriptiveException) [inline], [final], [virtual]
```

De-reference.

Returns a const reference to the contained object, the reference cannot be modified.

Returns

Contained object

Implements **os::nodeFrame**< **dataType** > (p. 240).

```
template<class dataType > bool os::BOUNDED_QUEUE_NODE< dataType >::randomAccess ( )
const [inline], [final], [virtual]
```

Returns if the node can be accessed randomly.

Returns

object::RANDOM_ACCESS

Reimplemented from **os::nodeFrame**< **dataType** > (p. 240).

```
template<class dataType > void os::BOUNDED_QUEUE_NODE< dataType >::remove ( ) throw
descriptiveException) [inline], [final], [virtual]
```

Remove this node from the **BOUNDED_QUEUE** (p. 129).

Returns

void

Reimplemented from **os::nodeFrame**< **dataType** > (p. 240).

```
template<class dataType > bool os::BOUNDED_QUEUE_NODE< dataType >::valid ( ) const
[inline], [final], [virtual]
```

Valid data query Checks if the provided data is valid.

Returns

true if valid, else, false

Reimplemented from **os::nodeFrame**< **dataType** > (p. 241).

12.6.4 Member Data Documentation

```
template<class dataType > const bool os::BOUNDED_QUEUE_NODE< dataType >::ITERABLE =  
true [static]
```

BOUNDED_QUEUE (p. 129) frames are iterable.

```
template<class dataType > size_t os::BOUNDED_QUEUE_NODE< dataType >::position  
[private]
```

Current position of the **BOUNDED_QUEUE** (p. 129) iterator.

```
template<class dataType > const bool os::BOUNDED_QUEUE_NODE< dataType  
>::RANDOM_ACCESS = true [static]
```

BOUNDED_QUEUE (p. 129) frames allow random-access.

12.7 os::constantPrinter Class Reference

Prints constant arrays to files.

Public Member Functions

- **constantPrinter** (std::string fileName, bool has_cpp=false) throw (descriptiveException)
Single constructor.
- virtual ~**constantPrinter** () throw (descriptiveException)
Virtual destructor.
- void **addInclude** (std::string includeName) throw (descriptiveException)
Add include file.
- void **addNamespace** (std::string namesp) throw (descriptiveException)
Add a namespace.
- void **removeNamespace** () throw (descriptiveException)
Remove namespace.
- void **addComment** (std::string comment) throw (descriptiveException)
Insert a comment.
- bool **hasCPP** () const throw ()
Returns if the object is writing to a .cpp file.
- bool **good** () const throw ()
Checks file status.
- void **addArray** (std::string name, uint32_t *arr, unsigned length) throw (descriptiveException)
Add a uin32_t array.*

Private Member Functions

- `std::string capitalize (std::string str) const throw ()`
Capitalizes the string argument.
- `std::string tabs () const throw ()`
Returns current tab depth.

Private Attributes

- `std::ofstream hFile`
Output file for the .h file.
- `std::ofstream cppFile`
Output file for the .cpp file.
- `bool _has_cpp`
Holds if the object is generating a .cpp.
- `unsigned namespaceDepth`
Current namespace depth.

12.7.1 Detailed Description

Prints constant arrays to files.

This class outputs configured and populated constant arrays into .h and .cpp files, depending on the configuration. This class is meant to be used as a tool for automatically generating source code files.

12.7.2 Constructor & Destructor Documentation

`os::constantPrinter::constantPrinter (std::string fileName, bool has_cpp = false) throw descriptiveException`

Single constructor.

Creates a file of "filename.h" and, if `has_cpp` is set to "true," "filename.cpp" with appropriate include guards and a comment indicating the source of the file.

Parameters

in	<i>fileName</i>	String representing the file name
in	<i>has_cpp</i>	Optional boolean defining if a .cpp will be written

`virtual os::constantPrinter::~~constantPrinter () throw descriptiveException [virtual]`

Virtual destructor.

Closes all namespaces and `#ifdefs`, closes the .h file and .cpp if appropriate.

12.7.3 Member Function Documentation

`void os::constantPrinter::addArray (std::string name, uint32_t * arr, unsigned length) throw descriptiveException`

Add a `uint32_t*` array.

Added an unsigned 32 bit integer array to the .h and .cpp file. Note that this array will be declared as constant.

Parameters

in	<i>arr</i>	Array to be written to the files
in	<i>length</i>	Length of the received array

Returns

`void`

`void os::constantPrinter::addComment (std::string comment) throw descriptiveException`

Insert a comment.

Adds a comment. If the comment is a single line, `/'` will be used, otherwise, a standard multi-line comment format will be used.

Parameters

in	<i>comment</i>	Comment string to be added as a comment
----	----------------	---

Returns

`void`

`void os::constantPrinter::addInclude (std::string includeName) throw descriptiveException`

Add include file.

Prints out `"#include includeName"` to the .h file. Since the .cpp file includes the .h file, it will include all of the .h file's includes

Parameters

in	<i>includeName</i>	Name of header file to be included
----	--------------------	------------------------------------

Returns

`void`

`void os::constantPrinter::addNamespace (std::string namesp) throw descriptiveException`

Add a namespace.

Adds a new namespace. Namespaces nest, so this function increments **constantPrinter::namespaceDepth** (p. 147). Both the .h and .cpp file have this namespace added.

Parameters

in	<i>namesp</i>	Namespace added to the file
----	---------------	-----------------------------

Returns

void

`std::string os::constantPrinter::capitalize (std::string str) const throw)` [private]

Capitalizes the string argument.

Primarily used for `#ifdef` and `#define` include guards, this function returns the string it is passed but with every single letter capitalized.

Parameters

in	<i>str</i>	String to be capitalized
----	------------	--------------------------

Returns

`std::string` with each letter capitalized

`bool os::constantPrinter::good () const throw)` [inline]

Checks file status.

Checks to ensure that both the `.h` and `.cpp` file can be written to. Will not consider the `.cpp` file if the `.cpp` file is not being written to.

Returns

file status

`bool os::constantPrinter::hasCPP () const throw)` [inline]

Returns if the object is writing to a `.cpp` file.

Returns

constantPrinter::_has_cpp (p. 147)

`void os::constantPrinter::removeNamespace () throw descriptiveException)`

Remove namespace.

Ends the current namespace with a `}` in both the `.h` and `.cpp` file. Decrements **constantPrinter↵::namespaceDepth** (p. 147).

Returns

void

`std::string os::constantPrinter::tabs () const throw () [private]`

Returns current tab depth.

Again used to streamline large projects. This function returns an `std::string` with tab characters equal to the current number of nested namespaces.

Returns

`std::string` containing **`os::constantPrinter::namespaceDepth`** (p. 147) tabs

12.7.4 Member Data Documentation

`bool os::constantPrinter::_has_cpp [private]`

Holds if the object is generating a `.cpp`.

`std::ofstream os::constantPrinter::cppFile [private]`

Output file for the `.cpp` file.

`std::ofstream os::constantPrinter::hFile [private]`

Output file for the `.h` file.

`unsigned os::constantPrinter::namespaceDepth [private]`

Current namespace depth.

In order to streamline large projects, arrays of constants should be placed inside namespaces. This variable allows for the creation and management of nested namespaces.

12.8 `os::constIterator< dataType >` Class Template Reference

Generalized constant iterator.

Public Member Functions

- **`constIterator ()`**
Default constructor.
- **`constIterator (const constIterator< dataType > &cpy)`**
Copy constructor.
- **`constIterator (const iterator< dataType > &cpy)`**
Copy constructor.
- **`virtual ~constIterator ()`**
Virtual destructor.
- **`bool operator! () const throw ()`**
Inverted boolean conversion.
- **`operator bool () const throw ()`**
Boolean conversion.

- const **smart_ptr**< dataType > **constGet** () const throw (descriptiveException)
Member access.
- const **smart_ptr**< dataType > **get** () const throw (descriptiveException)
Member access.
- const **smart_ptr**< dataType > **operator&** () const throw (descriptiveException)
Member access.
- const **smart_ptr**< dataType > **operator->** () const throw (descriptiveException)
Member access.
- const dataType & **operator*** () const throw (descriptiveException)
De-reference.
- **constiterator**< dataType > **constAccess** (long offset) const throw (descriptiveException)
Access iterator by index.
- **constiterator**< dataType > **access** (long offset) const throw (descriptiveException)
Access iterator by index.
- const dataType & **operator[]** (size_t offset) const throw (descriptiveException)
- const dataType & **operator[]** (long offset) const throw (descriptiveException)
- const dataType & **operator[]** (int offset) const throw (descriptiveException)
- **constiterator**< dataType > **operator+** (size_t offset) const throw (descriptiveException)
- **constiterator**< dataType > **operator-** (size_t offset) const throw (descriptiveException)
- **constiterator**< dataType > **operator+** (long offset) const throw (descriptiveException)
- **constiterator**< dataType > **operator-** (long offset) const throw (descriptiveException)
- **constiterator**< dataType > **operator+** (int offset) const throw (descriptiveException)
- **constiterator**< dataType > **operator-** (int offset) const throw (descriptiveException)
- const **constiterator**< dataType > & **increment** (long offset=1) const
Increment this iterator Increments this iterator by some value, by default, 1. Note that if incrementing by more than one, the node in question must support random access.
- const **constiterator**< dataType > & **decrement** (long offset=1) const
Decrement this iterator Decrements this iterator by some value, by default, 1. Note that if decrementing by more than one, the node in question must support random access.
- const **constiterator**< dataType > & **operator+=** (size_t offset) const
- const **constiterator**< dataType > & **operator-=** (size_t offset) const
- const **constiterator**< dataType > & **operator+=** (long offset) const
- const **constiterator**< dataType > & **operator-=** (long offset) const
- const **constiterator**< dataType > & **operator+=** (int offset) const
- const **constiterator**< dataType > & **operator-=** (int offset) const
- **constiterator**< dataType > **operator++** (int param) const throw (descriptiveException)
- **constiterator**< dataType > **operator--** (int param) const throw (descriptiveException)
- const **constiterator**< dataType > & **operator++** () const throw (descriptiveException)
- const **constiterator**< dataType > & **operator--** () const throw (descriptiveException)
- bool **iterable** () const
Returns if the source is iterable.
- bool **randomAccess** () const
Returns if the source can be accessed randomly.
- int **compare** (const **constiterator**< dataType > &cmp) const

Compares two const iterators Uses the comparison definition of the node in an iterator to compare two iterators.

- **operator size_t () const**

Casts to size_t for hash functions Uses the size definition of the current node.

- **COMPARE_OPERATORS int compare (const iterator< dataType > &cmp) const**

*Compares a **constiterator** (p. 147) and an iterator Uses the comparison definition of the node in an iterator to compare two iterators.*

Private Member Functions

- **constiterator (const smart_ptr< nodeFrame< dataType > > node, const iteratorBase< dataType > &src)**

Construct with a frame and source.

Private Attributes

- **const iteratorBase< dataType > * source_structure**

Pointer to the iterator source.

- **smart_ptr< nodeFrame< dataType > > currentNode**

Pointer to the current node.

Friends

- **class iteratorBase< dataType >**

***iteratorBase** (p. 196) must have access to constructor*

- **class iterator< dataType >**

Two iterator types are friends.

12.8.1 Detailed Description

```
template<class dataType>
class os::constIterator< dataType >
```

Generalized constant iterator.

Since this iterator wraps the node frame, it can be generally used for any data-structure in this library. Note that this iterator is strictly immutable

12.8.2 Constructor & Destructor Documentation

```
template<class dataType> os::constIterator< dataType >::constIterator ( const smart_ptr<  
nodeFrame< dataType > > node, const iteratorBase< dataType > & src ) [inline], [private]
```

Construct with a frame and source.

Uses a reference to both a frame and an **iteratorBase** (p. 196) to construct an iterator which can be generally used.

```
template<class dataType> os::constliterator< dataType >::constliterator ( ) [inline]
```

Default constructor.

Constructs an empty iterator.

```
template<class dataType> os::constliterator< dataType >::constliterator ( const constliterator<
dataType > & cpy ) [inline]
```

Copy constructor.

Copies an **constliterator** (p. 147) into this iterator. This copy has a minimal performance penalty and retains the deletion protection of the original iterator.

Parameters

in	<i>cpy</i>	Immutable iterator
----	------------	--------------------

```
template<class dataType> os::constliterator< dataType >::constliterator ( const iterator<
dataType > & cpy ) [inline]
```

Copy constructor.

Copies an iterator into this **constliterator** (p. 147). This copy has a minimal performance penalty, and adds new data protection to the iterator data.

Parameters

in	<i>cpy</i>	Immutable iterator
----	------------	--------------------

```
template<class dataType> virtual os::constliterator< dataType >::~constliterator ( ) [inline],
[virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.8.3 Member Function Documentation

```
template<class dataType> constliterator<dataType> os::constliterator< dataType >::access (
long offset ) const throw descriptiveException() [inline]
```

Access iterator by index.

Access an iterator offset from the current iterator by some value. If an iterator cannot be randomly accessed, an exception will be thrown.

Parameters

in	<i>offset</i>	Value to offset by
----	---------------	--------------------

Returns

Offset iterator, immutable

```
template<class dataType> int os::constiterator< dataType >::compare ( const constiterator<
dataType > & cmp ) const [inline]
```

Compares two const iterators Uses the comparison definition of the node in an iterator to compare two iterators.

Returns

0 if equal, 1 if greater than, -1 is less than

```
template<class dataType> COMPARE_OPERATORS int os::constiterator< dataType >::compare
( const iterator< dataType > & cmp ) const
```

Compares a **constiterator** (p. 147) and an iterator Uses the comparison definition of the node in an iterator to compare two iterators.

Returns

0 if equal, 1 if greater than, -1 is less than

```
template<class dataType> constiterator<dataType> os::constiterator< dataType >::constAccess
( long offset ) const throw descriptiveException() [inline]
```

Access iterator by index.

Access an iterator offset from the current iterator by some value. If an iterator cannot be randomly accessed, an exception will be thrown.

Parameters

in	<i>offset</i>	Value to offset by
-----------	---------------	--------------------

Returns

Offset iterator, immutable

```
template<class dataType> const smart_ptr<dataType> os::constiterator< dataType >::constGet (
) const throw descriptiveException() [inline]
```

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

```
template<class dataType> const constiterator<dataType>& os::constiterator< dataType  
>::decrement ( long offset = 1 ) const [inline]
```

Decrement this iterator Decrements this iterator by some value, by default, 1. Note that if decrementing by more than one, the node in question must support random access.

Parameters

in	<i>offset</i>	Value to offset by
-----------	---------------	--------------------

Returns

This iterator decremented

```
template<class dataType> const smart_ptr<dataType> os::constiterator< dataType >::get ( )  
const throw descriptiveException() [inline]
```

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

```
template<class dataType> const constiterator<dataType>& os::constiterator< dataType  
>::increment ( long offset = 1 ) const [inline]
```

Increment this iterator Increments this iterator by some value, by default, 1. Note that if incrementing by more than one, the node in question must support random access.

Parameters

in	<i>offset</i>	Value to offset by
-----------	---------------	--------------------

Returns

This iterator incremented

```
template<class dataType> bool os::constiterator< dataType >::iterable ( ) const [inline]
```

Returns if the source is iterable.

Returns

source_structure->**iterable()** (p. 152)

```
template<class dataType> os::constliterator< dataType >::operator bool ( ) const throw )  
[inline]
```

Boolean conversion.

Returns

currentNode

```
template<class dataType> os::constliterator< dataType >::operator size_t ( ) const [inline]
```

Casts to size_t for hash functions Uses the size definition of the current node.

Returns

hash value

```
template<class dataType> bool os::constliterator< dataType >::operator! ( ) const throw )  
[inline]
```

Inverted boolean conversion.

Returns

!currentNode

```
template<class dataType> const smart_ptr<dataType> os::constliterator< dataType >::operator&  
( ) const throw descriptiveException) [inline]
```

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

```
template<class dataType> const dataType& os::constliterator< dataType >::operator* ( ) const  
throw descriptiveException) [inline]
```

De-reference.

Returns a const reference to the contained object, the reference cannot be modified.

Returns

Contained object

```
template<class dataType> constliterator<dataType> os::constliterator< dataType >::operator+ (   
size_t offset ) const throw descriptiveException) [inline]
```

```
template<class dataType> constliterator<dataType> os::constliterator< dataType >::operator+ (   
long offset ) const throw descriptiveException) [inline]
```

```
template<class dataType> constliterator<dataType> os::constliterator< dataType >::operator+ (   
int offset ) const throw descriptiveException) [inline]
```

```

template<class dataType> constliterator<dataType> os::constliterator< dataType >::operator++ (
int param ) const throw descriptiveException() [inline]

template<class dataType> const constliterator<dataType>& os::constliterator< dataType
>::operator++ ( ) const throw descriptiveException() [inline]

template<class dataType> const constliterator<dataType>& os::constliterator< dataType
>::operator+= ( size_t offset ) const [inline]

template<class dataType> const constliterator<dataType>& os::constliterator< dataType
>::operator+= ( long offset ) const [inline]

template<class dataType> const constliterator<dataType>& os::constliterator< dataType
>::operator+= ( int offset ) const [inline]

template<class dataType> constliterator<dataType> os::constliterator< dataType >::operator- (
size_t offset ) const throw descriptiveException() [inline]

template<class dataType> constliterator<dataType> os::constliterator< dataType >::operator- (
long offset ) const throw descriptiveException() [inline]

template<class dataType> constliterator<dataType> os::constliterator< dataType >::operator- (
int offset ) const throw descriptiveException() [inline]

template<class dataType> constliterator<dataType> os::constliterator< dataType >::operator-- (
int param ) const throw descriptiveException() [inline]

template<class dataType> const constliterator<dataType>& os::constliterator< dataType
>::operator-- ( ) const throw descriptiveException() [inline]

template<class dataType> const constliterator<dataType>& os::constliterator< dataType
>::operator-= ( size_t offset ) const [inline]

template<class dataType> const constliterator<dataType>& os::constliterator< dataType
>::operator-= ( long offset ) const [inline]

template<class dataType> const constliterator<dataType>& os::constliterator< dataType
>::operator-= ( int offset ) const [inline]

template<class dataType> const smart_ptr<dataType> os::constliterator< dataType >::operator->
( ) const throw descriptiveException() [inline]

```

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

```

template<class dataType> const dataType& os::constliterator< dataType >::operator[] ( size_t
offset ) const throw descriptiveException() [inline]

template<class dataType> const dataType& os::constliterator< dataType >::operator[] ( long
offset ) const throw descriptiveException() [inline]

```

```
template<class dataType> const dataType& os::constliterator< dataType >::operator[] ( int offset
) const throw descriptiveException() [inline]
```

```
template<class dataType> bool os::constliterator< dataType >::randomAccess ( ) const
[inline]
```

Returns if the source can be accessed randomly.

Returns

source_structure->**randomAccess**() (p. 155)

12.8.4 Friends And Related Function Documentation

```
template<class dataType> friend class iterator< dataType > [friend]
```

Two iterator types are friends.

```
template<class dataType> friend class iteratorBase< dataType > [friend]
```

iteratorBase (p. 196) must have access to constructor

12.8.5 Member Data Documentation

```
template<class dataType> smart_ptr<nodeFrame<dataType> > os::constliterator< dataType
>::currentNode [mutable], [private]
```

Pointer to the current node.

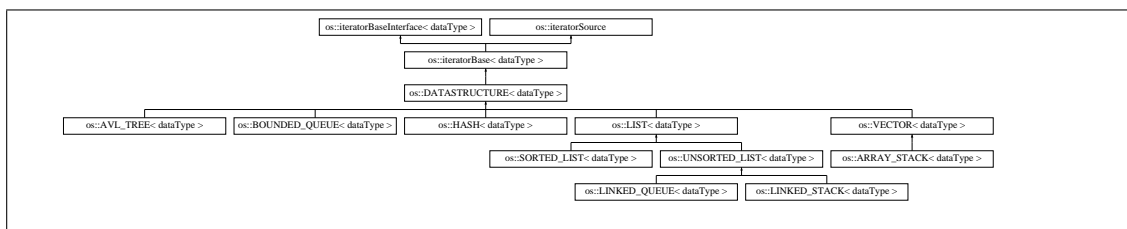
```
template<class dataType> const iteratorBase<dataType>* os::constliterator< dataType
>::source_structure [private]
```

Pointer to the iterator source.

12.9 os::DATASTRUCTURE< dataType > Class Template Reference

Object node definition.

Inheritance diagram for os::DATASTRUCTURE< dataType >:



Public Member Functions

- **DATASTRUCTURE** ()
Default constructor.
- virtual **~DATASTRUCTURE** ()
Virtual destructor.
- virtual bool **insert** (const dataType &x)=0
Insert item into the data-structure.
- bool **insertStructure** (CURRENT_CLASS &x)
Insert data-structure into the data-structure.
- virtual bool **remove** (const dataType &x)=0
Remove item from the data-structure.
- virtual bool **find** (const dataType &x) const =0
Searches for an item.
- virtual dataType & **access** (const dataType &x)=0
Mutable item access.
- virtual const dataType & **access** (const dataType &x) const =0
Immutable item access.
- virtual dataType & **at** (size_t i) throw (descriptiveException)
Access the data-structure by index.
- virtual const dataType & **at** (size_t i) const throw (descriptiveException)
Access the vector by data-structure.
- dataType & **operator[]** (size_t i) throw (descriptiveException)
Access the data-structure by index.
- const dataType & **operator[]** (size_t i) const throw (descriptiveException)
Access the data-structure by index.
- virtual size_t **size** () const =0
Access size of the structure.

Additional Inherited Members

12.9.1 Detailed Description

```
template<class dataType>
class os::DATASTRUCTURE< dataType >
```

Object node definition.

Note that this class is defined in three forms: objectDatastructure pointerDatastructure and raw↔ PointerDatastructure. In short, this class and it's alternate forms specify the details of the **nodeFrame** (p. 233), allowing for holding both objects and pointers to object in the various provided datastructures.

12.9.2 Constructor & Destructor Documentation

```
template<class dataType > os::DATASTRUCTURE< dataType >::DATASTRUCTURE ( )
[inline]
```

Default constructor.

```
template<class dataType > virtual os::DATASTRUCTURE< dataType >::~DATASTRUCTURE (
) [inline], [virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.9.3 Member Function Documentation

```
template<class dataType > virtual dataType& os::DATASTRUCTURE< dataType >::access (
const dataType & x ) [pure virtual]
```

Mutable item access.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Parameters

in	x	Value to be accessed
----	---	----------------------

Returns

Mutable item equal to x

Implemented in **os::AVL_TREE**< **dataType** > (p. 120), **os::LIST**< **dataType** > (p. 214), **os::BOUNDED_QUEUE**< **dataType** > (p. 131), **os::VECTOR**< **dataType** > (p. 280), and **os::HASH**< **dataType** > (p. 173).

```
template<class dataType > virtual const dataType& os::DATASTRUCTURE< dataType >::access (
const dataType & x ) const [pure virtual]
```

Immutable item access.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Parameters

in	x	Value to be accessed
----	---	----------------------

Returns

Immutable item equal to x

Implemented in **os::AVL_TREE**< **dataType** > (p. 120), **os::LIST**< **dataType** > (p. 214), **os::BOUNDED_QUEUE**< **dataType** > (p. 132), **os::VECTOR**< **dataType** > (p. 280), and **os::HASH**< **dataType** > (p. 173).

```
template<class dataType > virtual dataType& os::DATASTRUCTURE< dataType >::at ( size_t i )
throw descriptiveException() [inline], [virtual]
```

Access the data-structure by index.

Parameters

in	<i>i</i>	Index of the data-structure
----	----------	-----------------------------

Returns

Reference to ith

Reimplemented in **os::AVL_TREE**< **dataType** > (p. 121), **os::BOUNDED_QUEUE**< **dataType** > (p. 132), and **os::VECTOR**< **dataType** > (p. 280).

```
template<class dataType > virtual const dataType& os::DATASTRUCTURE< dataType >::at (
size_t i ) const throw descriptiveException() [inline], [virtual]
```

Access the vector by data-structure.

Parameters

in	<i>i</i>	Index of the data-structure
----	----------	-----------------------------

Returns

Immutable reference to ith

Reimplemented in **os::AVL_TREE**< **dataType** > (p. 121), **os::BOUNDED_QUEUE**< **dataType** > (p. 132), and **os::VECTOR**< **dataType** > (p. 281).

```
template<class dataType > virtual bool os::DATASTRUCTURE< dataType >::find ( const
dataType & x ) const [pure virtual]
```

Searches for an item.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references.

Parameters

in	<i>x</i>	Value to be found
----	----------	-------------------

Returns

True if found, else, false

Implemented in **os::AVL_TREE**< **dataType** > (p. 123), **os::LIST**< **dataType** > (p. 214), **os::BOUNDED_QUEUE**< **dataType** > (p. 133), **os::VECTOR**< **dataType** > (p. 281), and **os::HASH**< **dataType** > (p. 174).

```
template<class dataType > virtual bool os::DATASTRUCTURE< dataType >::insert ( const
dataType & x ) [pure virtual]
```

Insert item into the data-structure.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references.

Parameters

in	x	Value to be inserted
----	---	----------------------

Returns

True if inserted, else, false

Implemented in **os::AVL_TREE**< dataType > (p. 124), **os::SORTED_LIST**< dataType > (p. 270), **os::UNSORTED_LIST**< dataType > (p. 277), **os::BOUNDED_QUEUE**< dataType > (p. 134), **os::HASH**< dataType > (p. 175), and **os::VECTOR**< dataType > (p. 282).

```
template<class dataType > bool os::DATASTRUCTURE< dataType >::insertStructure (
CURRENT_CLASS & x ) [inline]
```

Insert data-structure into the data-structure.

Note that this function relies off of iteration being defined for a data-structure.

Parameters

in	x	Structure to be inserted
----	---	--------------------------

Returns

True if inserted, else, false

```
template<class dataType > dataType& os::DATASTRUCTURE< dataType >::operator[] ( size_t i
) throw descriptiveException) [inline]
```

Access the data-structure by index.

Parameters

in	i	Index of the data-structure
----	---	-----------------------------

Returns

Reference to ith element

```
template<class dataType > const dataType& os::DATASTRUCTURE< dataType >::operator[] (
size_t i ) const throw descriptiveException() [inline]
```

Access the data-structure by index.

Parameters

in	<i>i</i>	Index of the data-structure
----	----------	-----------------------------

Returns

Immutable reference to ith element

```
template<class dataType > virtual bool os::DATASTRUCTURE< dataType >::remove ( const
dataType & x ) [pure virtual]
```

Remove item from the data-structure.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references.

Parameters

in	<i>x</i>	Value to be removed
----	----------	---------------------

Returns

True if removed, else, false

Implemented in **os::AVL_TREE**< **dataType** > (p. 125), **os::LIST**< **dataType** > (p. 216), **os::BOUNDED_QUEUE**< **dataType** > (p. 135), **os::HASH**< **dataType** > (p. 175), and **os::VECTOR**< **dataType** > (p. 283).

```
template<class dataType > virtual size_t os::DATASTRUCTURE< dataType >::size ( ) const
[pure virtual]
```

Access size of the structure.

Returns

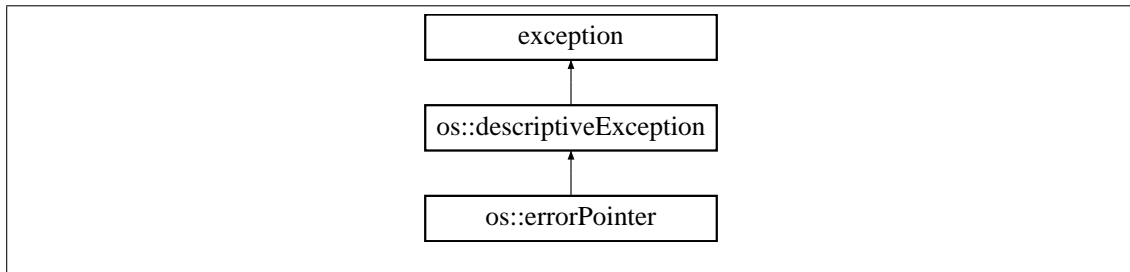
Number of elements in the structure

Implemented in **os::AVL_TREE**< **dataType** > (p. 126), **os::BOUNDED_QUEUE**< **dataType** > (p. 136), **os::VECTOR**< **dataType** > (p. 283), **os::LIST**< **dataType** > (p. 216), and **os::HASH**< **dataType** > (p. 176).

12.10 os::descriptiveException Class Reference

Basic exception with description.

Inheritance diagram for os::descriptiveException:



Public Member Functions

- **descriptiveException** () throw ()
Default constructor.
- **descriptiveException** (const **descriptiveException** &de) throw ()
Copy constructor.
- **descriptiveException** (const char *str) throw ()
Constructor with characters.
- **descriptiveException** (const std::string &str) throw ()
Constructor with string.
- virtual ~**descriptiveException** () throw ()
Destructor Cannot throw exception, as this is an exception and nothing will be there to catch any exception that this throws.
- const char * **what** () const throw ()
Return exception description.

Private Attributes

- const char * **_desc**
Character description.
- std::string **_str**
String description.

12.10.1 Detailed Description

Basic exception with description.

Allows for simple exception with either a string of constant character pointer.

12.10.2 Constructor & Destructor Documentation

os::descriptiveException::descriptiveException () throw) [inline]

Default constructor.

os::descriptiveException::descriptiveException (const **descriptiveException** & de) throw)
[inline]

Copy constructor.

os::descriptiveException::descriptiveException (const char * str) throw) [inline]

Constructor with characters.

Parameters

<i>str</i>	[in] Description
------------	------------------

os::descriptiveException::descriptiveException (const std::string & str) throw) [inline]

Constructor with string.

Parameters

<i>str</i>	[in] Description
------------	------------------

virtual os::descriptiveException::~~descriptiveException () throw) [inline], [virtual]

Destructor Cannot throw exception, as this is an exception and nothing will be there to catch any exception that this throws.

12.10.3 Member Function Documentation

const char* os::descriptiveException::what () const throw) [inline]

Return exception description.

Returns

Description

12.10.4 Member Data Documentation

const char* os::descriptiveException::_desc [private]

Character description.

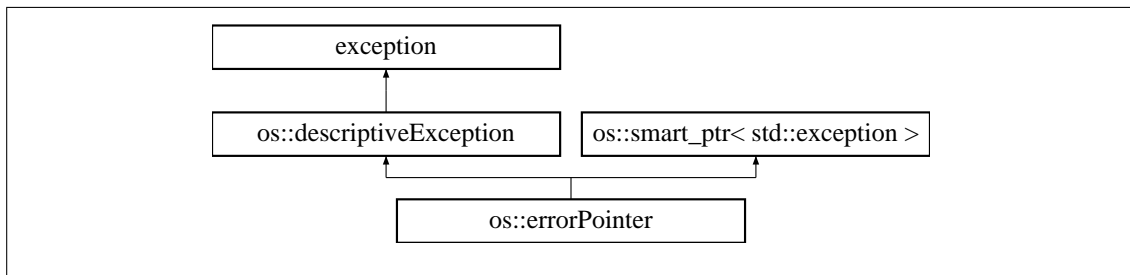
std::string os::descriptiveException::_str [private]

String description.

12.11 os::errorPointer Class Reference

Error pointer class.

Inheritance diagram for os::errorPointer:



Public Member Functions

- **errorPointer** () throw ()
Default constructor.
- **errorPointer** (const **errorPointer** &sp) throw ()
Copy constructor.
- **errorPointer** (const std::exception *rp, **smart_pointer_type** typ=**raw_type**) throw ()
Standard constructor.
- **errorPointer** (const std::exception *rp, const **void_rec** destructor) throw ()
Dynamic deletion constructor.
- virtual ~**errorPointer** () throw ()
Virtual destructor.
- const char * **what** () const final throw ()
***descriptiveException** (p. 161) overload*

12.11.1 Detailed Description

Error pointer class.

Uses the **descriptiveException** (p. 161) interface to allow a pointer to an exception to be handled like an **descriptiveException** (p. 161).

12.11.2 Constructor & Destructor Documentation

`os::errorPointer::errorPointer () throw) [inline]`

Default constructor.

Constructs an **os::smart_ptr** (p. 257) of type **os::null_type** (p. 100). All private data is set to 0 or NULL.

`os::errorPointer::errorPointer (const errorPointer & sp) throw) [inline]`

Copy constructor.

Constructs an **os::errorPointer** (p. 163) from an existing **os::errorPointer** (p. 163).

Parameters

in,out	sp	Reference to data being copied
--------	----	--------------------------------

```
os::errorPointer::errorPointer ( const std::exception * rp, smart_pointer_type typ = raw_type )  
throw ) [inline]
```

Standard constructor.

Constructs an **os::errorPointer** (p. 163) from a raw pointer and a type. This is the most commonly used **os::errorPointer** (p. 163) constructor, other than the copy constructor.

Parameters

in	rp	Raw pointer object is managing
in	typ	Defines reference count behaviour

```
os::errorPointer::errorPointer ( const std::exception * rp, const void_rec destructor ) throw )  
[inline]
```

Dynamic deletion constructor.

Constructs an **os::errorPointer** (p. 163) from a raw pointer and a destruction function.

Parameters

in	rp	Raw pointer object is managing
in	destructor	Defines the function to be executed on destroy

```
virtual os::errorPointer::~~errorPointer ( ) throw ) [inline], [virtual]
```

Virtual destructor.

Calls **os::smart_ptr<dataType>::teardown()** (p. 268) before destroying the object.

12.11.3 Member Function Documentation

```
const char* os::errorPointer::what ( ) const throw ) [inline], [final]
```

descriptiveException (p. 161) overload

Outputs the error or the exception pointer held in this class, if the pointer is NULL, a default warning will be returned.

Returns

character pointer to the error description

12.12 os::eventReceiver< senderType > Class Template Reference

Class which enables event receiving.

Public Member Functions

- **eventReceiver** ()
Default constructor.
- virtual **~eventReceiver** ()
Virtual destructor.
- void **pushSender** (**smart_ptr**< senderType > ptr)
Add a sender to the list.
- void **removeSender** (**smart_ptr**< senderType > ptr)
Remove sender from the sender list.

Private Member Functions

- virtual void **receiveEvent** (**smart_ptr**< senderType > src)
Receive event notification.
- void **receiveEvent** (void *src)
Receive event notification.

Private Attributes

- rawPointerAVLTree< senderType > **senders**
List of sender.
- **threadLock _lock**
Trigger lock Locks a receiver while it is receiving data.

Friends

- template<typename receiverType >
class **eventSender**

12.12.1 Detailed Description

```
template<class senderType>  
class os::eventReceiver< senderType >
```

Class which enables event receiving.

Each receiver contains a list of senders. When the receiver is destroyed, it removes itself from all senders to which it is registered.

12.12.2 Constructor & Destructor Documentation

```
template<class senderType > os::eventReceiver< senderType >::eventReceiver ( ) [inline]
```

Default constructor.

The default constructor for the smart set configures the only data type in this class properly. No additional constructor arguments are required.

```
template<class senderType > virtual os::eventReceiver< senderType >::~~eventReceiver ( )  
[virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.12.3 Member Function Documentation

```
template<class senderType > void os::eventReceiver< senderType >::pushSender ( smart_ptr<  
senderType > ptr )
```

Add a sender to the list.

Adds a sender of the sender type expected by this receiver type. Note that the sender type is expected to inherit from **os::eventSender** (p. 167).

Parameters

<i>ptr</i>	Sender to be added to the set
------------	-------------------------------

Returns

void

```
template<class senderType > virtual void os::eventReceiver< senderType >::receiveEvent ( smart_ptr< senderType > src ) [inline], [private], [virtual]
```

Receive event notification.

This function is meant to be reimplemented by all event receivers to do some action on the event.

Parameters

<i>src</i>	The source of the event
------------	-------------------------

Returns

void

```
template<class senderType > void os::eventReceiver< senderType >::receiveEvent ( void * src )  
[inline], [private]
```

Receive event notification.

Casts a void pointer to the type of the sender.

Parameters

<i>src</i>	The source of the event
------------	-------------------------

Returns

void

```
template<class senderType > void os::eventReceiver< senderType >::removeSender (
smart_ptr< senderType > ptr )
```

Remove sender from the sender list.

Removes a sender from the sender list. Note that this also removes this receiver from the receiver list of the sender which it is passed.

Parameters

<i>ptr</i>	Sender to be removed to the set
------------	---------------------------------

Returns

void

12.12.4 Friends And Related Function Documentation

```
template<class senderType > template<typename receiverType > friend class eventSender
[friend]
```

The sender must be able to remove itself from the private senders list inside the event receiver. Additionally, the sender must be able to send an event to the receiver.

12.12.5 Member Data Documentation

```
template<class senderType > threadLock os::eventReceiver< senderType >::_lock [private]
```

Trigger lock Locks a receiver while it is receiving data.

```
template<class senderType > rawPointerAVLTree<senderType> os::eventReceiver< senderType
>::senders [private]
```

List of sender.

When the receiver is destroyed, this list is used to remove itself from all its senders.

12.13 **os::eventSender**< receiverType > Class Template Reference

Class which enables event sending.

Public Member Functions

- **eventSender ()**
Default constructor.
- virtual **~eventSender ()**
Virtual destructor.
- void **pushReceivers (smart_ptr< receiverType > ptr)**
Add a receiver to the list.
- void **removeReceivers (smart_ptr< receiverType > ptr)**
Remove receiver from the receiver list.

Protected Member Functions

- virtual void **sendEvent (smart_ptr< receiverType > ptr)**
Receive event notification.
- virtual void **sendEvent (smart_ptr< receiverType > ptr, unsigned arg, void *data)**
Receive event notification, flexible data.
- void **triggerEvent ()**
Sends an event to all receivers.
- void **triggerEvent (unsigned arg, void *data)**
Sends an event to all receivers with data.

Private Attributes

- rawPointerAVLTree< receiverType > **receivers**
List of receivers.
- **threadLock _lock**
Trigger lock Locks an event while it is being triggered.

Friends

- template<typename senderType >
class **eventReceiver**
Friendship with the senderType The receiver must be able to remove itself from the private receivers list inside the event sender.

12.13.1 Detailed Description

```
template<class receiverType>
class os::eventSender< receiverType >
```

Class which enables event sending.

Each sender contains a list of receivers. When an event is triggered, the sender iterates through the list to send the event to all receivers.

12.13.2 Constructor & Destructor Documentation

```
template<class receiverType > os::eventSender< receiverType >::eventSender ( ) [inline]
```

Default constructor.

The default constructor for the smart set configures the only data type in this class properly. No additional constructor arguments are required.

```
template<class receiverType > virtual os::eventSender< receiverType >::~~eventSender ( )  
[virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.13.3 Member Function Documentation

```
template<class receiverType > void os::eventSender< receiverType >::pushReceivers ( smart_ptr< receiverType > ptr )
```

Add a receiver to the list.

Adds a receiver of the receiver type expected by this sender type. Note that the receiver type is expected to inherit from **os::eventReceiver** (p. 165).

Parameters

<i>ptr</i>	Receiver to be added to the set
------------	---------------------------------

Returns

void

```
template<class receiverType > void os::eventSender< receiverType >::removeReceivers ( smart_ptr< receiverType > ptr )
```

Remove receiver from the receiver list.

Removes a receiver from the receiver list. Note that this also removes this sender from the sender list of the receiver which it is passed.

Parameters

<i>ptr</i>	Receiver to be removed to the set
------------	-----------------------------------

Returns

void

```
template<class receiverType > virtual void os::eventSender< receiverType >::sendEvent (
smart_ptr< receiverType > ptr ) [protected], [virtual]
```

Receive event notification.

This function can be re-implemented by event senders. This function allows some function other than "receiveEvent" to be sent by the event sender to an event receiver.

Parameters

in	<i>ptr</i>	The target of the event
----	------------	-------------------------

Returns

void

```
template<class receiverType > virtual void os::eventSender< receiverType >::sendEvent (
smart_ptr< receiverType > ptr, unsigned arg, void * data ) [inline], [protected], [virtual]
```

Receive event notification, flexible data.

This function can be re-implemented by event senders. This function allows some function other than "receiveEvent" to be sent by the event sender to an event receiver.

Parameters

in	<i>ptr</i>	The target of the event
in	<i>arg</i>	Event flag
in	<i>data</i>	Void pointer to data

Returns

void

```
template<class receiverType > void os::eventSender< receiverType >::triggerEvent ( )
[protected]
```

Sends an event to all receivers.

Iterates through the set of receivers and sends an event to each one. This calls the **os::eventSender**<**receiverType**>::sendEvent (p. 170) function with each receiver as an argument.

Returns

void

```
template<class receiverType > void os::eventSender< receiverType >::triggerEvent ( unsigned
arg, void * data ) [protected]
```

Sends an event to all receivers with data.

Iterates through the set of receivers and sends an event to each one. This calls the **os::eventSender<receiverType>::sendEvent** (p. 170) function with each receiver as an argument. This event trigger passes data arguments.

Parameters

in	arg	Event flag
in	data	Void pointer to data

Returns

void

12.13.4 Friends And Related Function Documentation

```
template<class receiverType > template<typename senderType > friend class eventReceiver
[friend]
```

Friendship with the senderType The receiver must be able to remove itself from the private receivers list inside the event sender.

12.13.5 Member Data Documentation

```
template<class receiverType > threadLock os::eventSender< receiverType >::_lock [private]
```

Trigger lock Locks an event while it is being triggered.

```
template<class receiverType > rawPointerAVLTree<receiverType> os::eventSender< receiverType >::receivers [private]
```

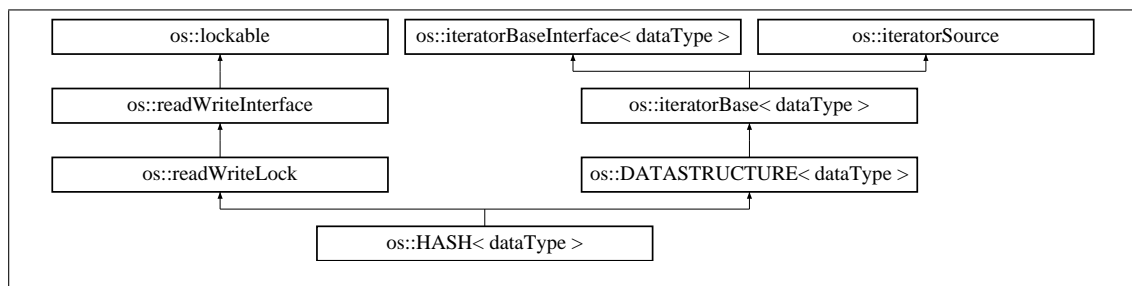
List of receivers.

This list is used to send events to all receivers. When the sender is destroyed, it must remove itself from all its receivers.

12.14 os::HASH< dataType > Class Template Reference

Iterable hash table.

Inheritance diagram for os::HASH< dataType >:



Public Member Functions

- **HASH** ()
Constructs the hash table Uses the correct comparison function construct the hash table.
- virtual **~HASH** ()
Destroys the hash Removes every element from the hash. Depending on the hash format, this may delete everythin in the hash.
- **size_t size** () const final
Access size of the hash.
- **bool iterable** () const final
Returns if the relevant datastructure type is iterable.
- **bool randomAccess** () const final
Returns if the relevant datastructure type can be accessed randomly.
- **bool insert** (const dataType &x) final
Insert item into the hash.
- **bool remove** (const dataType &x) final
Remove item from the hash.
- **bool find** (const dataType &x) const final
Searches for an item in the hash.
- **dataType & access** (const dataType &x) final
Mutable item access.
- **const dataType & access** (const dataType &x) const final
Immutable item access.

Static Public Attributes

- static const bool **ITERABLE** = true
Hashes are iterable.
- static const bool **RANDOM_ACCESS** = false
Hashes do not allow random access.

Protected Member Functions

- **smart_ptr< nodeFrame< dataType > > getFirstNode** () final
Access to first node.
- **smart_ptr< nodeFrame< dataType > > getLastNode** () final
Access to last node.
- const **smart_ptr< nodeFrame< dataType > > getFirstNodeConst** () const final
Constant access to first node.
- const **smart_ptr< nodeFrame< dataType > > getLastNodeConst** () const final
Constant access to last node.
- **smart_ptr< nodeFrame< dataType > > searchNode** (const smart_ptr< dataType > dt) final
Search for a node.
- const **smart_ptr< nodeFrame< dataType > > searchNodeConst** (const smart_ptr< dataType > dt) const final
Const search for a node.

Private Attributes

- **simpleHash**< dataType > **_hashTable**

Hash table This table stores the data inside this particular hash.

Additional Inherited Members

12.14.1 Detailed Description

```
template<class dataType>
class os::HASH< dataType >
```

Iterable hash table.

Uses the structure defined by the **os::simpleHash** (p. 250) to define a hash table which is iterable.

12.14.2 Constructor & Destructor Documentation

```
template<class dataType> os::HASH< dataType >::HASH ( ) [inline]
```

Constructs the hash table Uses the correct comparison function construct the hash table.

```
template<class dataType> virtual os::HASH< dataType >::~~HASH ( ) [inline], [virtual]
```

Destroys the hash Removes every element from the hash. Depending on the hash format, this may delete everything in the hash.

12.14.3 Member Function Documentation

```
template<class dataType> dataType& os::HASH< dataType >::access ( const dataType & x )
[inline], [final], [virtual]
```

Mutable item access.

Accesses an element in the hash, O(1). Note that different forms of the hash accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Returns

Mutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 157).

```
template<class dataType> const dataType& os::HASH< dataType >::access ( const dataType & x
) const [inline], [final], [virtual]
```

Immutable item access.

Accesses an element in the hash, O(1). Note that different forms of the hash accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Returns

Immutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 157).

```
template<class dataType> bool os::HASH< dataType >::find ( const dataType & x ) const  
[inline], [final], [virtual]
```

Searches for an item in the hash.

Finds an element in the hash, O(1). Note that different forms of the hash accept pointers instead of object references.

Returns

True if found, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 158).

```
template<class dataType> smart_ptr<nodeFrame<dataType> > os::HASH< dataType  
>::getFirstNode ( ) [final], [protected], [virtual]
```

Access to first node.

Returns

First node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 198).

```
template<class dataType> const smart_ptr<nodeFrame<dataType> > os::HASH< dataType  
>::getFirstNodeConst ( ) const [final], [protected], [virtual]
```

Constant access to first node.

Returns

Immutable first node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 198).

```
template<class dataType> smart_ptr<nodeFrame<dataType> > os::HASH< dataType  
>::getLastNode ( ) [final], [protected], [virtual]
```

Access to last node.

Returns

Last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 198).

```
template<class dataType> const smart_ptr<nodeFrame<dataType> > os::HASH< dataType  
>::getLastNodeConst ( ) const [final], [protected], [virtual]
```

Constant access to last node.

Returns

Immutable last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 199).

```
template<class dataType> bool os::HASH< dataType >::insert ( const dataType & x ) [inline],  
[final], [virtual]
```

Insert item into the hash.

Inserts an item into hash, O(1). Note that different forms of the hash accept pointers instead of object references.

Returns

True

Implements **os::DATASTRUCTURE**< **dataType** > (p. 159).

```
template<class dataType> bool os::HASH< dataType >::iterable ( ) const [inline], [final],  
[virtual]
```

Returns if the relevant datastructure type is iterable.

Returns

true

Reimplemented from **os::iteratorSource** (p. 205).

```
template<class dataType> bool os::HASH< dataType >::randomAccess ( ) const [inline],  
[final], [virtual]
```

Returns if the relevant datastructure type can be accessed randomly.

Returns

true

Reimplemented from **os::iteratorSource** (p. 205).

```
template<class dataType> bool os::HASH< dataType >::remove ( const dataType & x )  
[inline], [final], [virtual]
```

Remove item from the hash.

Removes an element in the hash, O(1). Note that different forms of the hash accept pointers instead of object references.

Returns

True if removed, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 160).


```
template<class dataType> smart_ptr<nodeFrame<dataType> > os::HASH< dataType  
>::searchNode ( const smart_ptr< dataType > dt ) [final], [protected], [virtual]
```

Search for a node.

Parameters

in	<i>dt</i>	Pointer to search for
-----------	-----------	-----------------------

Returns

Mutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 199).

```
template<class dataType> const smart_ptr<nodeFrame<dataType> > os::HASH< dataType  
>::searchNodeConst ( const smart_ptr< dataType > dt ) const [final], [protected],  
[virtual]
```

Const search for a node.

Parameters

in	<i>dt</i>	Pointer to search for
-----------	-----------	-----------------------

Returns

Immutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 200).

```
template<class dataType> size_t os::HASH< dataType >::size ( ) const [inline], [final],  
[virtual]
```

Access size of the hash.

Returns

Number of elements in the hash

Implements **os::DATASTRUCTURE**< **dataType** > (p. 160).

12.14.4 Member Data Documentation

```
template<class dataType> simpleHash<dataType> os::HASH< dataType >::_hashTable  
[private]
```

Hash table This table stores the data inside this particular hash.

```
template<class dataType> const bool os::HASH< dataType >::ITERABLE = true [static]
```

Hashes are iterable.

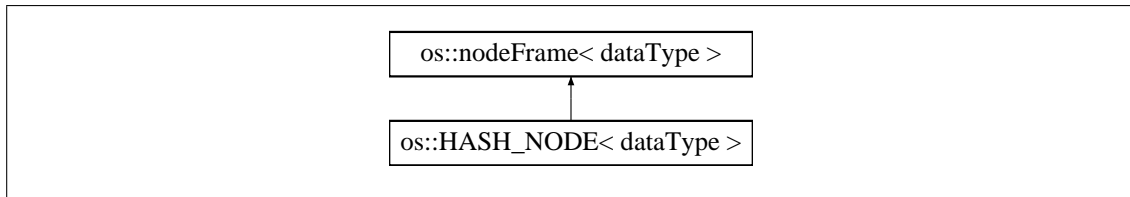
```
template<class dataType> const bool os::HASH< dataType >::RANDOM_ACCESS = false
[static]
```

Hashes do not allow random access.

12.15 os::HASH_NODE< dataType > Class Template Reference

Hash node.

Inheritance diagram for os::HASH_NODE< dataType >:



Public Member Functions

- **~HASH_NODE** ()
Destructor Class is not designed to be inherited from.
- **smart_ptr**< dataType > **get** () final throw ()
Returns a pointer.
- const **smart_ptr**< dataType > **constGet** () const final throw ()
Returns a const pointer.
- dataType & **operator*** () final throw (descriptiveException)
De-reference.
- const dataType & **operator*** () const final throw (descriptiveException)
De-reference.
- bool **valid** () const final
Valid data query Checks if the provided data is valid.
- bool **iterable** () const final
Returns if the node is iterable.
- bool **randomAccess** () const final
Returns if the node can be accessed randomly.
- **smart_ptr**< **nodeFrame**< dataType > > **getNext** () final throw (descriptiveException)
Returns the next hash frame.
- const **smart_ptr**< **nodeFrame**< dataType > > **getNextConst** () const final throw (descriptiveException)
Returns the next hash frame.
- **smart_ptr**< **nodeFrame**< dataType > > **getPrev** () final throw (descriptiveException)
Returns the previous hash frame.
- const **smart_ptr**< **nodeFrame**< dataType > > **getPrevConst** () const final throw (descriptiveException)
Returns the previous hash frame.

Returns the previous hash frame.

- void **remove** () final throw (descriptiveException)
Remove this node from the hash.

Static Public Attributes

- static const bool **ITERABLE** = true
Hash frames are iterable.
- static const bool **RANDOM_ACCESS** = false
Hash frames do not allow random-access.

Private Member Functions

- **HASH_NODE** (HASH< dataType > *src, size_t pos)
*Private constructor This constructor is not designed to be accessed by anything other than the **HASH** (p. 171) class.*

Private Attributes

- size_t **position**
Current position of the hash iterator.

12.15.1 Detailed Description

```
template<class dataType>
class os::HASH_NODE< dataType >
```

Hash node.

Used by the iterator to iterate through a hash.

12.15.2 Constructor & Destructor Documentation

```
template<class dataType > os::HASH_NODE< dataType >::HASH_NODE ( HASH< dataType > *
src, size_t pos ) [inline], [private]
```

Private constructor This constructor is not designed to be accessed by anything other than the **HASH** (p. 171) class.

```
template<class dataType > os::HASH_NODE< dataType >::~HASH_NODE ( ) [inline]
```

Destructor Class is not designed to be inherited from.

12.15.3 Member Function Documentation

```
template<class dataType > const smart_ptr<dataType> os::HASH_NODE< dataType >::constGet
( ) const throw ) [inline], [final], [virtual]
```

Returns a const pointer.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

Implements **os::nodeFrame< dataType >** (p. 236).

```
template<class dataType > smart_ptr<dataType> os::HASH_NODE< dataType >::get ( ) throw (
[inline], [final], [virtual]
```

Returns a pointer.

Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

Implements **os::nodeFrame< dataType >** (p. 237).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::HASH_NODE< dataType
>::getNext ( ) throw descriptiveException) [inline], [final], [virtual]
```

Returns the next hash frame.

Returns

Next node, mutable

Reimplemented from **os::nodeFrame< dataType >** (p. 237).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::HASH_NODE<
dataType >::getNextConst ( ) const throw descriptiveException) [inline], [final],
[virtual]
```

Returns the next hash frame.

Returns

Next node, immutable

Reimplemented from **os::nodeFrame< dataType >** (p. 238).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::HASH_NODE< dataType
>::getPrev ( ) throw descriptiveException) [inline], [final], [virtual]
```

Returns the previous hash frame.

Returns

Previous node, mutable

Reimplemented from **os::nodeFrame< dataType >** (p. 238).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::HASH_NODE<
dataType >::getPrevConst ( ) const throw descriptiveException) [inline], [final],
[virtual]
```

Returns the previous hash frame.

Returns

Previous node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 238).

```
template<class dataType > bool os::HASH_NODE< dataType >::iterable ( ) const [inline],
[final], [virtual]
```

Returns if the node is iterable.

Returns

object::ITERABLE

Reimplemented from **os::nodeFrame**< **dataType** > (p. 239).

```
template<class dataType > dataType& os::HASH_NODE< dataType >::operator* ( ) throw
descriptiveException) [inline], [final], [virtual]
```

De-reference.

Returns a reference to the contained object, the reference can be modified.

Returns

Contained object

Implements **os::nodeFrame**< **dataType** > (p. 239).

```
template<class dataType > const dataType& os::HASH_NODE< dataType >::operator* ( ) const
throw descriptiveException) [inline], [final], [virtual]
```

De-reference.

Returns a const reference to the contained object, the reference cannot be modified.

Returns

Contained object

Implements **os::nodeFrame**< **dataType** > (p. 240).

```
template<class dataType > bool os::HASH_NODE< dataType >::randomAccess ( ) const
[inline], [final], [virtual]
```

Returns if the node can be accessed randomly.

Returns

object::RANDOM_ACCESS

Reimplemented from **os::nodeFrame**< **dataType** > (p. 240).

```
template<class dataType > void os::HASH_NODE< dataType >::remove ( ) throw
descriptiveException [inline], [final], [virtual]
```

Remove this node from the hash.

Returns

void

Reimplemented from **os::nodeFrame**< **dataType** > (p. 240).

```
template<class dataType > bool os::HASH_NODE< dataType >::valid ( ) const [inline],
[final], [virtual]
```

Valid data query Checks if the provided data is valid.

Returns

true if valid, else, false

Reimplemented from **os::nodeFrame**< **dataType** > (p. 241).

12.15.4 Member Data Documentation

```
template<class dataType > const bool os::HASH_NODE< dataType >::ITERABLE = true
[static]
```

Hash frames are iterable.

```
template<class dataType > size_t os::HASH_NODE< dataType >::position [private]
```

Current position of the hash iterator.

```
template<class dataType > const bool os::HASH_NODE< dataType >::RANDOM_ACCESS = false
[static]
```

Hash frames do not allow random-access.

12.16 **os::indirectMatrix**< dataType > Class Template Reference

Indirect matrix.

Public Member Functions

- **indirectMatrix** (uint32_t w=0, uint32_t h=0) throw ()
Default constructor.
- **indirectMatrix** (const **matrix**< dataType > &m) throw ()
Copy constructor.
- **indirectMatrix** (const **indirectMatrix**< dataType > &m) throw ()
Copy constructor.

- **indirectMatrix** (const **smart_ptr**< dataType > d, uint32_t w, uint32_t h) throw ()
Data array constructor.
- **indirectMatrix** (**smart_ptr**< **smart_ptr**< dataType > > d, uint32_t w, uint32_t h) throw ()
Indirect data array constructor.
- virtual ~**indirectMatrix** () throw (std::exception)
Virtual destructor.
- **indirectMatrix**< dataType > & **operator=** (const **matrix**< dataType > &m) throw ()
Equality constructor.
- **indirectMatrix**< dataType > & **operator=** (const **indirectMatrix**< dataType > &m) throw ()
Equality constructor.
- **smart_ptr**< dataType > & **get** (uint32_t w, uint32_t h) throw ()
Return pointer to a matrix element.
- const **smart_ptr**< dataType > & **constGet** (uint32_t w, uint32_t h) const throw ()
Return constant pointer to a matrix element.
- **smart_ptr**< dataType > & **operator()** (uint32_t w, uint32_t h) throw ()
Return pointer to a matrix element.
- **smart_ptr**< **smart_ptr**< dataType > > **getArray** () throw ()
Return pointer to the pointer array.
- const **smart_ptr**< **smart_ptr**< dataType > > **getConstArray** () const throw ()
Return a constant pointer to the pointer array.
- uint32_t **width** () const throw ()
Return _width of matrix.
- uint32_t **height** () const throw ()
Return _height of matrix.

Private Attributes

- uint32_t **_width**
Width of the matrix.
- uint32_t **_height**
Height of the matrix.
- **smart_ptr**< **smart_ptr**< dataType > > **data**
Data array pointers.

Friends

- class **matrix**< **dataType** >
Raw matrix interacting with indirect matrix.

12.16.1 Detailed Description

```
template<class dataType>
class os::indirectMatrix< dataType >
```

Indirect matrix.

This matrix class contains an array to pointers of the data type. It can interact with os::matrix<data←Type>.

12.16.2 Constructor & Destructor Documentation

```
template<class dataType> os::indirectMatrix< dataType >::indirectMatrix ( uint32_t w = 0,  
uint32_t h = 0 ) throw )
```

Default constructor.

Constructs array of size w*h and sets all of the data to 0. If no `_width` and `_height` are provided, the data array is not initialized.

Parameters

in	<i>w</i>	Width of matrix, default 0
in	<i>h</i>	Height of matrix, default 0

```
template<class dataType> os::indirectMatrix< dataType >::indirectMatrix ( const matrix<  
dataType > & m ) throw )
```

Copy constructor.

Constructs a new indirect matrix from the given raw matrix. The indirect matrix converts the array of object to an array of pointers.

Parameters

in	<i>m</i>	Indirect matrix to be copied
----	----------	------------------------------

```
template<class dataType> os::indirectMatrix< dataType >::indirectMatrix ( const  
indirectMatrix< dataType > & m ) throw )
```

Copy constructor.

Constructs a new indirect matrix from the given indirect matrix. The two indirect matrices do not share data array, the new indirect matrix builds its own array.

Parameters

in	<i>m</i>	Indirect matrix to be copied
----	----------	------------------------------

```
template<class dataType> os::indirectMatrix< dataType >::indirectMatrix ( const smart_ptr<  
dataType > d, uint32_t w, uint32_t h ) throw )
```

Data array constructor.

Constructs a new indirect matrix from an array of the correct data type. This constructor will build an new indirect array based on the specified size.

Parameters

in	<i>d</i>	Data array to be copied
----	----------	-------------------------

Parameters

in	<i>w</i>	Width of matrix
in	<i>d</i>	Height of matrix

```
template<class dataType> os::indirectMatrix< dataType >::indirectMatrix ( smart_ptr<
smart_ptr< dataType > > d, uint32_t w, uint32_t h ) throw )
```

Indirect data array constructor.

Constructs a new indirect matrix from an indirect array of the correct data type. This constructor will build an new indirect array based on the specified size.

Parameters

in	<i>d</i>	Indirect data array to be copied
in	<i>w</i>	Width of matrix
in	<i>d</i>	Height of matrix

```
template<class dataType> virtual os::indirectMatrix< dataType >::~indirectMatrix ( ) throw
std::exception) [inline], [virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.16.3 Member Function Documentation

```
template<class dataType> const smart_ptr<dataType>& os::indirectMatrix< dataType
>::constGet ( uint32_t w, uint32_t h ) const throw )
```

Return constant pointer to a matrix element.

Uses a `_width` and `_height` position to index an element of the array. This function returns a constant reference, meaning changes cannot be made to the matrix.

Parameters

in	<i>w</i>	X position
in	<i>h</i>	Y position

Returns

Constant reference to matrix element pointer

```
template<class dataType> smart_ptr<dataType>& os::indirectMatrix< dataType >::get (
uint32_t w, uint32_t h ) throw )
```

Return pointer to a matrix element.

Uses a `_width` and `_height` position to index an element of the array. This function returns a reference, allowing for changes to be made to the matrix.

Parameters

in	w	X position
in	h	Y position

Returns

Modifiable reference to matrix element pointer

```
template<class dataType> smart_ptr<smart_ptr<dataType> > os::indirectMatrix< dataType
>::getArray ( ) throw ) [inline]
```

Return pointer to the pointer array.

The array which is returned allows for modification of the array. It is up to functions using this array to ensure the integrity of the indirect matrix.

Returns

os::indirectMatrix<dataType>::data (p. 187)

```
template<class dataType> const smart_ptr<smart_ptr<dataType> > os::indirectMatrix<
dataType>::getConstArray ( ) const throw ) [inline]
```

Return a constant pointer to the pointer array.

The array which is returned allows for access to the array. The provided array may not be modified.

Returns

os::indirectMatrix<dataType>::data (p. 187)

```
template<class dataType> uint32_t os::indirectMatrix< dataType >::height ( ) const throw )
[inline]
```

Return `_height` of matrix.

Returns

indirectMatrix<dataType>::_height (p. 187)

```
template<class dataType> smart_ptr<dataType>& os::indirectMatrix< dataType >::operator() (
uint32_t w, uint32_t h ) throw ) [inline]
```

Return pointer to a matrix element.

Uses a `_width` and `_height` position to index an element of the array. This function returns a reference, allowing for changes to be made to the matrix.

Parameters

in	<i>w</i>	X position
in	<i>h</i>	Y position

Returns

Modifiable reference to matrix element pointer

```
template<class dataType> indirectMatrix<dataType>& os::indirectMatrix< dataType >::operator=
( const matrix< dataType > & m ) throw )
```

Equality constructor.

Re-constructs the indirect matrix from a raw matrix. Note that the two matrices do not share the same data array.

Parameters

in	<i>m</i>	Reference to matrix being copied
-----------	----------	----------------------------------

Returns

Reference to self

```
template<class dataType> indirectMatrix<dataType>& os::indirectMatrix< dataType >::operator=
( const indirectMatrix< dataType > & m ) throw )
```

Equality constructor.

Re-constructs the indirect matrix from another indirect matrix. Note that the two matrices do not share the same data array.

Parameters

in	<i>m</i>	Reference to matrix being copied
-----------	----------	----------------------------------

Returns

Reference to self

```
template<class dataType> uint32_t os::indirectMatrix< dataType >::width ( ) const throw )  
[inline]
```

Return _width of matrix.

Returns

indirectMatrix<dataType>::_width (p. 187)

12.16.4 Friends And Related Function Documentation

```
template<class dataType> friend class matrix< dataType > [friend]
```

Raw matrix interacting with indirect matrix.

The `os::matrix<dataType>` class must be able to access the size and data of the indirect matrix because and raw matrix can be constructed from an indirect matrix.

12.16.5 Member Data Documentation

```
template<class dataType> uint32_t os::indirectMatrix< dataType >::_height [private]
```

Height of the matrix.

```
template<class dataType> uint32_t os::indirectMatrix< dataType >::_width [private]
```

Width of the matrix.

```
template<class dataType> smart_ptr<smart_ptr<dataType> > os::indirectMatrix< dataType  
>::data [private]
```

Data array pointers.

For the indirect matrix class, this array contains pointers to all of the data used by the matrix in a block of size `_width*_height`.

12.17 `os::iterator< dataType >` Class Template Reference

Generalized iterator.

Public Member Functions

- **iterator** ()
Default constructor.
- **iterator** (const **iterator**< dataType > &cpy)
Copy constructor.
- virtual ~**iterator** ()
Virtual destructor.

- **bool operator! ()** const throw ()
Inverted boolean conversion.
- **operator bool ()** const throw ()
Boolean conversion.
- **smart_ptr< dataType > get ()** throw (descriptiveException)
Member access.
- const **smart_ptr< dataType > constGet ()** const throw (descriptiveException)
Member access.
- const **smart_ptr< dataType > get ()** const throw (descriptiveException)
Member access.
- **smart_ptr< dataType > operator& ()** throw (descriptiveException)
Member access.
- const **smart_ptr< dataType > operator& ()** const throw (descriptiveException)
Member access.
- **smart_ptr< dataType > operator-> ()** throw (descriptiveException)
Member access.
- const **smart_ptr< dataType > operator-> ()** const throw (descriptiveException)
Member access.
- **dataType & operator* ()** throw (descriptiveException)
De-reference.
- const **dataType & operator* ()** const throw (descriptiveException)
De-reference.
- void **remove ()**
Remove node from the list.
- **iterator< dataType > access (long offset)** const throw (descriptiveException)
Access iterator by index.
- **dataType & operator[] (size_t offset)** throw (descriptiveException)
- **dataType & operator[] (long offset)** throw (descriptiveException)
- **dataType & operator[] (int offset)** throw (descriptiveException)
- const **dataType & operator[] (size_t offset)** const throw (descriptiveException)
- const **dataType & operator[] (long offset)** const throw (descriptiveException)
- const **dataType & operator[] (int offset)** const throw (descriptiveException)
- **iterator< dataType > operator+ (size_t offset)** const throw (descriptiveException)
- **iterator< dataType > operator- (size_t offset)** const throw (descriptiveException)
- **iterator< dataType > operator+ (long offset)** const throw (descriptiveException)
- **iterator< dataType > operator- (long offset)** const throw (descriptiveException)
- **iterator< dataType > operator+ (int offset)** const throw (descriptiveException)
- **iterator< dataType > operator- (int offset)** const throw (descriptiveException)
- const **iterator< dataType > & increment (long offset=1)** const
Increment this iterator Increments this iterator by some value, by default, 1. Note that if incrementing by more than one, the node in question must support random access.
- const **iterator< dataType > & decrement (long offset=1)** const
Decrement this iterator Decrements this iterator by some value, by default, 1. Note that if decrementing by more than one, the node in question must support random access.

- **const iterator**< dataType > & **operator+=** (size_t offset) const
- **const iterator**< dataType > & **operator-=** (size_t offset) const
- **const iterator**< dataType > & **operator+=** (long offset) const
- **const iterator**< dataType > & **operator-=** (long offset) const
- **const iterator**< dataType > & **operator+=** (int offset) const
- **const iterator**< dataType > & **operator-=** (int offset) const
- **iterator**< dataType > **operator++** (int param) throw (descriptiveException)
- **iterator**< dataType > **operator--** (int param) throw (descriptiveException)
- **const iterator**< dataType > & **operator++** () const throw (descriptiveException)
- **const iterator**< dataType > & **operator--** () const throw (descriptiveException)
- **bool iterable** () const
Returns if the source is iterable.
- **bool randomAccess** () const
Returns if the source can be accessed randomly.
- **int compare** (const **iterator**< dataType > &cmp) const
Compares two iterators Uses the comparison definition of the node in an iterator to compare two iterators.
- **operator size_t** () const
Casts to size_t for hash functions Uses the size definition of the current node.
- **COMPARE_OPERATORS int compare** (const **constiterator**< dataType > &cmp) const
*Compares an iterator and a **constiterator** (p. 147) Uses the comparison definition of the node in an iterator to compare two iterators.*

Private Member Functions

- **iterator** (const **smart_ptr**< **nodeFrame**< dataType > > node, const **iteratorBase**< dataType > &src)
Construct with a frame and source.

Private Attributes

- **const iteratorBase**< dataType > * **source_structure**
Pointer to the iterator source.
- **smart_ptr**< **nodeFrame**< dataType > > **currentNode**
Pointer to the current node.

Friends

- **class iteratorBase**< dataType >
***iteratorBase** (p. 196) must have access to constructor*
- **class constiterator**< dataType >
Two iterator types are friends.

12.17.1 Detailed Description

```
template<class dataType>
class os::iterator< dataType >
```

Generalized iterator.

Since this iterator wraps the node frame, it can be generally used for any data-structure in this library.

12.17.2 Constructor & Destructor Documentation

```
template<class dataType> os::iterator< dataType >::iterator ( const smart_ptr< nodeFrame<
dataType > > node, const iteratorBase< dataType > & src ) [inline], [private]
```

Construct with a frame and source.

Uses a reference to both a frame and an **iteratorBase** (p. 196) to construct an iterator which can be generally used.

```
template<class dataType> os::iterator< dataType >::iterator ( ) [inline]
```

Default constructor.

Constructs an empty iterator.

```
template<class dataType> os::iterator< dataType >::iterator ( const iterator< dataType > & cpy )
[inline]
```

Copy constructor.

Copies an iterator into this iterator. This copy has a minimal performance penalty and retains the deletion protection of the original iterator.

Parameters

in	cpy	Immutable iterator
----	-----	--------------------

```
template<class dataType> virtual os::iterator< dataType >::~~iterator ( ) [inline], [virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.17.3 Member Function Documentation

```
template<class dataType> iterator<dataType> os::iterator< dataType >::access ( long offset )
const throw descriptiveException ) [inline]
```

Access iterator by index.

Access an iterator offset from the current node by some value. If an iterator cannot be randomly accessed, an exception will be thrown.

Returns

Offset iterator, mutable

```
template<class dataType> int os::iterator< dataType >::compare ( const iterator< dataType > &
cmp ) const [inline]
```

Compares two iterators Uses the comparison definition of the node in an iterator to compare two iterators.

Returns

0 if equal, 1 if greater than, -1 is less than

```
template<class dataType> COMPARE_OPERATORS int os::iterator< dataType >::compare (
const constiterator< dataType > & cmp ) const
```

Compares an iterator and a **constiterator** (p. 147) Uses the comparison definition of the node in an iterator to compare two iterators.

Returns

0 if equal, 1 if greater than, -1 is less than

```
template<class dataType> const smart_ptr<dataType> os::iterator< dataType >::constGet ( )
const throw descriptiveException) [inline]
```

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

```
template<class dataType> const iterator<dataType>& os::iterator< dataType >::decrement ( long
offset = 1 ) const [inline]
```

Decrement this iterator Decrements this iterator by some value, by default, 1. Note that if decrementing by more than one, the node in question must support random access.

Parameters

in	offset	Value to offset by
----	--------	--------------------

Returns

This iterator decremented

```
template<class dataType> smart_ptr<dataType> os::iterator< dataType >::get ( ) throw
descriptiveException) [inline]
```

Member access.

Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

```
template<class dataType> const smart_ptr<dataType> os::iterator< dataType >::get ( ) const  
throw descriptiveException() [inline]
```

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

```
template<class dataType> const iterator<dataType>& os::iterator< dataType >::increment ( long  
offset = 1 ) const [inline]
```

Increment this iterator Increments this iterator by some value, by default, 1. Note that if incrementing by more than one, the node in question must support random access.

Parameters

in	<i>offset</i>	Value to offset by
-----------	---------------	--------------------

Returns

This iterator incremented

```
template<class dataType> bool os::iterator< dataType >::iterable ( ) const [inline]
```

Returns if the source is iterable.

Returns

source_structure->**iterable()** (p. 192)

```
template<class dataType> os::iterator< dataType >::operator bool ( ) const throw ) [inline]
```

Boolean conversion.

Returns

currentNode

```
template<class dataType> os::iterator< dataType >::operator size_t ( ) const [inline]
```

Casts to size_t for hash functions Uses the size definition of the current node.

Returns

hash value

```
template<class dataType> bool os::iterator< dataType >::operator! ( ) const throw ) [inline]
```

Inverted boolean conversion.

Returns

!currentNode

```
template<class dataType> smart_ptr<dataType> os::iterator< dataType >::operator& ( ) throw  
descriptiveException) [inline]
```

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

```
template<class dataType> const smart_ptr<dataType> os::iterator< dataType >::operator& ( )  
const throw descriptiveException) [inline]
```

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

```
template<class dataType> dataType& os::iterator< dataType >::operator* ( ) throw  
descriptiveException) [inline]
```

De-reference.

Returns a reference to the contained object, the reference can be modified.

Returns

Contained object

```
template<class dataType> const dataType& os::iterator< dataType >::operator* ( ) const throw  
descriptiveException) [inline]
```

De-reference.

Returns a const reference to the contained object, the reference cannot be modified.

Returns

Contained object

```
template<class dataType> iterator<dataType> os::iterator< dataType >::operator+ ( size_t offset  
) const throw descriptiveException) [inline]
```

```
template<class dataType> iterator<dataType> os::iterator< dataType >::operator+ ( long offset )  
const throw descriptiveException) [inline]
```

```

template<class dataType> iterator<dataType> os::iterator< dataType >::operator+ ( int offset )
const throw descriptiveException) [inline]

template<class dataType> iterator<dataType> os::iterator< dataType >::operator++ ( int param )
throw descriptiveException) [inline]

template<class dataType> const iterator<dataType>& os::iterator< dataType >::operator++ ( )
const throw descriptiveException) [inline]

template<class dataType> const iterator<dataType>& os::iterator< dataType >::operator+= (
size_t offset ) const [inline]

template<class dataType> const iterator<dataType>& os::iterator< dataType >::operator+= (
long offset ) const [inline]

template<class dataType> const iterator<dataType>& os::iterator< dataType >::operator+= ( int
offset ) const [inline]

template<class dataType> iterator<dataType> os::iterator< dataType >::operator- ( size_t offset
) const throw descriptiveException) [inline]

template<class dataType> iterator<dataType> os::iterator< dataType >::operator- ( long offset )
const throw descriptiveException) [inline]

template<class dataType> iterator<dataType> os::iterator< dataType >::operator- ( int offset )
const throw descriptiveException) [inline]

template<class dataType> iterator<dataType> os::iterator< dataType >::operator-- ( int param )
throw descriptiveException) [inline]

template<class dataType> const iterator<dataType>& os::iterator< dataType >::operator-- ( )
const throw descriptiveException) [inline]

template<class dataType> const iterator<dataType>& os::iterator< dataType >::operator-= (
size_t offset ) const [inline]

template<class dataType> const iterator<dataType>& os::iterator< dataType >::operator-= ( long
offset ) const [inline]

template<class dataType> const iterator<dataType>& os::iterator< dataType >::operator-= ( int
offset ) const [inline]

template<class dataType> smart_ptr<dataType> os::iterator< dataType >::operator-> ( ) throw
descriptiveException) [inline]

```

Member access.

Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

```

template<class dataType> const smart_ptr<dataType> os::iterator< dataType >::operator-> ( )
const throw descriptiveException) [inline]

```

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

```
template<class dataType> dataType& os::iterator< dataType >::operator[] ( size_t offset ) throw descriptiveException() [inline]
```

```
template<class dataType> dataType& os::iterator< dataType >::operator[] ( long offset ) throw descriptiveException() [inline]
```

```
template<class dataType> dataType& os::iterator< dataType >::operator[] ( int offset ) throw descriptiveException() [inline]
```

```
template<class dataType> const dataType& os::iterator< dataType >::operator[] ( size_t offset ) const throw descriptiveException() [inline]
```

```
template<class dataType> const dataType& os::iterator< dataType >::operator[] ( long offset ) const throw descriptiveException() [inline]
```

```
template<class dataType> const dataType& os::iterator< dataType >::operator[] ( int offset ) const throw descriptiveException() [inline]
```

```
template<class dataType> bool os::iterator< dataType >::randomAccess ( ) const [inline]
```

Returns if the source can be accessed randomly.

Returns

source_structure->**randomAccess**() (p. 195)

```
template<class dataType> void os::iterator< dataType >::remove ( ) [inline]
```

Remove node from the list.

Returns

void

12.17.4 Friends And Related Function Documentation

```
template<class dataType> friend class constIterator< dataType > [friend]
```

Two iterator types are friends.

```
template<class dataType> friend class iteratorBase< dataType > [friend]
```

iteratorBase (p. 196) must have access to constructor

12.17.5 Member Data Documentation

```
template<class dataType> smart_ptr<nodeFrame<dataType> > os::iterator< dataType >::currentNode [mutable], [private]
```

Pointer to the current node.

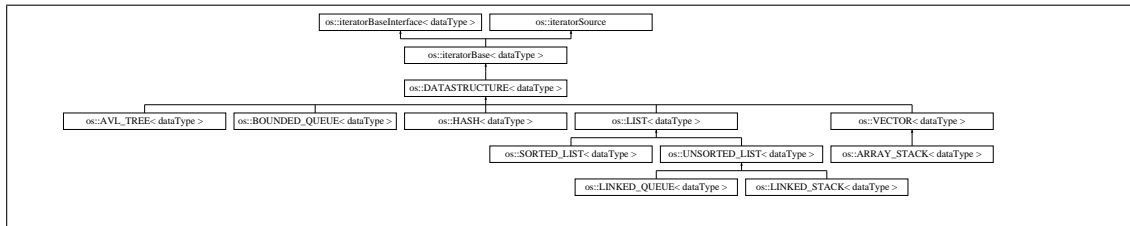
```
template<class dataType> const iteratorBase<dataType>* os::iterator< dataType  
>::source_structure [private]
```

Pointer to the iterator source.

12.18 os::iteratorBase< dataType > Class Template Reference

Iterator source.

Inheritance diagram for os::iteratorBase< dataType >:



Public Member Functions

- **iterator**< dataType > **first** () final
Get first iterator.
- **iterator**< dataType > **last** () final
Get last iterator.
- **constiterator**< dataType > **constFirst** () const final
Get first iterator.
- **constiterator**< dataType > **constLast** () const final
Get last iterator.
- **iterator**< dataType > **search** (const dataType &dt) final
Search for a node.
- **constiterator**< dataType > **constSearch** (const dataType &dt) const final
Serach for a node.

Protected Member Functions

- virtual **smart_ptr**< **nodeFrame**< dataType > > **getFirstNode** ()=0
Access to first node.
- virtual **smart_ptr**< **nodeFrame**< dataType > > **getLastNode** ()=0
Access to last node.
- virtual const **smart_ptr**< **nodeFrame**< dataType > > **getFirstNodeConst** () const =0
Constant access to first node.
- virtual const **smart_ptr**< **nodeFrame**< dataType > > **getLastNodeConst** () const =0
Constant access to last node.
- virtual **smart_ptr**< **nodeFrame**< dataType > > **searchNode** (const **smart_ptr**< dataType > dt)=0

Search for a node.

- virtual const **smart_ptr**< **nodeFrame**< dataType > > **searchNodeConst** (const **smart_ptr**< dataType > dt) const =0

Const search for a node.

Friends

- class **iterator**< **dataType** >
Iterator must have access to counter.
- class **constiterator**< **dataType** >
***constiterator** (p. 147) must have access to counter*

Additional Inherited Members

12.18.1 Detailed Description

```
template<class dataType>
class os::iteratorBase< dataType >
```

Iterator source.

Defines a class template for an object which generates an iterator.

12.18.2 Member Function Documentation

```
template<class dataType> constiterator<dataType> os::iteratorBase< dataType >::constFirst (
) const [final], [virtual]
```

Get first iterator.

Returns

Immutable first element

Implements **os::iteratorBaseInterface**< **dataType** > (p. 201).

```
template<class dataType> constiterator<dataType> os::iteratorBase< dataType >::constLast (
) const [final], [virtual]
```

Get last iterator.

Returns

Immutable last element

Implements **os::iteratorBaseInterface**< **dataType** > (p. 201).

```
template<class dataType> constiterator<dataType> os::iteratorBase< dataType >::constSearch
( const dataType & dt ) const [final], [virtual]
```

Search for a node.

Parameters

in	dt	Node to search for
----	----	--------------------

Returns

Immutable element

Implements **os::iteratorBaseInterface< dataType >** (p. 202).

```
template<class dataType> iterator<dataType> os::iteratorBase< dataType >::first ( ) [final],  
[virtual]
```

Get first iterator.

Returns

Mutable first element

Implements **os::iteratorBaseInterface< dataType >** (p. 202).

```
template<class dataType> virtual smart_ptr<nodeFrame<dataType> > os::iteratorBase<  
dataType >::getFirstNode ( ) [protected], [pure virtual]
```

Access to first node.

Returns

First node in the structure

Implemented in **os::AVL_TREE< dataType >** (p. 123), **os::LIST< dataType >** (p. 214), **os::BOUNDED_QUEUE< dataType >** (p. 133), **os::VECTOR< dataType >** (p. 281), and **os::HASH< dataType >** (p. 174).

```
template<class dataType> virtual const smart_ptr<nodeFrame<dataType> > os::iteratorBase<  
dataType >::getFirstNodeConst ( ) const [protected], [pure virtual]
```

Constant access to first node.

Returns

Immutable first node in the structure

Implemented in **os::AVL_TREE< dataType >** (p. 123), **os::LIST< dataType >** (p. 215), **os::BOUNDED_QUEUE< dataType >** (p. 134), **os::VECTOR< dataType >** (p. 281), and **os::HASH< dataType >** (p. 174).

```
template<class dataType> virtual smart_ptr<nodeFrame<dataType> > os::iteratorBase<  
dataType >::getLastNode ( ) [protected], [pure virtual]
```

Access to last node.

Returns

Last node in the structure

Implemented in **os::AVL_TREE< dataType >** (p. 124), **os::LIST< dataType >** (p. 215), **os::BOUNDED_QUEUE< dataType >** (p. 134), **os::VECTOR< dataType >** (p. 282), and **os::HASH< dataType >** (p. 174).

```
template<class dataType> virtual const smart_ptr<nodeFrame<dataType> > os::iteratorBase<
dataType>::getLastNodeConst ( ) const [protected], [pure virtual]
```

Constant access to last node.

Returns

Immutable last node in the structure

Implemented in **os::AVL_TREE< dataType >** (p. 124), **os::LIST< dataType >** (p. 215), **os::BOUNDED_QUEUE< dataType >** (p. 134), **os::VECTOR< dataType >** (p. 282), and **os::HASH< dataType >** (p. 175).

```
template<class dataType> iterator<dataType> os::iteratorBase< dataType >::last ( ) [final],
[virtual]
```

Get last iterator.

Returns

Mutable last element

Implements **os::iteratorBaseInterface< dataType >** (p. 202).

```
template<class dataType> iterator<dataType> os::iteratorBase< dataType >::search ( const
dataType & dt ) [final], [virtual]
```

Search for a node.

Returns

Mutable element

Implements **os::iteratorBaseInterface< dataType >** (p. 203).

```
template<class dataType> virtual smart_ptr<nodeFrame<dataType> > os::iteratorBase<
dataType>::searchNode ( const smart_ptr< dataType > dt ) [protected], [pure virtual]
```

Search for a node.

Parameters

<i>Pointer</i>	to search for
----------------	---------------

Returns

Mutable found node, if applicable

Implemented in **os::AVL_TREE**< **dataType** > (p. 125), **os::BOUNDED_QUEUE**< **dataType** > (p. 135), **os::VECTOR**< **dataType** > (p. 283), **os::LIST**< **dataType** > (p. 216), and **os::HASH**< **dataType** > (p. 176).

```
template<class dataType> virtual const smart_ptr<nodeFrame<dataType> > os::iteratorBase<
dataType >::searchNodeConst ( const smart_ptr< dataType > dt ) const [protected], [pure
virtual]
```

Const search for a node.

Parameters

<i>Pointer</i>	to search for
----------------	---------------

Returns

Immutable found node, if applicable

Implemented in **os::AVL_TREE**< **dataType** > (p. 125), **os::BOUNDED_QUEUE**< **dataType** > (p. 136), **os::VECTOR**< **dataType** > (p. 283), **os::LIST**< **dataType** > (p. 216), and **os::HASH**< **dataType** > (p. 176).

12.18.3 Friends And Related Function Documentation

```
template<class dataType> friend class constiterator< dataType > [friend]
```

constiterator (p. 147) must have access to counter

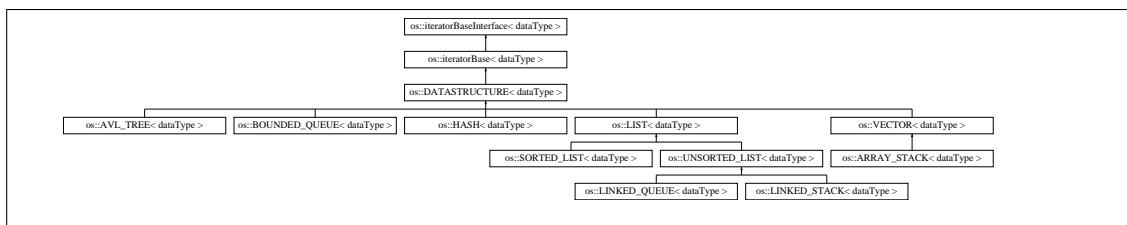
```
template<class dataType> friend class iterator< dataType > [friend]
```

Iterator must have access to counter.

12.19 os::iteratorBaseInterface< dataType > Class Template Reference

Iterator base interface.

Inheritance diagram for os::iteratorBaseInterface< dataType >:



Public Member Functions

- virtual **iterator**< dataType > **first** ()=0
Get first iterator.
- virtual **iterator**< dataType > **last** ()=0
Get last iterator.
- virtual **constiterator**< dataType > **constFirst** () const =0
Get first iterator.
- virtual **constiterator**< dataType > **constLast** () const =0
Get last iterator.
- **constiterator**< dataType > **first** () const
Get first iterator.
- **constiterator**< dataType > **last** () const
Get last iterator.
- virtual **iterator**< dataType > **search** (const dataType &dt)=0
Search for a node.
- virtual **constiterator**< dataType > **constSearch** (const dataType &dt) const =0
Search for a node.
- **constiterator**< dataType > **search** (const dataType &dt) const
Search for a node.

12.19.1 Detailed Description

```
template<class dataType>
class os::iteratorBaseInterface< dataType >
```

Iterator base interface.

This interface is used to access iterators. It is designed to be used by the **os::iteratorBase** (p. 196), however, classes which provide iterator access may want to extend this interface.

12.19.2 Member Function Documentation

```
template<class dataType > virtual constiterator<dataType> os::iteratorBaseInterface< dataType >::constFirst ( ) const [pure virtual]
```

Get first iterator.

Returns

Immutable first element

Implemented in **os::iteratorBase**< **dataType** > (p. 197).

```
template<class dataType > virtual constiterator<dataType> os::iteratorBaseInterface< dataType >::constLast ( ) const [pure virtual]
```

Get last iterator.

Returns

Immutable last element

Implemented in **os::iteratorBase**< **dataType** > (p. 197).

```
template<class dataType > virtual constIterator<dataType> os::iteratorBaseInterface< dataType  
>::constSearch ( const dataType & dt ) const [pure virtual]
```

Search for a node.

Parameters

in	<i>dt</i>	Node to search for
-----------	-----------	--------------------

Returns

Immutable element

Implemented in **os::iteratorBase**< **dataType** > (p. 197).

```
template<class dataType > virtual iterator<dataType> os::iteratorBaseInterface< dataType  
>::first ( ) [pure virtual]
```

Get first iterator.

Returns

Mutable first element

Implemented in **os::iteratorBase**< **dataType** > (p. 198).

```
template<class dataType > constIterator<dataType> os::iteratorBaseInterface< dataType >::first  
( ) const [inline]
```

Get first iterator.

Returns

Immutable first element

```
template<class dataType > virtual iterator<dataType> os::iteratorBaseInterface< dataType  
>::last ( ) [pure virtual]
```

Get last iterator.

Returns

Mutable last element

Implemented in **os::iteratorBase**< **dataType** > (p. 199).

```
template<class dataType > constIterator<dataType> os::iteratorBaseInterface< dataType >::last  
( ) const [inline]
```

Get last iterator.

Returns

Immutable last element

```
template<class dataType > virtual iterator<dataType> os::iteratorBaseInterface< dataType  
>::search ( const dataType & dt ) [pure virtual]
```

Search for a node.

Parameters

in	<i>dt</i>	Node to search for
-----------	-----------	--------------------

Returns

Mutable element

Implemented in **os::iteratorBase**< **dataType** > (p. 199).

```
template<class dataType > constiterator<dataType> os::iteratorBaseInterface< dataType  
>::search ( const dataType & dt ) const [inline]
```

Search for a node.

Parameters

in	<i>dt</i>	Node to search for
-----------	-----------	--------------------

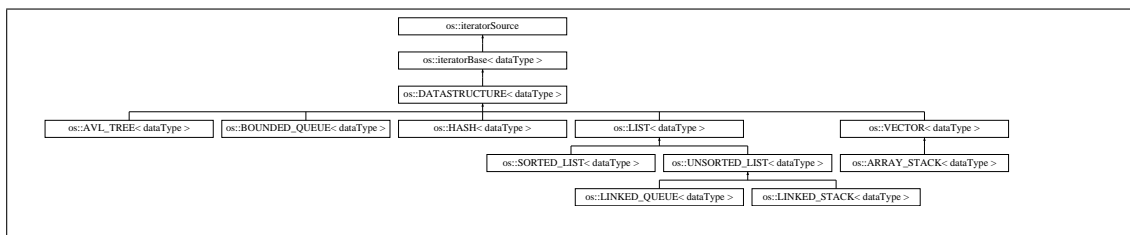
Returns

Immutable element

12.20 os::iteratorSource Class Reference

Base class for iterator source.

Inheritance diagram for os::iteratorSource:



Public Types

- enum **tearDown** { **IGNORE_OVERFLOW** =0, **THROW_ERROR**, **WAIT** }
Defines behavior on teardown.

Public Member Functions

- **iteratorSource** ()
Default constructor.
- virtual **~iteratorSource** () throw (descriptiveException)
Virtual destructor.
- unsigned **outstandingIterator** () const
Access outstanding iterator counter.
- void **setTearDown** (tearDown td=IGNORE_OVERFLOW)
Set tear-down behavior.
- **tearDown** getTearDown () const
Access tear-down behavior.
- virtual bool **iterable** () const
Returns if the source is iterable.
- virtual bool **randomAccess** () const
Returns if the source can be accessed randomly.

Static Public Attributes

- static const bool **ITERABLE** = false
By default, an iterated source is not iterable.
- static const bool **RANDOM_ACCESS** = false
No random-access by default.

Protected Attributes

- std::atomic< unsigned > **_outstandingIterator**
Outbound iterator count.
- **tearDown_tearDown**
Defines the tear-down behavior.

12.20.1 Detailed Description

Base class for iterator source.

Provides an atomic iterator counter in-order to track the number of dispatched iterators.

12.20.2 Member Enumeration Documentation

enum **os::iteratorSource::tearDown**

Defines behavior on teardown.

Enumerator

- IGNORE_OVERFLOW** Ignore outstanding iterators.
- THROW_ERROR** Throw error for outstanding iterators.
- WAIT** Wait for the destruction of outstanding iterators.

12.20.3 Constructor & Destructor Documentation

`os::iteratorSource::iteratorSource () [inline]`

Default constructor.

Sets the iterator count to 0 and initializes the standard tear-down behaviour. Constructor is protected because this class has no meaning if not extended.

`virtual os::iteratorSource::~iteratorSource () throw descriptiveException [inline],
[virtual]`

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called. This particular destructor has behaviour defined by the value of `_tearDown`.

12.20.4 Member Function Documentation

tearDown `os::iteratorSource::getTearDown () const [inline]`

Access tear-down behavior.

Returns

iteratorSource::_tearDown (p. 206)

`virtual bool os::iteratorSource::iterable () const [inline], [virtual]`

Returns if the source is iterable.

Returns

`object::ITERABLE`

Reimplemented in **os::AVL_TREE< dataType >** (p. 124), **os::BOUNDED_QUEUE< dataType >** (p. 134), **os::VECTOR< dataType >** (p. 282), **os::LIST< dataType >** (p. 215), and **os::HASH< dataType >** (p. 175).

`unsigned os::iteratorSource::outstandingIterator () const [inline]`

Access outstanding iterator counter.

Returns

iteratorSource::_outstandingIterator (p. 206)

`virtual bool os::iteratorSource::randomAccess () const [inline], [virtual]`

Returns if the source can be accessed randomly.

Returns

`object::RANDOM_ACCESS`

Reimplemented in **os::AVL_TREE< dataType >** (p. 124), **os::BOUNDED_QUEUE< dataType >** (p. 135), **os::VECTOR< dataType >** (p. 282), **os::LIST< dataType >** (p. 215), and **os::HASH< dataType >** (p. 175).

void os::iteratorSource::setTearDown (**tearDown** td = IGNORE_OVERFLOW) [inline]

Set tear-down behavior.

Parameters

in	td	New tear-down value
----	----	---------------------

Returns

void

12.20.5 Member Data Documentation

std::atomic<unsigned> os::iteratorSource::_outstandingIterator [mutable], [protected]

Outbound iterator count.

Holds the number of iterators which have been sent out into the larger world.

tearDown os::iteratorSource::_tearDown [protected]

Defines the tear-down behavior.

const bool os::iteratorSource::ITERABLE = false [static]

By default, an iterated source is not iterable.

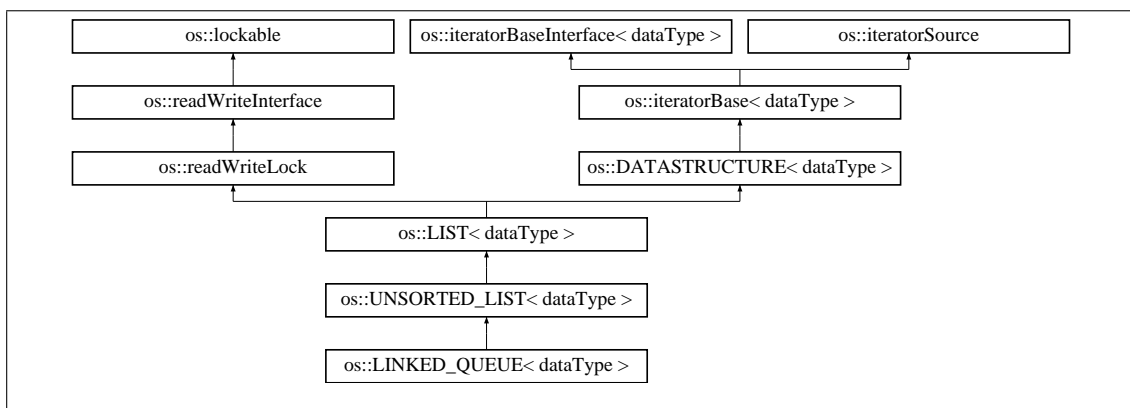
const bool os::iteratorSource::RANDOM_ACCESS = false [static]

No random-access by default.

12.21 os::LINKED_QUEUE< dataType > Class Template Reference

Queue built on-top of a list.

Inheritance diagram for os::LINKED_QUEUE< dataType >:



Public Member Functions

- **LINKED_QUEUE ()**
Default constructor Simple constructor which implicitly call the constructor of it's base.
- virtual **~LINKED_QUEUE ()**
Destroys the stack Uses the destructor function defined by the unsorted list.
- void **pop ()**
Remove top element.
- bool **push (const dataType &x)**
Add element.
- const dataType & **constTop () const**
Const access top element.
- dataType & **top ()**
Access top element.
- const dataType & **top () const**
Const access top element.
- bool **push (smart_ptr< dataType > x)**
- const **smart_ptr< dataType > constTop () const**
- **smart_ptr< dataType > top ()**
- const **smart_ptr< dataType > top () const**

Additional Inherited Members

12.21.1 Detailed Description

```
template<class dataType>
class os::LINKED_QUEUE< dataType >
```

Queue built on-top of a list.

Insertion occurs and the end, removal from the beginning. Note that this class retains all the functionality of it's parent, the unsorted linked list.

12.21.2 Constructor & Destructor Documentation

```
template<class dataType > os::LINKED_QUEUE< dataType >::LINKED_QUEUE ( ) [inline]
```

Default constructor Simple constructor which implicitly call the constructor of it's base.

```
template<class dataType > virtual os::LINKED_QUEUE< dataType >::~~LINKED_QUEUE ( )
[inline], [virtual]
```

Destroys the stack Uses the destructor function defined by the unsorted list.

12.21.3 Member Function Documentation

```
template<class dataType > const dataType& os::LINKED_QUEUE< dataType >::constTop ( )  
const [inline]
```

Const access top element.

Returns the top element of the queue. Note that this operation will throw an exception if the queue is empty.

Returns

Immutable top element

```
template<class dataType > const smart_ptr<dataType> os::LINKED_QUEUE< dataType  
>::constTop ( ) const [inline]
```

```
template<class dataType > void os::LINKED_QUEUE< dataType >::pop ( ) [inline]
```

Remove top element.

Attempts to remove the top element. Throws an exception if the stack is empty.

Returns

void

```
template<class dataType > bool os::LINKED_QUEUE< dataType >::push ( const dataType & x )  
[inline]
```

Add element.

Inserts an element at the front of the queue

Parameters

in	x	Element to be inserted
----	---	------------------------

Returns

true

```
template<class dataType > bool os::LINKED_QUEUE< dataType >::push ( smart_ptr< dataType  
> x ) [inline]
```

```
template<class dataType > dataType& os::LINKED_QUEUE< dataType >::top ( ) [inline]
```

Access top element.

Returns the top element of the queue. Note that this operation will throw an exception if the queue is empty.

Returns

Mutable top element

```
template<class dataType > const dataType& os::LINKED_QUEUE< dataType >::top ( ) const
[inline]
```

Const access top element.

Returns the top element of the queue. Note that this operation will throw an exception if the queue is empty.

Returns

Immutable top element

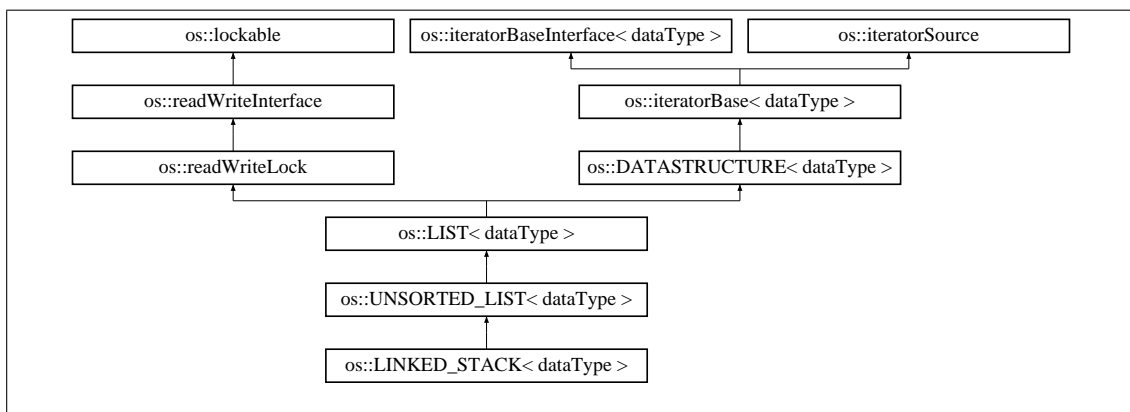
```
template<class dataType > smart_ptr<dataType> os::LINKED_QUEUE< dataType >::top ( )
[inline]
```

```
template<class dataType > const smart_ptr<dataType> os::LINKED_QUEUE< dataType >::top (
) const [inline]
```

12.22 os::LINKED_STACK< dataType > Class Template Reference

Stack built on-top of a list.

Inheritance diagram for os::LINKED_STACK< dataType >:



Public Member Functions

- **LINKED_STACK** ()

Default constructor Simple constructor which implicitly call the constructor of it's base.

- virtual **~LINKED_STACK** ()

Destroys the stack Uses the destructor function defined by the unsorted list.

- void **pop** ()

Remove top element.

- bool **push** (const dataType &x)

Add element.

- const dataType & **constTop** () const

Const access top element.

- `dataType & top ()`
Access top element.
- `const dataType & top () const`
Const access top element.
- `bool push (smart_ptr< dataType > x)`
- `const smart_ptr< dataType > constTop () const`
- `smart_ptr< dataType > top ()`
- `const smart_ptr< dataType > top () const`

Additional Inherited Members

12.22.1 Detailed Description

```
template<class dataType>
class os::LINKED_STACK< dataType >
```

Stack built on-top of a list.

Insertion occurs and the top, removal from the top. Note that this class retains all the functionality of it's parent, the unsorted linked list.

12.22.2 Constructor & Destructor Documentation

```
template<class dataType > os::LINKED_STACK< dataType >::LINKED_STACK ( ) [inline]
```

Default constructor Simple constructor which implicitly call the constructor of it's base.

```
template<class dataType > virtual os::LINKED_STACK< dataType >::~LINKED_STACK ( )
[inline], [virtual]
```

Destroys the stack Uses the destructor function defined by the unsorted list.

12.22.3 Member Function Documentation

```
template<class dataType > const dataType& os::LINKED_STACK< dataType >::constTop ( )
const [inline]
```

Const access top element.

Returns the top element of the stack. Note that this operation will throw an exception if the stack is empty.

Returns

Immutable top element

```
template<class dataType > const smart_ptr<dataType> os::LINKED_STACK< dataType
>::constTop ( ) const [inline]
```

```
template<class dataType > void os::LINKED_STACK< dataType >::pop ( ) [inline]
```

Remove top element.

Attempts to remove the top element. Throws an exception if the stack is empty.

Returns

void

```
template<class dataType > bool os::LINKED_STACK< dataType >::push ( const dataType & x )  
[inline]
```

Add element.

Inserts an element at the top of the stack

Parameters

in	x	Element to be inserted
----	---	------------------------

Returns

true

```
template<class dataType > bool os::LINKED_STACK< dataType >::push ( smart_ptr< dataType  
> x ) [inline]
```

```
template<class dataType > dataType& os::LINKED_STACK< dataType >::top ( ) [inline]
```

Access top element.

Returns the top element of the stack. Note that this operation will throw an exception if the stack is empty.

Returns

Mutable top element

```
template<class dataType > const dataType& os::LINKED_STACK< dataType >::top ( ) const  
[inline]
```

Const access top element.

Returns the top element of the stack. Note that this operation will throw an exception if the stack is empty.

Returns

Immutable top element

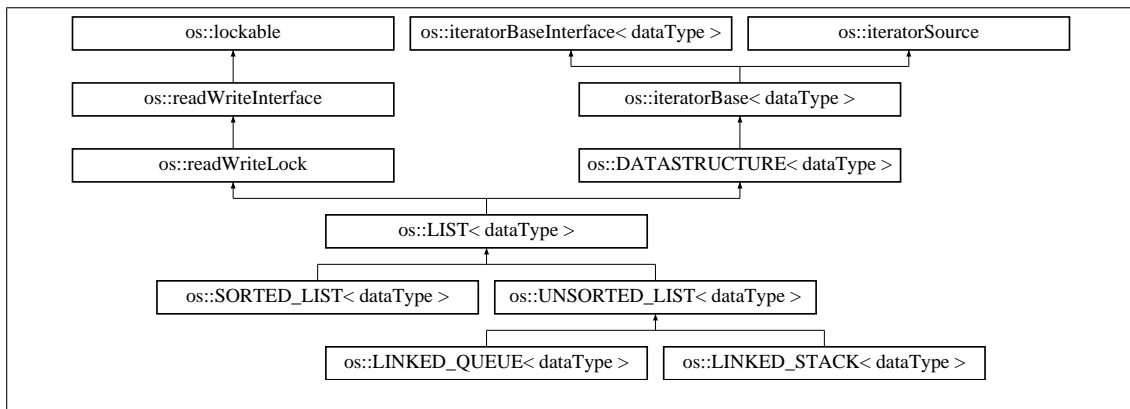
```
template<class dataType > smart_ptr<dataType> os::LINKED_STACK< dataType >::top ( )  
[inline]
```

```
template<class dataType > const smart_ptr<dataType> os::LINKED_STACK< dataType >::top ( ) const  
[inline]
```

12.23 os::LIST< dataType > Class Template Reference

List framework Template for a linked list, insertion is fully defined in subsequent extensions of this class.

Inheritance diagram for `os::LIST< dataType >`:



Public Member Functions

- **LIST ()**
Constructs the linked list Sets the head and tail to NULL, number of elements in the list to 0.
- **virtual ~LIST ()**
Destroys the list Removes every element from the list. Depending on the list format, this may delete everythin in the list.
- **size_t size () const final**
Access size of the list.
- **bool iterable () const final**
Returns if the relevant node type is iterable.
- **bool randomAccess () const final**
Returns if the relevant node type can be accessed randomly.
- **bool remove (const dataType &x) final**
Remove item from the data-structure.
- **bool find (const dataType &x) const final**
Searches for an item.
- **dataType & access (const dataType &x) final**
Mutable item access.
- **const dataType & access (const dataType &x) const final**
Immutable item access.

Static Public Attributes

- **static const bool ITERABLE = true**
Lists are iterable.
- **static const bool RANDOM_ACCESS = false**
Lists do not allow random access.

Protected Member Functions

- **smart_ptr< nodeFrame< dataType > > getFirstNode ()** final
Access to first node.
- **smart_ptr< nodeFrame< dataType > > getLastNode ()** final
Access to last node.
- **const smart_ptr< nodeFrame< dataType > > getFirstNodeConst ()** const final
Constant access to first node.
- **const smart_ptr< nodeFrame< dataType > > getLastNodeConst ()** const final
Constant access to last node.
- **smart_ptr< nodeFrame< dataType > > searchNode (const smart_ptr< dataType > dt)** final
Search for a node.
- **const smart_ptr< nodeFrame< dataType > > searchNodeConst (const smart_ptr< dataType > dt)** const final
Const search for a node.

Protected Attributes

- **size_t _size**
Size of the list.
- **smart_ptr< ITERATOR_CLASS > beg**
First pointer.
- **smart_ptr< ITERATOR_CLASS > end**
Last pointer.

Additional Inherited Members

12.23.1 Detailed Description

```
template<class dataType>
class os::LIST< dataType >
```

List framework Template for a linked list, insertion is fully defined in subsequent extensions of this class.

12.23.2 Constructor & Destructor Documentation

```
template<class dataType > os::LIST< dataType >::LIST ( ) [inline]
```

Constructs the linked list Sets the head and tail to NULL, number of elements in the list to 0.

```
template<class dataType > virtual os::LIST< dataType >::~~LIST ( ) [inline], [virtual]
```

Destroys the list Removes every element from the list. Depending on the list format, this may delete everythin in the list.

12.23.3 Member Function Documentation

```
template<class dataType > dataType& os::LIST< dataType >::access ( const dataType & x )  
[inline], [final], [virtual]
```

Mutable item access.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Returns

Mutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 157).

```
template<class dataType > const dataType& os::LIST< dataType >::access ( const dataType & x )  
const [inline], [final], [virtual]
```

Immutable item access.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Returns

Immutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 157).

```
template<class dataType > bool os::LIST< dataType >::find ( const dataType & x ) const  
[inline], [final], [virtual]
```

Searches for an item.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references.

Returns

True if found, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 158).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::LIST< dataType  
>::getFirstNode ( ) [inline], [final], [protected], [virtual]
```

Access to first node.

Returns

First node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 198).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::LIST< dataType  
>::getFirstNodeConst ( ) const [inline], [final], [protected], [virtual]
```

Constant access to first node.

Returns

Immutable first node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 198).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::LIST< dataType  
>::getLastNode ( ) [inline], [final], [protected], [virtual]
```

Access to last node.

Returns

Last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 198).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::LIST< dataType  
>::getLastNodeConst ( ) const [inline], [final], [protected], [virtual]
```

Constant access to last node.

Returns

Immutable last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 199).

```
template<class dataType > bool os::LIST< dataType >::iterable ( ) const [inline], [final],  
[virtual]
```

Returns if the relevant node type is iterable.

Returns

true

Reimplemented from **os::iteratorSource** (p. 205).

```
template<class dataType > bool os::LIST< dataType >::randomAccess ( ) const [inline],  
[final], [virtual]
```

Returns if the relevant node type can be accessed randomly.

Returns

true

Reimplemented from **os::iteratorSource** (p. 205).


```
template<class dataType > bool os::LIST< dataType >::remove ( const dataType & x )
[inline], [final], [virtual]
```

Remove item from the data-structure.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references.

Returns

True if removed, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 160).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::LIST< dataType
>::searchNode ( const smart_ptr< dataType > dt ) [inline], [final], [protected],
[virtual]
```

Search for a node.

Parameters

in	<i>dt</i>	Pointer to search for
-----------	-----------	-----------------------

Returns

Mutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 199).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::LIST< dataType
>::searchNodeConst ( const smart_ptr< dataType > dt ) const [inline], [final],
[protected], [virtual]
```

Const search for a node.

Parameters

in	<i>dt</i>	Pointer to search for
-----------	-----------	-----------------------

Returns

Immutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 200).

```
template<class dataType > size_t os::LIST< dataType >::size ( ) const [inline], [final],
[virtual]
```

Access size of the list.

Returns

Number of elements in the list

Implements **os::DATASTRUCTURE**< **dataType** > (p. 160).

12.23.4 Member Data Documentation

```
template<class dataType > size_t os::LIST< dataType >::_size [protected]
```

Size of the list.

```
template<class dataType > smart_ptr<ITERATOR_CLASS > os::LIST< dataType >::beg  
[protected]
```

First pointer.

```
template<class dataType > smart_ptr<ITERATOR_CLASS > os::LIST< dataType >::end  
[protected]
```

Last pointer.

```
template<class dataType > const bool os::LIST< dataType >::ITERABLE = true [static]
```

Lists are iterable.

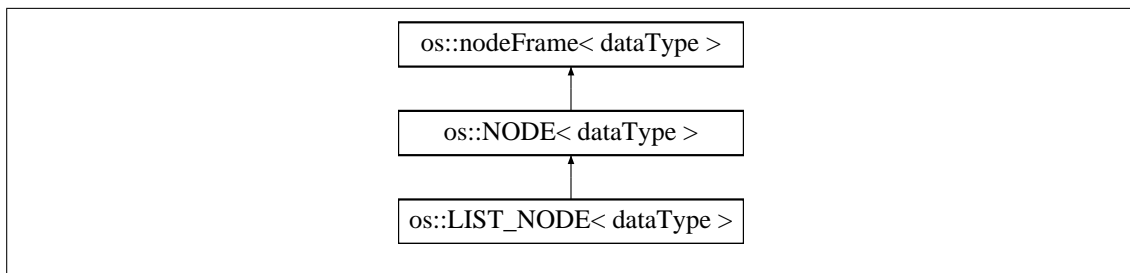
```
template<class dataType > const bool os::LIST< dataType >::RANDOM_ACCESS = false  
[static]
```

Lists do not allow random access.

12.24 os::LIST_NODE< dataType > Class Template Reference

List node Node used in linked lists to store each element.

Inheritance diagram for **os::LIST_NODE**< dataType >:



Public Member Functions

- **~LIST_NODE ()**
Basic destructor Merely dereferences and deletes the contained objects. Does not disassemble the list.
- **void remove ()** final throw (descriptiveException)
Remove this node from the list.
- **bool iterable ()** const final
Returns if the node is iterable.
- **bool randomAccess ()** const final
Returns if the node can be accessed randomly.
- **smart_ptr< nodeFrame< dataType > > getNext ()** final throw (descriptiveException)
Returns the next vector frame.
- **const smart_ptr< nodeFrame< dataType > > getNextConst ()** const final throw (descriptiveException)
Returns the next vector frame.
- **smart_ptr< nodeFrame< dataType > > getPrev ()** final throw (descriptiveException)
Returns the previous vector frame.
- **const smart_ptr< nodeFrame< dataType > > getPrevConst ()** const final throw (descriptiveException)
Returns the previous vector frame.

Static Public Attributes

- static const bool **ITERABLE** = true
List frames are iterable.
- static const bool **RANDOM_ACCESS** = false
List frames do not allow random-access.

Private Member Functions

- **LIST_NODE (DRIVER_CLASS *src, const dataType &dt)**
Private constructor.

Private Attributes

- **smart_ptr< LIST_NODE > prev**
Previous node.
- **smart_ptr< LIST_NODE > next**
Next node.

Additional Inherited Members

12.24.1 Detailed Description

```
template<class dataType>
class os::LIST_NODE< dataType >
```

List node Node used in linked lists to store each element.

12.24.2 Constructor & Destructor Documentation

```
template<class dataType > os::LIST_NODE< dataType >::LIST_NODE ( DRIVER_CLASS * src,
const dataType & dt ) [inline], [private]
```

Private constructor.

Only lists can access the node constructor.

```
template<class dataType > os::LIST_NODE< dataType >::~LIST_NODE ( ) [inline]
```

Basic destructor Merely dereferences and deletes the contained objects. Does not disassemble the list.

12.24.3 Member Function Documentation

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::LIST_NODE< dataType
>::getNext ( ) throw descriptiveException) [inline], [final], [virtual]
```

Returns the next vector frame.

Returns

Next node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 237).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::LIST_NODE<
dataType >::getNextConst ( ) const throw descriptiveException) [inline], [final],
[virtual]
```

Returns the next vector frame.

Returns

Next node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 238).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::LIST_NODE< dataType
>::getPrev ( ) throw descriptiveException) [inline], [final], [virtual]
```

Returns the previous vector frame.

Returns

Previous node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 238).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::LIST_NODE<
dataType >::getPrevConst ( ) const throw descriptiveException) [inline], [final],
[virtual]
```

Returns the previous vector frame.

Returns

Previous node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 238).

```
template<class dataType > bool os::LIST_NODE< dataType >::iterable ( ) const [inline],
[final], [virtual]
```

Returns if the node is iterable.

Returns

object::ITERABLE

Reimplemented from **os::nodeFrame**< **dataType** > (p. 239).

```
template<class dataType > bool os::LIST_NODE< dataType >::randomAccess ( ) const
[inline], [final], [virtual]
```

Returns if the node can be accessed randomly.

Returns

object::RANDOM_ACCESS

Reimplemented from **os::nodeFrame**< **dataType** > (p. 240).

```
template<class dataType > void os::LIST_NODE< dataType >::remove ( ) throw
descriptiveException) [final], [virtual]
```

Remove this node from the list.

Returns

void

Reimplemented from **os::nodeFrame**< **dataType** > (p. 240).

12.24.4 Member Data Documentation

```
template<class dataType > const bool os::LIST_NODE< dataType >::ITERABLE = true [static]
```

List frames are iterable.

```
template<class dataType > smart_ptr<LIST_NODE > os::LIST_NODE< dataType >::next
[private]
```

Next node.

```
template<class dataType > smart_ptr<LIST_NODE > os::LIST_NODE< dataType >::prev
[private]
```

Previous node.

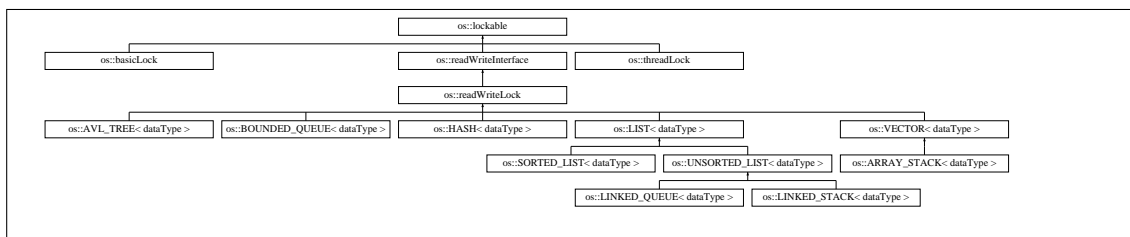
```
template<class dataType > const bool os::LIST_NODE< dataType >::RANDOM_ACCESS = false
[static]
```

List frames do not allow random-access.

12.25 os::lockable Class Reference

An interface defining a class which can be locked.

Inheritance diagram for os::lockable:



Public Member Functions

- **lockable** (bool dMode=**NO_LOCK_CHECK**) throw ()
Lockable constructor.
- virtual ~**lockable** () throw (descriptiveException)
Virtual destructor.
- void **setDeletionMode** (bool delMode) throw ()
Set deletion mode.
- bool **deletionMode** () const throw ()
Access deletion mode.
- virtual void **lock** () const =0 throw ()
Locks the lock.
- virtual void **unlock** () const =0 throw (descriptiveException)
Unlocks the lock.
- virtual bool **locked** () const =0 throw ()
Checks if the lock is locked.
- virtual bool **try_lock** () const =0 throw ()
Attempt to lock the object.
- void **acquire** () const throw ()
*Calls **lockable::lock()** (p. 223)*
- void **release** () const throw (descriptiveException)
*Calls **lockable::unlock()** (p. 223)*

- bool **attemptLock** () const throw ()
Calls **lockable::try_lock()** (p. 223)

Static Public Attributes

- static const bool **NO_LOCK_CHECK** =false
Do check lock status on deletion.
- static const bool **DELETION_LOCK_CHECK** =true
Check lock status on deletion.

Private Attributes

- bool **_delMode**

12.25.1 Detailed Description

An interface defining a class which can be locked.

This interface is used by various types of locks to define a common method of both locking and unlocking.

12.25.2 Constructor & Destructor Documentation

```
os::lockable::lockable ( bool dMode = NO_LOCK_CHECK ) throw ) [inline]
```

Lockable constructor.

```
virtual os::lockable::~~lockable ( ) throw descriptiveException ) [inline], [virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.25.3 Member Function Documentation

```
void os::lockable::acquire ( ) const throw ) [inline]
```

Calls **lockable::lock()** (p. 223)

```
bool os::lockable::attemptLock ( ) const throw ) [inline]
```

Calls **lockable::try_lock()** (p. 223)

```
bool os::lockable::deletionMode ( ) const throw ) [inline]
```

Access deletion mode.

Returns

lockable::_delMode (p. 224)

virtual void os::lockable::lock () const throw) [pure virtual]

Locks the lock.

Returns

void

Implemented in **os::readWriteLock** (p.247), **os::threadLock** (p.275), and **os::basicLock** (p.128).

virtual bool os::lockable::locked () const throw) [pure virtual]

Checks if the lock is locked.

Returns

True if locked, else, false

Implemented in **os::readWriteLock** (p.248), **os::threadLock** (p.275), and **os::basicLock** (p.128).

void os::lockable::release () const throw **descriptiveException**) [inline]

Calls **lockable::unlock()** (p.223)

void os::lockable::setDeletionMode (bool delMode) throw) [inline]

Set deletion mode.

Parameters

in	<i>delMode</i>	Deletion mode to bind
----	----------------	-----------------------

Returns

void

virtual bool os::lockable::try_lock () const throw) [pure virtual]

Attempt to lock the object.

Locks the object if possible, otherwise, returns false.

Returns

True if lock successful

Implemented in **os::readWriteLock** (p.248), **os::threadLock** (p.275), and **os::basicLock** (p.128).

virtual void os::lockable::unlock () const throw **descriptiveException**) [pure virtual]

Unlocks the lock.

Returns

void

Implemented in **os::readWriteLock** (p.249), **os::threadLock** (p.275), and **os::basicLock** (p.128).

12.25.4 Member Data Documentation

bool os::lockable::_delMode [private]

Holds deletion mode of the lockable object

const bool os::lockable::DELETION_LOCK_CHECK =true [static]

Check lock status on deletion.

const bool os::lockable::NO_LOCK_CHECK =false [static]

Do check lock status on deletion.

12.26 os::matrix< dataType > Class Template Reference

Raw matrix.

Public Member Functions

- **matrix** (uint32_t w=0, uint32_t h=0) throw ()
Default constructor.
- **matrix** (const **matrix**< dataType > &m) throw ()
Copy constructor.
- **matrix** (const **indirectMatrix**< dataType > &m) throw ()
Copy constructor.
- **matrix** (const **smart_ptr**< dataType > d, uint32_t w, uint32_t h) throw ()
Data array constructor.
- **matrix** (**smart_ptr**< **smart_ptr**< dataType > > d, uint32_t w, uint32_t h) throw ()
Indirect data array constructor.
- virtual ~**matrix** () throw (std::exception)
Virtual destructor.
- **matrix**< dataType > & **operator=** (const **matrix**< dataType > &m) throw ()
Equality constructor.
- **matrix**< dataType > & **operator=** (const **indirectMatrix**< dataType > &m) throw ()
Equality constructor.
- dataType & **get** (uint32_t w, uint32_t h) throw ()
Return matrix element.
- const dataType & **constGet** (uint32_t w, uint32_t h) const throw ()

- *Return constant matrix element.*
- `dataType & operator() (uint32_t w, uint32_t h) throw ()`
Return matrix element.
- `smart_ptr< dataType > getArray () throw ()`
Return pointer to the array.
- `const smart_ptr< dataType > getConstArray () const throw ()`
Return a constant pointer to the array.
- `uint32_t width () const throw ()`
Return _width of matrix.
- `uint32_t height () const throw ()`
Return _height of matrix.

Private Attributes

- `uint32_t _width`
Width of the matrix.
- `uint32_t _height`
Height of the matrix.
- `smart_ptr< dataType > data`
Data array.

Friends

- `class indirectMatrix< dataType >`
Indirect matrix interacting with raw matrix.

12.26.1 Detailed Description

```
template<class dataType>
class os::matrix< dataType >
```

Raw matrix.

This matrix class contains an array of the data type. It can interact with `os::indirectMatrix<dataType>`.

12.26.2 Constructor & Destructor Documentation

```
template<class dataType> os::matrix< dataType >::matrix ( uint32_t w = 0, uint32_t h = 0 )
throw ()
```

Default constructor.

Constructs array of size w*h and sets all of the data to 0. If no `_width` and `_height` are provided, the data array is not initialized.

Parameters

in	<i>w</i>	Width of matrix, default 0
in	<i>h</i>	Height of matrix, default 0

```
template<class dataType> os::matrix< dataType >::matrix ( const matrix< dataType > & m )
throw )
```

Copy constructor.

Constructs a new raw matrix from the given raw matrix. The two matrices do not share the same data array.

Parameters

in	<i>m</i>	Matrix to be copied
----	----------	---------------------

```
template<class dataType> os::matrix< dataType >::matrix ( const indirectMatrix< dataType > &
m ) throw )
```

Copy constructor.

Constructs a new raw matrix from the given indirect matrix. The raw matrix converts the array of pointers to an array of objects

Parameters

in	<i>m</i>	Indirect matrix to be copied
----	----------	------------------------------

```
template<class dataType> os::matrix< dataType >::matrix ( const smart_ptr< dataType > d,
uint32_t w, uint32_t h ) throw )
```

Data array constructor.

Constructs a new raw matrix from an array of the correct data type. This constructor will build an new array based on the specified size.

Parameters

in	<i>d</i>	Data array to be copied
in	<i>w</i>	Width of matrix
in	<i>d</i>	Height of matrix

```
template<class dataType> os::matrix< dataType >::matrix ( smart_ptr< smart_ptr< dataType >
> d, uint32_t w, uint32_t h ) throw )
```

Indirect data array constructor.

Constructs a new raw matrix from an indirect array of the correct data type. This constructor will build an new array based on the specified size.

Parameters

in	<i>d</i>	Indirect data array to be copied
----	----------	----------------------------------

Parameters

in	<i>w</i>	Width of matrix
in	<i>d</i>	Height of matrix

```
template<class dataType> virtual os::matrix< dataType >::~matrix ( ) throw std::exception)
[inline], [virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.26.3 Member Function Documentation

```
template<class dataType> const dataType& os::matrix< dataType >::constGet ( uint32_t w,
uint32_t h ) const throw )
```

Return constant matrix element.

Uses a `_width` and `_height` position to index an element of the array. This function returns a constant reference, meaning changes cannot be made to the matrix.

Parameters

in	<i>w</i>	X position
in	<i>h</i>	Y position

Returns

Constant reference to matrix element

```
template<class dataType> dataType& os::matrix< dataType >::get ( uint32_t w, uint32_t h )
throw )
```

Return matrix element.

Uses a `_width` and `_height` position to index an element of the array. This function returns a reference, allowing for changes to be made to the matrix.

Parameters

in	<i>w</i>	X position
in	<i>h</i>	Y position

Returns

Modifiable reference to matrix element

```
template<class dataType> smart_ptr<dataType> os::matrix< dataType >::getArray ( ) throw )  
[inline]
```

Return pointer to the array.

The array which is returned allows for modification of the array. It is up to functions using this array to ensure the integrity of the matrix.

Returns

os::matrix<dataType>::data (p. 230)

```
template<class dataType> const smart_ptr<dataType> os::matrix< dataType >::getConstArray ( ) const throw ) [inline]
```

Return a constant pointer to the array.

The array which is returned allows for access to the array. The provided array may not be modified.

Returns

os::matrix<dataType>::data (p. 230)

```
template<class dataType> uint32_t os::matrix< dataType >::height ( ) const throw ) [inline]
```

Return _height of matrix.

Returns

matrix<dataType>::_height (p. 229)

```
template<class dataType> dataType& os::matrix< dataType >::operator() ( uint32_t w, uint32_t h ) throw ) [inline]
```

Return matrix element.

Uses a _width and _height position to index an element of the array. This function returns a reference, allowing for changes to be made to the matrix.

Parameters

in	<i>w</i>	X position
in	<i>h</i>	Y position

Returns

Modifiable reference to matrix element

```
template<class dataType> matrix<dataType>& os::matrix< dataType >::operator= ( const matrix< dataType > & m ) throw )
```

Equality constructor.

Re-constructs the raw matrix from another raw matrix. Note that the two matrices do not share the same data array.

Parameters

in	m	Reference to matrix being copied
----	---	----------------------------------

Returns

Reference to self

```
template<class dataType> matrix<dataType>& os::matrix< dataType >::operator= ( const  
indirectMatrix< dataType > & m ) throw )
```

Equality constructor.

Re-constructs the raw matrix from an indirect matrix. Note that the two matrices do not share the same data array.

Parameters

in	m	Reference to matrix being copied
----	---	----------------------------------

Returns

Reference to self

```
template<class dataType> uint32_t os::matrix< dataType >::width ( ) const throw ) [inline]
```

Return _width of matrix.

Returns

matrix<**dataType**>::_width (p. 229)

12.26.4 Friends And Related Function Documentation

```
template<class dataType> friend class indirectMatrix< dataType > [friend]
```

Indirect matrix interacting with raw matrix.

The `os::indirectMatrix<dataType>` class must be able to access the size and data of the raw matrix because an indirect matrix can be constructed from a raw matrix.

12.26.5 Member Data Documentation

```
template<class dataType> uint32_t os::matrix< dataType >::_height [private]
```

Height of the matrix.

```
template<class dataType> uint32_t os::matrix< dataType >::_width [private]
```

Width of the matrix.

```
template<class dataType> smart_ptr<dataType> os::matrix< dataType >::data [private]
```

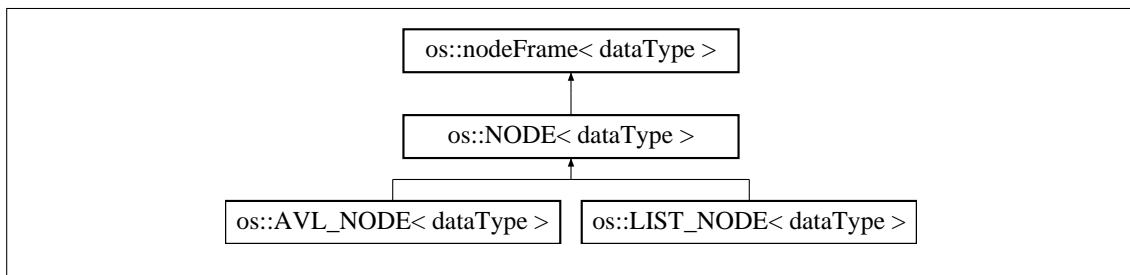
Data array.

For the raw matrix class, this array contains all of the data used by the matrix in a block of size `_width*_height`.

12.27 os::NODE< dataType > Class Template Reference

Object node definition.

Inheritance diagram for `os::NODE< dataType >`:



Public Member Functions

- **NODE** ()
Default constructor.
- **NODE** (const **NODE** &cph)
Copy constructor.
- virtual ~**NODE** ()
Virtual destructor.
- **NODE** (const dataType &dt)
Data constructor Initializes the node with specific data.
- **smart_ptr**< dataType > **get** () final throw ()
Access data pointer.
- const **smart_ptr**< dataType > **constGet** () const final throw ()
Const access data pointer.
- dataType & **operator*** () final throw (descriptiveException)
Access to data reference.
- const dataType & **operator*** () const final throw (descriptiveException)
Const access to data reference.
- bool **valid** () const final
Valid data query.
- int **compare** (const objectNode< dataType > &cmp) const
Compare two nodes.
- **operator size_t** () const
size_t cast

Protected Attributes

- `dataType _data`

Data held by this node.

Additional Inherited Members

12.27.1 Detailed Description

```
template<class dataType>
class os::NODE< dataType >
```

Object node definition.

Note that this class is defined in three forms: `objectNode`, `pointerNode` and `rawPointerNode`. In short, this class and it's alternate forms specify the details of the **nodeFrame** (p.233), allowing for holding both objects and pointers to object in the various provided data-structures.

12.27.2 Constructor & Destructor Documentation

```
template<class dataType > os::NODE< dataType >::NODE ( ) [inline]
```

Default constructor.

```
template<class dataType > os::NODE< dataType >::NODE ( const NODE< dataType > & cph )
[inline]
```

Copy constructor.

Parameters

in	<i>cph</i>	Node to be copied
----	------------	-------------------

```
template<class dataType > virtual os::NODE< dataType >::~NODE ( ) [inline], [virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

```
template<class dataType > os::NODE< dataType >::NODE ( const dataType & dt ) [inline]
```

Data constructor Initializes the node with specific data.

Parameters

in	<i>dt</i>	Reference to data
----	-----------	-------------------

12.27.3 Member Function Documentation

```
template<class dataType > int os::NODE< dataType >::compare ( const objectNode< dataType >
& cmp ) const [inline]
```

Compare two nodes.

Uses the == and > operators of the given data type to compare two nodes. Note that other forms of this class use different means of comparison.

Parameters

in	cmp	Node to compare against
----	-----	-------------------------

Returns

0 if equal, 1 if greater than, -1 if less than

```
template<class dataType > const smart_ptr<dataType> os::NODE< dataType >::constGet ( )
const throw ) [inline], [final], [virtual]
```

Const access data pointer.

Returns a pointer to the data held in this node. Note that this function will return "data" if the node is in pointer form.

Returns

Immutable pointer to data

Implements **os::nodeFrame**< **dataType** > (p. 236).

```
template<class dataType > smart_ptr<dataType> os::NODE< dataType >::get ( ) throw )
[inline], [final], [virtual]
```

Access data pointer.

Returns a pointer to the data held in this node. Note that this function will return "data" if the node is in pointer form.

Returns

Mutable pointer to _data

Implements **os::nodeFrame**< **dataType** > (p. 237).

```
template<class dataType > os::NODE< dataType >::operator size_t ( ) const [inline]
```

size_t cast

Used in hash functions to place this object in a hash table. Note that other forms of this class use different methods for converting to size_t.

Returns

data cast to size_t

```
template<class dataType > dataType& os::NODE< dataType >::operator* ( ) throw
descriptiveException) [inline], [final], [virtual]
```

Access to data reference.

Returns an object reference to the data enclosed in the node. Note that if the node is in pointer form, a reference to the object pointed to by the pointer will be returned.

Returns

Mutable object reference

Implements **os::nodeFrame**< **dataType** > (p. 239).

```
template<class dataType > const dataType& os::NODE< dataType >::operator* ( ) const throw
descriptiveException) [inline], [final], [virtual]
```

Const access to data reference.

Returns an object reference to the data enclosed in the node. Note that if the node is in pointer form, a reference to the object pointed to by the pointer will be returned.

Returns

Immutable object reference

Implements **os::nodeFrame**< **dataType** > (p. 240).

```
template<class dataType > bool os::NODE< dataType >::valid ( ) const [inline], [final],
[virtual]
```

Valid data query.

Checks if the provided data is valid. Note that pointer forms of this class return false if the provided pointer is NULL

Returns

true

Reimplemented from **os::nodeFrame**< **dataType** > (p. 241).

12.27.4 Member Data Documentation

```
template<class dataType > dataType os::NODE< dataType >::_data [protected]
```

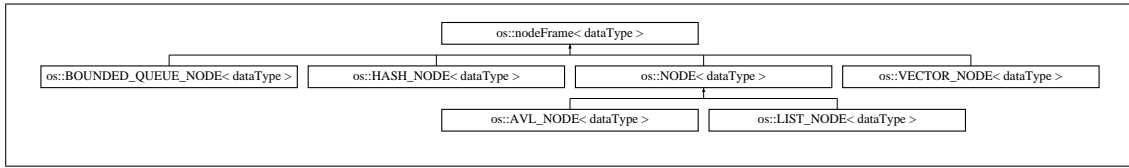
Data held by this node.

Note that other forms of this class hold pointers instead of objects

12.28 **os::nodeFrame**< **dataType** > Class Template Reference

nodeFrame (p. 233) interface

Inheritance diagram for **os::nodeFrame**< **dataType** >:



Public Member Functions

- virtual **~nodeFrame** ()
Virtual destructor.
- virtual **smart_ptr**< dataType > **get** ()=0 throw ()
Returns a pointer.
- virtual const **smart_ptr**< dataType > **constGet** () const =0 throw ()
Returns a const pointer.
- const **smart_ptr**< dataType > **get** () const throw ()
Returns a const pointer.
- **smart_ptr**< dataType > **operator->** () throw ()
Member access.
- const **smart_ptr**< dataType > **operator->** () const throw ()
Member access.
- virtual dataType & **operator*** ()=0 throw (descriptiveException)
De-reference.
- virtual const dataType & **operator*** () const =0 throw (descriptiveException)
De-reference.
- virtual bool **valid** () const
Valid data query Checks if the provided data is valid.
- **operator bool** () const throw ()
Boolean conversion.
- bool **operator!** () const throw ()
Inverted boolean conversion.
- virtual bool **iterable** () const
Returns if the node is iterable.
- virtual bool **randomAccess** () const
Returns if the node can be accessed randomly.
- virtual bool **isObject** () const =0
True if holding an object.
- virtual bool **isPointer** () const =0
True is holding a pointer.
- virtual bool **usingComparison** () const =0
Defines which comparison method is used.
- virtual void **remove** () throw (descriptiveException)
Remove node from data structure.
- virtual **smart_ptr**< **nodeFrame**< dataType > > **getNext** () throw (descriptiveException)

Returns the next **nodeFrame** (p. 233).

- virtual const **smart_ptr< nodeFrame< dataType > > getNextConst** () const throw (descriptiveException)

Returns the next **nodeFrame** (p. 233).

- const **smart_ptr< nodeFrame< dataType > > getNext** () const throw (descriptiveException)

Returns the next **nodeFrame** (p. 233).

- virtual **smart_ptr< nodeFrame< dataType > > getPrev** () throw (descriptiveException)

Returns the previous **nodeFrame** (p. 233).

- virtual const **smart_ptr< nodeFrame< dataType > > getPrevConst** () const throw (descriptiveException)

Returns the previous **nodeFrame** (p. 233).

- const **smart_ptr< nodeFrame< dataType > > getPrev** () const throw (descriptiveException)

Returns the previous **nodeFrame** (p. 233).

- virtual **smart_ptr< nodeFrame< dataType > > access** (long offset) throw (descriptiveException)

Access node by index.

- virtual const **smart_ptr< nodeFrame< dataType > > constAccess** (long offset) const throw (descriptiveException)

Access node by index.

- const **smart_ptr< nodeFrame< dataType > > access** (long offset) const throw (descriptiveException)

Access node by index.

Static Public Attributes

- static const bool **ITERABLE** = false

By default, nodeFrames are not iterable.

- static const bool **RANDOM_ACCESS** = false

No random-access by default.

12.28.1 Detailed Description

```
template<class dataType>
class os::nodeFrame< dataType >
```

nodeFrame (p. 233) interface

Declares a number of functions to be used by various nodes inside data-structures.

12.28.2 Constructor & Destructor Documentation

```
template<class dataType > virtual os::nodeFrame< dataType >::~nodeFrame ( ) [inline],
[virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.28.3 Member Function Documentation

```
template<class dataType > virtual smart_ptr<nodeFrame<dataType> > os::nodeFrame<
dataType >::access ( long offset ) throw descriptiveException) [inline], [virtual]
```

Access node by index.

Access a node offset from the current node by some value. If a node cannot be randomly accessed, an exception will be thrown.

Returns

Offset node, mutable

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 140), **os::VECTOR_NODE**< **dataType** > (p. 309), and **os::AVL_NODE**< **dataType** > (p. 112).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::nodeFrame<
dataType >::access ( long offset ) const throw descriptiveException) [inline]
```

Access node by index.

Access a node offset from the current node by some value. If a node cannot be randomly accessed, an exception will be thrown.

Returns

Offset node, immutable

```
template<class dataType > virtual const smart_ptr<nodeFrame<dataType> > os::nodeFrame<
dataType >::constAccess ( long offset ) const throw descriptiveException) [inline],
[virtual]
```

Access node by index.

Access a node offset from the current node by some value. If a node cannot be randomly accessed, an exception will be thrown.

Returns

Offset node, immutable

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 140), **os::VECTOR_NODE**< **dataType** > (p. 309), and **os::AVL_NODE**< **dataType** > (p. 112).

```
template<class dataType > virtual const smart_ptr<dataType> os::nodeFrame< dataType
>::constGet ( ) const throw ) [pure virtual]
```

Returns a const pointer.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

Implemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 140), **os::VECTOR_NODE**< **dataType** > (p. 309), **os::HASH_NODE**< **dataType** > (p. 178), and **os::NODE**< **dataType** > (p. 232).

```
template<class dataType > virtual smart_ptr<dataType> os::nodeFrame< dataType >::get ( )  
throw ) [pure virtual]
```

Returns a pointer.

Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

Implemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 140), **os::VECTOR_NODE**< **dataType** > (p. 310), **os::HASH_NODE**< **dataType** > (p. 179), and **os::NODE**< **dataType** > (p. 232).

```
template<class dataType > const smart_ptr<dataType> os::nodeFrame< dataType >::get ( )  
const throw ) [inline]
```

Returns a const pointer.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

```
template<class dataType > virtual smart_ptr<nodeFrame<dataType> > os::nodeFrame<  
dataType >::getNext ( ) throw descriptiveException) [inline], [virtual]
```

Returns the next **nodeFrame** (p. 233).

Attempts to return the next node frame, assuming that the node in question is iterable. If a node is not iterable, an exception will be thrown.

Returns

Next node, mutable

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 141), **os::VECTOR_NODE**< **dataType** > (p. 310), **os::HASH_NODE**< **dataType** > (p. 179), **os::AVL_NODE**< **dataType** > (p. 113), and **os::LIST_NODE**< **dataType** > (p. 219).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::nodeFrame<  
dataType >::getNext ( ) const throw descriptiveException) [inline]
```

Returns the next **nodeFrame** (p. 233).

Attempts to return the next node frame, assuming that the node in question is iterable. If a node is not iterable, an exception will be thrown.

Returns

Next node, immutable

```
template<class dataType > virtual const smart_ptr<nodeFrame<dataType> > os::nodeFrame<
dataType >::getNextConst ( ) const throw descriptiveException [inline], [virtual]
```

Returns the next **nodeFrame** (p. 233).

Attempts to return the next node frame, assuming that the node in question is iterable. If a node is not iterable, an exception will be thrown.

Returns

Next node, immutable

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 141), **os::VECTOR_NODE**< **dataType** > (p. 310), **os::HASH_NODE**< **dataType** > (p. 179), **os::AVL_NODE**< **dataType** > (p. 113), and **os::LIST_NODE**< **dataType** > (p. 219).

```
template<class dataType > virtual smart_ptr<nodeFrame<dataType> > os::nodeFrame<
dataType >::getPrev ( ) throw descriptiveException [inline], [virtual]
```

Returns the previous **nodeFrame** (p. 233).

Attempts to return the previous node frame, assuming that the node in question is iterable. If a node is not iterable, an exception will be thrown.

Returns

Previous node, mutable

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 141), **os::VECTOR_NODE**< **dataType** > (p. 310), **os::HASH_NODE**< **dataType** > (p. 179), **os::AVL_NODE**< **dataType** > (p. 113), and **os::LIST_NODE**< **dataType** > (p. 219).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::nodeFrame<
dataType >::getPrev ( ) const throw descriptiveException [inline]
```

Returns the previous **nodeFrame** (p. 233).

Attempts to return the previous node frame, assuming that the node in question is iterable. If a node is not iterable, an exception will be thrown.

Returns

Previous node, immutable

```
template<class dataType > virtual const smart_ptr<nodeFrame<dataType> > os::nodeFrame<
dataType >::getPrevConst ( ) const throw descriptiveException [inline], [virtual]
```

Returns the previous **nodeFrame** (p. 233).

Attempts to return the previous node frame, assuming that the node in question is iterable. If a node is not iterable, an exception will be thrown.

Returns

Previous node, immutable

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 141), **os::VECTOR_NODE**< **dataType** > (p. 310), **os::HASH_NODE**< **dataType** > (p. 180), **os::AVL_NODE**< **dataType** > (p. 113), and **os::LIST_NODE**< **dataType** > (p. 220).

```
template<class dataType > virtual bool os::nodeFrame< dataType >::isObject ( ) const [pure virtual]
```

True if holding an object.

```
template<class dataType > virtual bool os::nodeFrame< dataType >::isPointer ( ) const [pure virtual]
```

True is holding a pointer.

```
template<class dataType > virtual bool os::nodeFrame< dataType >::iterable ( ) const [inline], [virtual]
```

Returns if the node is iterable.

Returns

`object::ITERABLE`

Reimplemented in **os::BOUNDED_QUEUE_NODE**< dataType > (p. 141), **os::VECTOR_NODE**< dataType > (p. 311), **os::HASH_NODE**< dataType > (p. 180), **os::AVL_NODE**< dataType > (p. 114), and **os::LIST_NODE**< dataType > (p. 220).

```
template<class dataType > os::nodeFrame< dataType >::operator bool ( ) const throw [inline]
```

Boolean conversion.

Returns

valid() (p. 241)

```
template<class dataType > bool os::nodeFrame< dataType >::operator! ( ) const throw [inline]
```

Inverted boolean conversion.

Returns

!valid()

```
template<class dataType > virtual dataType& os::nodeFrame< dataType >::operator* ( ) throw descriptiveException [pure virtual]
```

De-reference.

Returns a reference to the contained object, the reference can be modified.

Returns

Contained object

Implemented in **os::BOUNDED_QUEUE_NODE**< dataType > (p. 142), **os::VECTOR_NODE**< dataType > (p. 311), **os::HASH_NODE**< dataType > (p. 180), and **os::NODE**< dataType > (p. 233).


```
template<class dataType > virtual const dataType& os::nodeFrame< dataType >::operator* ( )
const throw descriptiveException) [pure virtual]
```

De-reference.

Returns a const reference to the contained object, the reference cannot be modified.

Returns

Contained object

Implemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 142), **os::VECTOR_NODE**< **dataType** > (p. 311), **os::HASH_NODE**< **dataType** > (p. 180), and **os::NODE**< **dataType** > (p. 233).

```
template<class dataType > smart_ptr<dataType> os::nodeFrame< dataType >::operator-> ( )
throw ) [inline]
```

Member access.

Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

```
template<class dataType > const smart_ptr<dataType> os::nodeFrame< dataType >::operator->
( ) const throw ) [inline]
```

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

```
template<class dataType > virtual bool os::nodeFrame< dataType >::randomAccess ( ) const
[inline], [virtual]
```

Returns if the node can be accessed randomly.

Returns

object::RANDOM_ACCESS

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 142), **os::VECTOR_NODE**< **dataType** > (p. 311), **os::HASH_NODE**< **dataType** > (p. 180), **os::AVL_NODE**< **dataType** > (p. 115), and **os::LIST_NODE**< **dataType** > (p. 220).

```
template<class dataType > virtual void os::nodeFrame< dataType >::remove ( ) throw
descriptiveException) [inline], [virtual]
```

Remove node from data structure.

Returns

void

Reimplemented in **os::BOUNDED_QUEUE_NODE< dataType >** (p. 142), **os::VECTOR_NODE< dataType >** (p. 311), **os::HASH_NODE< dataType >** (p. 181), **os::AVL_NODE< dataType >** (p. 115), and **os::LIST_NODE< dataType >** (p. 220).

```
template<class dataType > virtual bool os::nodeFrame< dataType >::usingComparison ( ) const  
[pure virtual]
```

Defines which comparison method is used.

```
template<class dataType > virtual bool os::nodeFrame< dataType >::valid ( ) const [inline],  
[virtual]
```

Valid data query Checks if the provided data is valid.

Returns

true if valid, else, false

Reimplemented in **os::BOUNDED_QUEUE_NODE< dataType >** (p. 142), **os::VECTOR_NODE< dataType >** (p. 312), **os::HASH_NODE< dataType >** (p. 181), and **os::NODE< dataType >** (p. 233).

12.28.4 Member Data Documentation

```
template<class dataType > const bool os::nodeFrame< dataType >::ITERABLE = false  
[static]
```

By default, nodeFrames are not iterable.

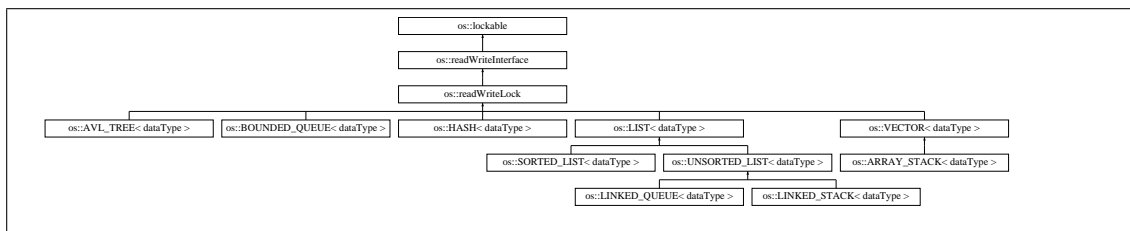
```
template<class dataType > const bool os::nodeFrame< dataType >::RANDOM_ACCESS = false  
[static]
```

No random-access by default.

12.29 os::readWriteInterface Class Reference

Read/write interface.

Inheritance diagram for os::readWriteInterface:



Public Member Functions

- virtual **~readWriteInterface** () throw (descriptiveException)
Virtual destructor.
- virtual void **increment** () const =0 throw ()
Acquires the structure for reading.
- virtual bool **try_increment** () const =0 throw ()
Attempt to acquire the structure for reading.
- virtual void **decrement** () const =0 throw (descriptiveException)
Releases the structure after reading.
- void **operator++** () const throw ()
Calls increment.
- void **operator--** () const throw (descriptiveException)
Calls decrement.
- void **operator++** (int param) const throw ()
Calls increment.
- void **operator--** (int param) const throw (descriptiveException)
Calls decrement.
- void **readLock** () const throw ()
Calls increment.
- void **attemptReadLock** () const throw ()
Calls try_increment.
- void **readUnlock** () const throw (descriptiveException)
Calls decrement.

Additional Inherited Members

12.29.1 Detailed Description

Read/write interface.

Defines an interface to be used by class which need one writer but allow multiple readers.

12.29.2 Constructor & Destructor Documentation

```
virtual os::readWriteInterface::~readWriteInterface ( ) throw descriptiveException) [inline],  
[virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.29.3 Member Function Documentation

`void os::readWriteInterface::attemptReadLock () const throw) [inline]`

Calls `try_increment`.

Returns

`void`

`virtual void os::readWriteInterface::decrement () const throw descriptiveException) [pure virtual]`

Releases the structure after reading.

Returns

`void`

Implemented in **`os::readWriteLock`** (p. 247).

`virtual void os::readWriteInterface::increment () const throw) [pure virtual]`

Acquires the structure for reading.

Returns

`void`

Implemented in **`os::readWriteLock`** (p. 247).

`void os::readWriteInterface::operator++ () const throw) [inline]`

Calls `increment`.

Returns

`void`

`void os::readWriteInterface::operator++ (int param) const throw) [inline]`

Calls `increment`.

Returns

`void`

`void os::readWriteInterface::operator-- () const throw descriptiveException) [inline]`

Calls `decrement`.

Returns

`void`

```
void os::readWriteInterface::operator-- ( int param ) const throw descriptiveException)
[inline]
```

Calls decrement.

Returns

void

```
void os::readWriteInterface::readLock ( ) const throw ) [inline]
```

Calls increment.

Returns

void

```
void os::readWriteInterface::readUnlock ( ) const throw descriptiveException) [inline]
```

Calls decrement.

Returns

void

```
virtual bool os::readWriteInterface::try_increment ( ) const throw ) [pure virtual]
```

Attempt to acquire the structure for reading.

Returns

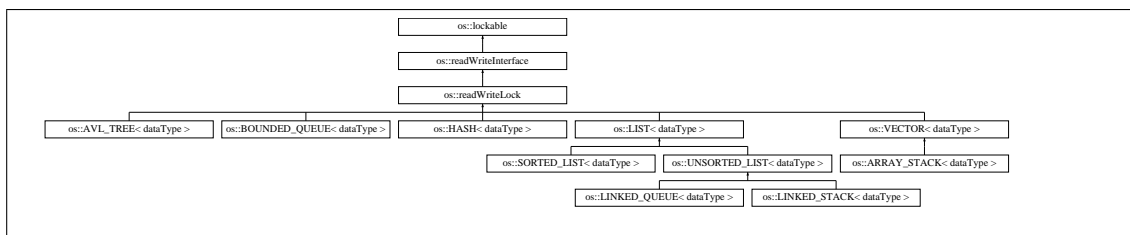
True if acquired, else, false

Implemented in **os::readWriteLock** (p. 248).

12.30 os::readWriteLock Class Reference

Read/write lock.

Inheritance diagram for os::readWriteLock:



Public Types

- enum **lockType** { **STANDARD** =0, **RECURSIVE** }

Enum for read/write lock types.

Public Member Functions

- **readWriteLock** (**readWriteLock::lockType** typ=**readWriteLock::STANDARD**) throw ()
Read/write lock constructor.
- virtual **~readWriteLock** () throw (descriptiveException)
Virtual destructor.
- void **lock** () const final throw ()
Lock to prevent reads.
- bool **try_lock** () const final throw ()
Attempt to lock the object.
- void **unlock** () const final throw (descriptiveException)
Unlock to allow reads.
- void **setMax** (size_t mx) throw (descriptiveException)
Set maximum number of readers.
- void **increment** () const final throw ()
Acquires the structure for reading.
- bool **try_increment** () const final throw ()
Attempt to acquire the structure for reading.
- void **decrement** () const final throw (descriptiveException)
Releases the structure after reading.
- bool **locked** () const throw ()
Checks if the lock is locked.
- **lockType** type () const throw ()
Return the type of lock used.
- std::thread::id **lockedThreadID** () const throw ()
Return the thread which has locked the object.
- size_t **getMax** () const
Return the maximum number of reader threads.
- size_t **numReaders** () const
Returns the number of readers.
- size_t **counter** () const
Returns the number of readers.

Private Member Functions

- **readWriteLock** (const **readWriteLock** &cpy)
Undefined copy-constructor.

Private Attributes

- **threadCounter lockedThread**
Holds the thread which last locked.
- **lockType _type**
The type of lock.
- **basicLock lckb**
Basic lock used to secure intrinsics.
- **simpleHash< threadCounter > * _hash**
Pointer to reader hash bank.
- **bool _attemptingLock**
Holds writer lock action state.
- **bool _locked**
Holds writer lock status.
- **size_t _counter**
Number of threads which have locked.
- **size_t _max**
Maximum number of readers.

Additional Inherited Members

12.30.1 Detailed Description

Read/write lock.

Allows for multiple threads accessing a thread to read, but only one accessing the thread to write.

12.30.2 Member Enumeration Documentation

enum **os::readWriteLock::lockType**

Enum for read/write lock types.

Enumerator

STANDARD Use standard mutex rules.

RECURSIVE Use recursive mutex rules.

12.30.3 Constructor & Destructor Documentation

`os::readWriteLock::readWriteLock (const readWriteLock & cpy) [inline], [private]`

Undefined copy-constructor.

`os::readWriteLock::readWriteLock (readWriteLock::lockType typ = readWriteLock::STANDARD
) throw)`

Read/write lock constructor.

Constructs a read/write lock, accepting a lock type definition.

Parameters

in	typ	Standard lock by default
----	-----	--------------------------

virtual os::readWriteLock::~~readWriteLock () throw **descriptiveException**) [virtual]

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.30.4 Member Function Documentation

size_t os::readWriteLock::counter () const [inline]

Returns the number of readers.

Returns

readWriteLock::_counter (p. 249)

void os::readWriteLock::decrement () const throw **descriptiveException**) [final], [virtual]

Releases the structure after reading.

Returns

void

Implements **os::readWriteInterface** (p. 243).

size_t os::readWriteLock::getMax () const [inline]

Return the maximum number of reader threads.

Returns

readWriteLock::_max (p. 249)

void os::readWriteLock::increment () const throw) [final], [virtual]

Acquires the structure for reading.

Returns

void

Implements **os::readWriteInterface** (p. 243).

void os::readWriteLock::lock () const throw) [final], [virtual]

Lock to prevent reads.

Returns

void

Implements **os::lockable** (p. 223).

bool os::readWriteLock::locked () const throw) [inline], [virtual]

Checks if the lock is locked.

Returns

True if locked, else, false

Implements **os::lockable** (p. 223).

std::thread::id os::readWriteLock::lockedThreadID () const throw) [inline]

Return the thread which has locked the object.

Returns

readWriteLock::lockedThread.id()

size_t os::readWriteLock::numReaders () const

Returns the number of readers.

Returns

readWriteLock::_counter (p. 249) or **readWriteLock::_hash** (p. 249)->numElements

void os::readWriteLock::setMax (size_t mx) throw **descriptiveException**)

Set maximum number of readers.

Please note that if the **readWriteLock** (p. 244) is declared to be recursive, this will also set the size of the hash table.

Returns

void

bool os::readWriteLock::try_increment () const throw) [final], [virtual]

Attempt to acquire the structure for reading.

Returns

True if acquired, else, false

Implements **os::readWriteInterface** (p. 244).

bool os::readWriteLock::try_lock () const throw) [final], [virtual]

Attempt to lock the object.

Locks the object if possible, otherwise, returns false.

Returns

True if lock successful

Implements **os::lockable** (p. 223).

lockType os::readWriteLock::type () const throw) [inline]

Return the type of lock used.

Returns

readWriteLock::_type (p. 249)

void os::readWriteLock::unlock () const throw **descriptiveException**) [final], [virtual]

Unlock to allow reads.

Returns

void

Implements **os::lockable** (p. 223).

12.30.5 Member Data Documentation

bool os::readWriteLock::_attemptingLock [mutable], [private]

Holds writer lock action state.

size_t os::readWriteLock::_counter [mutable], [private]

Number of threads which have locked.

simpleHash<threadCounter>* os::readWriteLock::_hash [mutable], [private]

Pointer to reader hash bank.

bool os::readWriteLock::_locked [mutable], [private]

Holds writer lock status.

size_t os::readWriteLock::_max [private]

Maximum number of readers.

lockType os::readWriteLock::_type [private]

The type of lock.

basicLock os::readWriteLock::lckb [private]

Basic lock used to secure intrinsics.

threadCounter os::readWriteLock::lockedThread [mutable], [private]

Holds the thread which last locked.

12.31 os::simpleHash< dataType > Class Template Reference

Basic hash function.

Public Member Functions

- **simpleHash** (size_t(&hashFunc)(const dataType &) throw())=**simpleHash**< dataType >::**defaultHashFunction**, bool(&compFunc)(const dataType &, const dataType &) throw())=**simpleHash**< dataType >::**defaultCompareFunction**, size_t sz=**DEFAULT_SIZE**) throw (descriptiveException)
Simple hash constructor Instantiates the simple hash constructor with a hash function and size.
- virtual ~**simpleHash** ()
Virtual destructor.
- size_t **hashFunction** (const dataType &ntp) throw ()
Call the hash function.
- void **setSize** (size_t sz)
Attempts to set the size of the hash table.
- size_t **size** () const throw ()
Return size of hash table.
- size_t **numElements** () const throw ()
Return the number of elements in the structure.
- dataType & **insert** (const dataType &ntp) throw (descriptiveException)
Insert data into the table.
- bool **remove** (const dataType &ntp)
Remove data from the table.
- bool **exists** (const dataType &ntp) const throw ()
Check if data exists in the table.
- size_t **find** (const dataType &ntp) const throw ()
Find data in the table.
- void **empty** ()
Clear the hash table.
- bool **atPosition** (size_t pos) const throw ()
Check position in the table.
- bool **check** (size_t pos) const throw ()
Check position in the table.
- dataType & **at** (size_t pos) throw (descriptiveException)
Access element by index.
- const dataType & **at** (size_t pos) const throw (descriptiveException)
Const access element by index.
- dataType & **operator[]** (size_t pos) throw (descriptiveException)
Access element by index.
- const dataType & **operator[]** (size_t pos) const throw (descriptiveException)
Const access element by index.

Static Public Member Functions

- static size_t **defaultHashFunction** (const dataType &dt) throw ()
Default hash function.
- static bool **defaultCompareFunction** (const dataType &dt1, const dataType &dt2) throw ()
Default compare function.

Static Public Attributes

- static const size_t **DEFAULT_SIZE** =2048
Default size of the hash table.

Private Attributes

- bool * **_isTaken**
Pointer to taken array Exists parallel to the table and informs whether the object in that position in the table is valid.
- dataType * **_table**
Point to the table Contains the actual data for the hash table.
- size_t **_size**
Size of the hash table.
- size_t **_numElements**
Holds the number of elements.
- size_t(&) **_hashFunction** (const dataType &) throw ()
Reference to the hash function This function is used to convert an object to a position in the hash table.
- bool(&) **_compareFunction** (const dataType &, const dataType &) throw ()
Reference to the comparison function This function is used to compare two objects in the table.

12.31.1 Detailed Description

```
template<class dataType>
class os::simpleHash< dataType >
```

Basic hash function.

This class defines a very simple hash structure to be used by classes which may require use of a hash table.

12.31.2 Constructor & Destructor Documentation

```
template<class dataType> os::simpleHash< dataType >::simpleHash ( size_t(&)(const dataType
&) throw hashFunc() = simpleHash<dataType>::defaultHashFunction, bool(&)(const dataType
&, const dataType &) throw compFunc() = simpleHash<dataType>::defaultCompareFunction,
size_t sz = DEFAULT_SIZE ) throw descriptiveException) [inline]
```

Simple hash constructor Instantiates the simple hash constructor with a hash function and size.

Parameters

in	<i>hashFunc</i>	os::simpleHash<dataType>::defaultHashFunction (p. 253) by default
in	<i>compFunc</i>	os::simpleCompare<dataType>::defaultCompareFunction by default
in	<i>sz</i>	DEFAULT_SIZE by default

```
template<class dataType> virtual os::simpleHash< dataType >::~simpleHash ( ) [inline],  
[virtual]
```

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.31.3 Member Function Documentation

```
template<class dataType> dataType& os::simpleHash< dataType >::at ( size_t pos ) throw  
descriptiveException) [inline]
```

Access element by index.

Parameters

in	<i>pos</i>	Position to access
----	------------	--------------------

Returns

Reference to object at position

```
template<class dataType> const dataType& os::simpleHash< dataType >::at ( size_t pos ) const  
throw descriptiveException) [inline]
```

Const access element by index.

Parameters

in	<i>pos</i>	Position to access
----	------------	--------------------

Returns

Constant reference to object at position

```
template<class dataType> bool os::simpleHash< dataType >::atPosition ( size_t pos ) const  
throw ) [inline]
```

Check position in the table.

Parameters

in	<i>pos</i>	Position to check
----	------------	-------------------

Returns

true if occupied, else, false

```
template<class dataType> bool os::simpleHash< dataType >::check ( size_t pos ) const throw )  
[inline]
```

Check position in the table.

Parameters

in	<i>pos</i>	Position to check
----	------------	-------------------

Returns

true if occupied, else, false

```
template<class dataType> static bool os::simpleHash< dataType >::defaultCompareFunction (   
const dataType & dt1, const dataType & dt2 ) throw ) [inline], [static]
```

Default compare function.

Compares two objects using the defined equality operator. Note that for some data types, this may need to be re-defined.

Parameters

in	<i>dt1</i>	Data to be compared
in	<i>dt2</i>	Data to be compared

Returns

True if equal, else, false

```
template<class dataType> static size_t os::simpleHash< dataType >::defaultHashFunction (   
const dataType & dtp ) throw ) [inline], [static]
```

Default hash function.

Casts an object to `size_t`. This allows the hash table to use the size representation of an element to place it in the hash table.

Parameters

in	<i>dt</i>	Data type to be cast
----	-----------	----------------------

Returns

Data type cast to size

```
template<class dataType> void os::simpleHash< dataType >::empty ( ) [inline]
```

Clear the hash table.

Returns

void

```
template<class dataType> bool os::simpleHash< dataType >::exists ( const dataType & dtp )  
const throw ) [inline]
```

Check if data exists in the table.

This function should have $O(1)$, although, if the table is sufficiently full, this function approaches $O(n)$ where n is the size of the table.

Parameters

in	<i>dtp</i>	Data to be checked
----	------------	--------------------

Returns

True if found, else, false

```
template<class dataType> size_t os::simpleHash< dataType >::find ( const dataType & dtp )  
const throw ) [inline]
```

Find data in the table.

This function should have $O(1)$, although, if the table is sufficiently full, this function approaches $O(n)$ where n is the size of the table.

Parameters

in	<i>dtp</i>	Data to be checked
----	------------	--------------------

Returns

Position in the hash

```
template<class dataType> size_t os::simpleHash< dataType >::hashFunction ( const dataType &  
dtp ) throw ) [inline]
```

Call the hash function.

Parameters

in	<i>dtp</i>	Data type to be hashed
----	------------	------------------------

Returns

`simpleHash::_hashFunction(dtp)`

```
template<class dataType> dataType& os::simpleHash< dataType >::insert ( const dataType & dtp  
) throw descriptiveException() [inline]
```

Insert data into the table.

This function should have $O(1)$, although, if the table is sufficiently full, this function approaches $O(n)$ where n is the size of the table.

Parameters

in	<i>dtp</i>	Data to be inserted
----	------------	---------------------

Returns

`void`

```
template<class dataType> size_t os::simpleHash< dataType >::numElements ( ) const throw ( )  
[inline]
```

Return the number of elements in the structure.

Returns

simpleHash::_numElements (p.257)

```
template<class dataType> dataType& os::simpleHash< dataType >::operator[] ( size_t pos )  
throw descriptiveException() [inline]
```

Access element by index.

Parameters

in	<i>pos</i>	Position to access
----	------------	--------------------

Returns

Reference to object at position

```
template<class dataType> const dataType& os::simpleHash< dataType >::operator[] ( size_t pos  
) const throw descriptiveException() [inline]
```

Const access element by index.

Parameters

in	<i>pos</i>	Position to access
----	------------	--------------------

Returns

Constant reference to object at position

```
template<class dataType> bool os::simpleHash< dataType >::remove ( const dataType & dtp )  
[inline]
```

Remove data from the table.

This function should have $O(1)$, although, if the table is sufficiently full, this function approaches $O(n)$ where n is the size of the table.

Parameters

in	<i>dtp</i>	Data to be removed
----	------------	--------------------

Returns

True if deleted, else, false

```
template<class dataType> void os::simpleHash< dataType >::setSize ( size_t sz ) [inline]
```

Attempts to set the size of the hash table.

Sets the size of the hash table and re-inserts all data. Note that this function will fail if the size is out of bounds or if the new size cannot fit all of the old data.

Parameters

in	<i>sz</i>	Target size
----	-----------	-------------

Returns

void

```
template<class dataType> size_t os::simpleHash< dataType >::size ( ) const throw ) [inline]
```

Return size of hash table.

Returns

simpleHash::_size (p. 257)

12.31.4 Member Data Documentation

```
template<class dataType> bool(& os::simpleHash< dataType >::_compareFunction(const  
dataType &, const dataType &) throw ) [private]
```

Reference to the comparison function This function is used to compare two objects in the table.

```
template<class dataType> size_t(&) os::simpleHash< dataType >::_hashFunction(const dataType
&) throw ) [private]
```

Reference to the hash function This function is used to convert an object to a position in the hash table.

```
template<class dataType> bool* os::simpleHash< dataType >::_isTaken [private]
```

Pointer to taken array Exists parallel to the table and informs whether the object in that position in the table is valid.

```
template<class dataType> size_t os::simpleHash< dataType >::_numElements [private]
```

Holds the number of elements.

```
template<class dataType> size_t os::simpleHash< dataType >::_size [private]
```

Size of the hash table.

```
template<class dataType> dataType* os::simpleHash< dataType >::_table [private]
```

Point to the table Contains the actual data for the hash table.

```
template<class dataType> const size_t os::simpleHash< dataType >::DEFAULT_SIZE =2048
[static]
```

Default size of the hash table.

12.32 os::smart_ptr< dataType > Class Template Reference

Reference counted pointer.

Public Member Functions

- **smart_ptr** () throw ()
Default constructor.
- **smart_ptr** (const **smart_pointer_type** t, const std::atomic< unsigned long > *rc, const dataType *rp, const **void_rec** f) throw ()
Forced constructor.
- **smart_ptr** (const **smart_ptr**< dataType > &sp) throw ()
Copy constructor.
- **smart_ptr** (const dataType *rp, **smart_pointer_type** typ=**raw_type**) throw ()
Standard constructor.
- **smart_ptr** (const dataType *rp, const **void_rec** destructor) throw ()
Dynamic deletion constructor.
- virtual ~**smart_ptr** ()
Virtual destructor.

- **smart_ptr** (const int rp) throw ()
Integer constructor.
- **smart_ptr** (const long rp) throw ()
Long constructor.
- **smart_ptr** (const unsigned long rp) throw ()
Unsigned long constructor.
- **smart_pointer_type** **getType** () const throw ()
Return type.
- dataType * **get** () throw ()
Return data.
- const dataType * **get** () const throw ()
Return constant data.
- const dataType * **constGet** () const throw ()
Return constant data.
- const std::atomic< unsigned long > * **getRefCount** () const throw ()
Return constant reference count.
- **void_rec** **getFunc** () const throw ()
Return deletion function.
- bool **operator!** () const throw ()
Inverted boolean conversion.
- **operator bool** () const throw ()
Boolean conversion.
- **operator size_t** () const throw ()
size_t conversion
- **operator long** () const throw ()
long conversion
- dataType & **operator*** () throw ()
De-reference conversion.
- const dataType & **operator*** () const throw ()
Constant de-reference conversion.
- dataType * **operator->** () throw ()
Pointer pass.
- const dataType * **operator->** () const throw ()
Constant pointer pass.
- dataType & **operator[]** (unsigned i) throw ()
Array de-reference.
- const dataType & **operator[]** (unsigned i) const throw ()
Constant array de-reference.
- **smart_ptr**< dataType > & **bind** (**smart_ptr**< dataType > sp) throw ()
Bind copy.
- **smart_ptr**< dataType > & **bind** (const dataType *rp) throw ()
Bind raw copy.
- **smart_ptr**< dataType > & **operator=** (const **smart_ptr**< dataType > source) throw ()

Equals copy.

- **smart_ptr**< dataType > & **operator=** (const dataType *source) throw ()

Bind raw copy.

- **smart_ptr**< dataType > & **operator=** (const int source) throw ()

Bind integer copy.

- **smart_ptr**< dataType > & **operator=** (const long source) throw ()

Bind long copy.

- **smart_ptr**< dataType > & **operator=** (const unsigned long source) throw ()

Bind unsigned long copy.

- int **compare** (const **smart_ptr**< dataType > &c) const throw ()

*Compare **os::smart_ptr** (p. 257).*

- int **compare** (const dataType *c) const throw ()

Compare raw pointers.

- int **compare** (const unsigned long c) const throw ()

Compare cast long.

Private Member Functions

- void **teardown** () throw (descriptiveException)

Delete data.

Private Attributes

- **smart_pointer_type** type

Stores the type.

- std::atomic< unsigned long > * **ref_count**

Reference count.

- dataType * **raw_ptr**

Pointer to data.

- **void_rec** func

Non-standard deletion.

12.32.1 Detailed Description

```
template<class dataType>
class os::smart_ptr< dataType >
```

Reference counted pointer.

The **os::smart_ptr** (p. 257) template class allows for automatic memory management. **os::smart_ptr** (p. 257)'s have a type defined by **os::smart_pointer_type** (p. 100) which defines the copy and deletion behaviour of the object.

12.32.2 Constructor & Destructor Documentation

```
template<class dataType> os::smart_ptr< dataType >::smart_ptr ( ) throw ) [inline]
```

Default constructor.

Constructs an **os::smart_ptr** (p. 257) of type **os::null_type** (p. 100). All private data is set to 0 or NULL.

```
template<class dataType> os::smart_ptr< dataType >::smart_ptr ( const smart_pointer_type  
t, const std::atomic< unsigned long > * rc, const dataType * rp, const void_rec f ) throw )  
[inline]
```

Forced constructor.

Constructs an **os::smart_ptr** (p. 257) explicitly from each of the parameters provided. This constructor is primarily used for testing purposes.

Parameters

in	<i>t</i>	Type definition for the object
in,out	<i>rp</i>	Pointer to the reference count
in	<i>rp</i>	Raw pointer object is managing
in	<i>f</i>	Dynamic deletion function

```
template<class dataType> os::smart_ptr< dataType >::smart_ptr ( const smart_ptr< dataType >  
& sp ) throw ) [inline]
```

Copy constructor.

Constructs an **os::smart_ptr** (p. 257) from an existing **os::smart_ptr** (p. 257). Will increment the reference count as defined by the received **os::smart_pointer_type** (p. 100).

Parameters

in,out	<i>sp</i>	Reference to data being copied
--------	-----------	--------------------------------

```
template<class dataType> os::smart_ptr< dataType >::smart_ptr ( const dataType * rp,  
smart_pointer_type typ = raw_type ) throw ) [inline]
```

Standard constructor.

Constructs an **os::smart_ptr** (p. 257) from a raw pointer and a type. This is the most commonly used **os::smart_ptr** (p. 257) constructor, other than the copy constructor. Note that **os::shared_type_dynamic_delete** (p. 101) cannot be constructed through this method.

Parameters

in	<i>rp</i>	Raw pointer object is managing
in	<i>typ</i>	Defines reference count behaviour

```
template<class dataType> os::smart_ptr< dataType >::smart_ptr ( const dataType * rp, const
void_rec destructor ) throw ) [inline]
```

Dynamic deletion constructor.

Constructs an **os::smart_ptr** (p. 257) from a raw pointer and a destruction function. This constructor generates an **os::smart_ptr** (p. 257) of type **os::shared_type_dynamic_delete** (p. 101).

Parameters

in	<i>rp</i>	Raw pointer object is managing
in	<i>destructor</i>	Defines the function to be executed on destroy

```
template<class dataType> virtual os::smart_ptr< dataType >::~smart_ptr ( ) [inline],
[virtual]
```

Virtual destructor.

Calls **os::smart_ptr**<**dataType**>::**teardown()** (p. 268) before destroying the object.

```
template<class dataType> os::smart_ptr< dataType >::smart_ptr ( const int rp ) throw )
[inline]
```

Integer constructor.

Constructs an **os::smart_ptr** (p. 257) from an integer. The assumption is that this integer is 0 (or NULL). This function is still legal if the integer is not NULL, this allows for casting, although such usage is discouraged.

Parameters

in	<i>rp</i>	Integer cast to raw pointer
----	-----------	-----------------------------

```
template<class dataType> os::smart_ptr< dataType >::smart_ptr ( const long rp ) throw )
[inline]
```

Long constructor.

Constructs an **os::smart_ptr** (p. 257) from a long. The assumption is that this long is 0 (or NULL). This function is still legal if the long is not NULL, this allows for casting, although such usage is discouraged.

Parameters

in	<i>rp</i>	Long cast to raw pointer
----	-----------	--------------------------

```
template<class dataType> os::smart_ptr< dataType >::smart_ptr ( const unsigned long rp )  
throw ) [inline]
```

Unsigned long constructor.

Constructs an **os::smart_ptr** (p. 257) from an unsigned long. The assumption is that this unsigned long is 0 (or NULL). This function is still legal if the unsigned long is not NULL, this allows for casting, although such usage is discouraged.

Parameters

in	<i>rp</i>	Unsigned long cast to raw pointer
----	-----------	-----------------------------------

12.32.3 Member Function Documentation

```
template<class dataType> smart_ptr<dataType>& os::smart_ptr< dataType >::bind ( smart_ptr< dataType > sp ) throw ) [inline]
```

Bind copy.

Binds to an **os::smart_ptr** (p. 257) from an existing **os::smart_ptr** (p. 257). Will increment the reference count as defined by the received **os::smart_pointer_type** (p. 100).

Parameters

in	<i>sp</i>	Reference to data being copied
----	-----------	--------------------------------

Returns

Reference to self

```
template<class dataType> smart_ptr<dataType>& os::smart_ptr< dataType >::bind ( const  
dataType * rp ) throw ) [inline]
```

Bind raw copy.

Binds to an **os::smart_ptr** (p. 257) from a dataType pointer. This new **os::smart_ptr** (p. 257) will be of type **os::raw_type** (p. 100) unless the dataType pointer is NULL, then it will be of type **os::null_type** (p. 100).

Parameters

in	<i>rp</i>	Reference to dataType pointer
----	-----------	-------------------------------

Returns

Reference to self

```
template<class dataType> int os::smart_ptr< dataType >::compare ( const smart_ptr< dataType  
> & c ) const throw ) [inline]
```

Compare **os::smart_ptr** (p. 257).

Compares two pointers to the same type by address and returns the result in the form of a 1,0 or -1. Note that the **os::smart_ptr<dataType>::type** (p. 269) of the objects does not factor into this comparison.

Parameters

in	c	os::smart_ptr<dataType>
----	---	-------------------------

Returns

1, 0, -1 (Greater than, equal to, less than)

```
template<class dataType> int os::smart_ptr< dataType >::compare ( const dataType * c ) const  
throw ) [inline]
```

Compare raw pointers.

Compares a **os::smart_ptr<dataType>** and a raw pointer of type **dataType** and returns the result in the form of a 1,0 or -1.

Parameters

in	c	Raw dataType pointer
----	---	----------------------

Returns

1, 0, -1 (Greater than, equal to, less than)

```
template<class dataType> int os::smart_ptr< dataType >::compare ( const unsigned long c )  
const throw ) [inline]
```

Compare cast long.

Compares a **os::smart_ptr<dataType>** and an unsigned long, returning the result in the form of a 1,0 or -1.

Parameters

in	c	Unsigned long cast to dataType pointer
----	---	--

Returns

1, 0, -1 (Greater than, equal to, less than)

```
template<class dataType> const dataType* os::smart_ptr< dataType >::constGet ( ) const throw  
) [inline]
```

Return constant data.

Returns the constant dataType pointer of the **os::smart_ptr** (p. 257).

Returns

dataType* in constant form, **os::smart_ptr**<dataType>::raw_ptr (p. 268)

```
template<class dataType> dataType* os::smart_ptr< dataType >::get ( ) throw ) [inline]
```

Return data.

Returns the dataType pointer of the **os::smart_ptr** (p. 257).

Returns

dataType* in modifiable form, **os::smart_ptr**<dataType>::raw_ptr (p. 268)

```
template<class dataType> const dataType* os::smart_ptr< dataType >::get ( ) const throw )  
[inline]
```

Return constant data.

Returns the constant dataType pointer of the **os::smart_ptr** (p. 257).

Returns

dataType* in constant form, **os::smart_ptr**<dataType>::raw_ptr (p. 268)

```
template<class dataType> void_rec os::smart_ptr< dataType >::getFunc ( ) const throw )  
[inline]
```

Return deletion function.

Returns the deletion function if it exists. (Note that the deletion function only exists in **os::shared_type_dynamic_delete** (p. 101) mode)

Returns

os::void_rec (p. 100) **os::smart_ptr**<dataType>::func (p. 268)

```
template<class dataType> const std::atomic<unsigned long>* os::smart_ptr< dataType  
>::getRefCount ( ) const throw ) [inline]
```

Return constant reference count.

Returns a constant pointer of the reference count.

Returns

unsigned long* in constant form, **os::smart_ptr**<dataType>::ref_count (p. 269)

```
template<class dataType> smart_pointer_type os::smart_ptr< dataType >::getType ( ) const  
throw ) [inline]
```

Return type.

Returns the **os::smart_pointer_type** (p. 100) of the **os::smart_ptr** (p. 257).

Returns

os::smart_pointer_type (p. 100) **os::smart_ptr**<**dataType**>::**type** (p. 269)

```
template<class dataType> os::smart_ptr< dataType >::operator bool ( ) const throw )  
[inline]
```

Boolean conversion.

Returns

os::smart_ptr<**dataType**>::**raw_ptr** (p. 268) cast to boolean

```
template<class dataType> os::smart_ptr< dataType >::operator long ( ) const throw )  
[inline]
```

long conversion

Returns

os::smart_ptr<**dataType**>::**raw_ptr** (p. 268) cast to long

```
template<class dataType> os::smart_ptr< dataType >::operator size_t ( ) const throw )  
[inline]
```

size_t conversion

Returns

os::smart_ptr<**dataType**>::**raw_ptr** (p. 268) cast to size_t

```
template<class dataType> bool os::smart_ptr< dataType >::operator! ( ) const throw )  
[inline]
```

Inverted boolean conversion.

Returns

Inverse of **os::smart_ptr**<**dataType**>::**raw_ptr** (p. 268) cast to boolean

```
template<class dataType> dataType& os::smart_ptr< dataType >::operator* ( ) throw )  
[inline]
```

De-reference conversion.

Returns

dataType reference of **os::smart_ptr**<**dataType**>::**raw_ptr** (p. 268) de-referenced

```
template<class dataType> const dataType& os::smart_ptr< dataType >::operator* ( ) const
throw ) [inline]
```

Constant de-reference conversion.

Returns

Constant dataType reference of **os::smart_ptr**<dataType>::raw_ptr (p. 268) de-referenced

```
template<class dataType> dataType* os::smart_ptr< dataType >::operator-> ( ) throw )
[inline]
```

Pointer pass.

Returns

os::smart_ptr<dataType>::raw_ptr (p. 268)

```
template<class dataType> const dataType* os::smart_ptr< dataType >::operator-> ( ) const
throw ) [inline]
```

Constant pointer pass.

Returns

Constant **os::smart_ptr**<dataType>::raw_ptr (p. 268)

```
template<class dataType> smart_ptr<dataType>& os::smart_ptr< dataType >::operator= ( const
smart_ptr< dataType > source ) throw ) [inline]
```

Equals copy.

Calls **os::smart_ptr**<dataType>::bind (p. 262).

Parameters

in	source	Reference to data being copied
----	--------	--------------------------------

Returns

Reference to self

```
template<class dataType> smart_ptr<dataType>& os::smart_ptr< dataType >::operator= ( const
dataType * source ) throw ) [inline]
```

Bind raw copy.

Calls **os::smart_ptr**<dataType>::bind (p. 262).

Parameters

in	source	Reference to dataType pointer
----	--------	-------------------------------

Returns

Reference to self

```
template<class dataType> smart_ptr<dataType>& os::smart_ptr< dataType >::operator= ( const  
int source ) throw ) [inline]
```

Bind integer copy.

Calls **os::smart_ptr<dataType>::bind** (p. 262) with the integer cast to a dataType pointer.

Parameters

in	source	Integer cast to raw pointer
----	--------	-----------------------------

Returns

Reference to self

```
template<class dataType> smart_ptr<dataType>& os::smart_ptr< dataType >::operator= ( const  
long source ) throw ) [inline]
```

Bind long copy.

Calls **os::smart_ptr<dataType>::bind** (p. 262) with the long cast to a dataType pointer.

Parameters

in	source	Long cast to raw pointer
----	--------	--------------------------

Returns

Reference to self

```
template<class dataType> smart_ptr<dataType>& os::smart_ptr< dataType >::operator= ( const  
unsigned long source ) throw ) [inline]
```

Bind unsigned long copy.

Calls **os::smart_ptr<dataType>::bind** (p. 262) with the unsigned long cast to a dataType pointer.

Parameters

in	source	Unsigned long cast to raw pointer
----	--------	-----------------------------------

Returns

Reference to self

```
template<class dataType> dataType& os::smart_ptr< dataType >::operator[] ( unsigned i ) throw  
 ) [inline]
```

Array de-reference.

Returns

dataType reference of **os::smart_ptr**<dataType>::raw_ptr (p. 268) incremented i de-referenced

```
template<class dataType> const dataType& os::smart_ptr< dataType >::operator[] ( unsigned i )  
const throw ) [inline]
```

Constant array de-reference.

Returns

Constant dataType reference of **os::smart_ptr**<dataType>::raw_ptr (p. 268) incremented i de-referenced

```
template<class dataType> void os::smart_ptr< dataType >::teardown ( ) throw  
descriptiveException) [inline], [private]
```

Delete data.

Tears down the **os::smart_ptr** (p. 257). Decrements the reference counter, if not of **os::raw_ptr** (p. 100) or **os::null_type** (p. 100), and delete **os::smart_ptr**<dataType>::raw_ptr (p. 268) if needed. Note that if **os::smart_ptr**<dataType>::raw_ptr (p. 268) is deleted, so is **os::smart_ptr**<dataType>::ref_count (p. 269).

Returns

void

12.32.4 Member Data Documentation

```
template<class dataType> void_rec os::smart_ptr< dataType >::func [private]
```

Non-standard deletion.

This is a pointer to a function used when the **os::smart_ptr** (p. 257) is of type **os::shared_type** or **os::dynamic_delete** (p. 101).

```
template<class dataType> dataType* os::smart_ptr< dataType >::raw_ptr [private]
```

Pointer to data.

The **os::smart_ptr**<dataType>::raw_ptr (p. 268) holds the pointer to the block of memory to be managed by the **os::smart_ptr** (p. 257). If this pointer is NULL, the **os::smart_ptr** (p. 257) is of type **os::null_type** (p. 100).

```
template<class dataType> std::atomic<unsigned long>* os::smart_ptr< dataType >::ref_count  
[private]
```

Reference count.

This pointer stores the current reference count of the **os::smart_ptr** (p. 257). Note that all **os::smart_ptr** (p. 257)'s which point to the same memory address with share the same reference counter. This counter is deleted with the pointer and if this counter is NULL, the **os::smart_ptr** (p. 257) is either of type **os::null_type** (p. 100) or **os::raw_type** (p. 100).

```
template<class dataType> smart_pointer_type os::smart_ptr< dataType >::type [private]
```

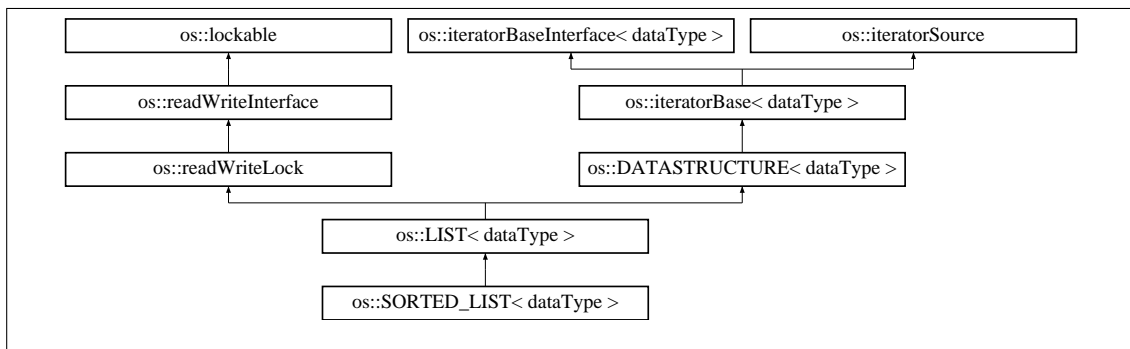
Stores the type.

Defines the type of the **os::smart_ptr** (p. 257). See **os::smart_pointer_type** (p. 100) for details on the available types.

12.33 os::SORTED_LIST< dataType > Class Template Reference

Sorted list A basic list which remains unsorted unless the sort function is called on it.

Inheritance diagram for **os::SORTED_LIST< dataType >**:



Public Member Functions

- **bool insert** (const dataType &x) final
Insert element into the list.

Static Public Attributes

- static const bool **ITERABLE** = true
Lists are iterable.
- static const bool **RANDOM_ACCESS** = false
Lists do not allow random access.

Additional Inherited Members

12.33.1 Detailed Description

```
template<class dataType>
class os::SORTED_LIST< dataType >
```

Sorted list A basic list which remains unsorted unless the sort function is called on it.

12.33.2 Member Function Documentation

```
template<class dataType > bool os::SORTED_LIST< dataType >::insert ( const dataType & x )
[inline], [final], [virtual]
```

Insert element into the list.

Parameters

in	x	Data type to be inserted
----	---	--------------------------

Returns

true if inserted, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 159).

12.33.3 Member Data Documentation

```
template<class dataType > const bool os::SORTED_LIST< dataType >::ITERABLE = true
[static]
```

Lists are iterable.

```
template<class dataType > const bool os::SORTED_LIST< dataType >::RANDOM_ACCESS =
false [static]
```

Lists do not allow random access.

12.34 os::threadCounter Class Reference

Thread counter.

Public Member Functions

- **threadCounter** (std::thread::id thr=std::this_thread::get_id()) throw ()
Thread ID constructor.
- **threadCounter** (const **threadCounter** &tcnt) throw ()
- std::thread::id **id** () const throw ()
Access thread ID.

- **size_t count ()** const throw ()
Access thread counter.
- **operator size_t ()** const
*Cast object to size_t Used to cast the **threadCounter** (p. 270) to a size_t for use in a simple hash table.*
- **threadCounter & operator++ ()** throw (descriptiveException)
Increment operator.
- **threadCounter operator++ (int dummy)** throw (descriptiveException)
Increment operator.
- **threadCounter & operator-- ()** throw (descriptiveException)
Decrement operator.
- **threadCounter operator-- (int dummy)** throw (descriptiveException)
Decrement operator.
- **int compare (const threadCounter &tcnt)** const
Compare function.

Static Public Attributes

- static std::hash< std::thread::id > **hashFunction**
Thread ID hash function.

Private Attributes

- std::thread::id **_thread**
Thread ID.
- size_t **_count**
Thread counter.

12.34.1 Detailed Description

Thread counter.

The thread counter is used by various mutexes to keep track of the thread ID's of threads which have acquired a lock.

12.34.2 Constructor & Destructor Documentation

os::threadCounter::threadCounter (std::thread::id thr = std::this_thread::get_id()) throw ()

Thread ID constructor.

Parameters

in	<i>thr</i>	Thread ID to initialize with
----	------------	------------------------------

`os::threadCounter::threadCounter (const threadCounter & tcnt) throw)`

Parameters

<code>in</code>	<code>tcnt</code>	Object to copy
-----------------	-------------------	----------------

12.34.3 Member Function Documentation

`int os::threadCounter::compare (const threadCounter & tcnt) const [inline]`

Compare function.

This comparison function is used in the comparison operators defined in the `COMPARE_OPERATORS` macro

Parameters

<code>in</code>	<code>tcnt</code>	Thread counter to compare against
-----------------	-------------------	-----------------------------------

Returns

Integer representing comparison

`size_t os::threadCounter::count () const throw) [inline]`

Access thread counter.

Returns

threadCounter::_count (p. 273)

`std::thread::id os::threadCounter::id () const throw) [inline]`

Access thread ID.

Returns

threadCounter::_thread (p. 273)

`os::threadCounter::operator size_t () const [inline]`

Cast object to `size_t` Used to cast the **threadCounter** (p. 270) to a `size_t` for use in a simple hash table.

`threadCounter& os::threadCounter::operator++ () throw descriptiveException)`

Increment operator.

Returns

`++this`

threadCounter os::threadCounter::operator++ (int dummy) throw **descriptiveException**)

Increment operator.

Returns

this++

threadCounter& os::threadCounter::operator-- () throw **descriptiveException**)

Decrement operator.

Returns

–this

threadCounter os::threadCounter::operator-- (int dummy) throw **descriptiveException**)

Decrement operator.

Returns

this–

12.34.4 Member Data Documentation

size_t os::threadCounter::_count [private]

Thread counter.

std::thread::id os::threadCounter::_thread [private]

Thread ID.

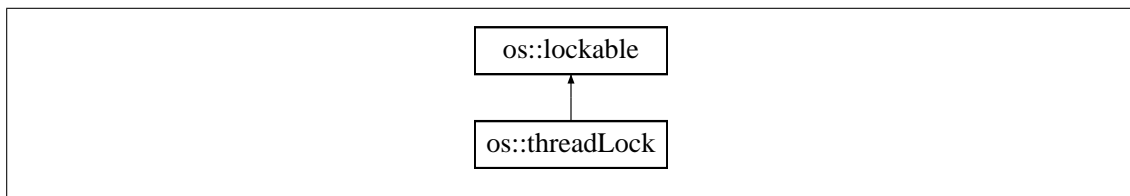
std::hash<std::thread::id> os::threadCounter::hashFunction [static]

Thread ID hash function.

12.35 os::threadLock Class Reference

Thread lock Wraps the std::recursive_mutex class allowing it to be accessed in a constant way. This thread will allow a lock to be called multiple times in a single thread.

Inheritance diagram for os::threadLock:



Public Member Functions

- **threadLock** (bool dMode=**NO_LOCK_CHECK**) throw ()
Default constructor.
- virtual **~threadLock** () throw (descriptiveException)
Virtual destructor.
- void **lock** () const final throw ()
Locks the std::recursive_mutex.
- void **unlock** () const final throw (descriptiveException)
Unlocks the std::recursive_mutex.
- bool **locked** () const final throw ()
Checks if the lock is locked.
- bool **try_lock** () const final throw ()
Attempt to lock the object.

Private Member Functions

- **threadLock** (const **threadLock** &cpy)
Undefined copy-constructor.

Private Attributes

- std::recursive_mutex **_mtx**
Mutable recursive mutex.
- size_t **_lockedStatus**
Locked flag.

Additional Inherited Members

12.35.1 Detailed Description

Thread lock Wraps the std::recursive_mutex class allowing it to be accessed in a constant way. This thread will allow a lock to be called multiple times in a single thread.

12.35.2 Constructor & Destructor Documentation

os::threadLock::threadLock (const **threadLock** & cpy) [inline], [private]

Undefined copy-constructor.

os::threadLock::threadLock (bool dMode = **NO_LOCK_CHECK**) throw ()

Default constructor.

virtual os::threadLock::~threadLock () throw **descriptiveException**) [virtual]

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.35.3 Member Function Documentation

void os::threadLock::lock () const throw) [final], [virtual]

Locks the std::recursive_mutex.

Returns

void

Implements **os::lockable** (p. 223).

bool os::threadLock::locked () const throw) [inline], [final], [virtual]

Checks if the lock is locked.

Returns

True if locked, else, false

Implements **os::lockable** (p. 223).

bool os::threadLock::try_lock () const throw) [final], [virtual]

Attempt to lock the object.

Locks the object if possible, otherwise, returns false.

Returns

True if lock successful

Implements **os::lockable** (p. 223).

void os::threadLock::unlock () const throw **descriptiveException**) [final], [virtual]

Unlocks the std::recursive_mutex.

Returns

void

Implements **os::lockable** (p. 223).

12.35.4 Member Data Documentation

size_t os::threadLock::_lockedStatus [mutable], [private]

Locked flag.

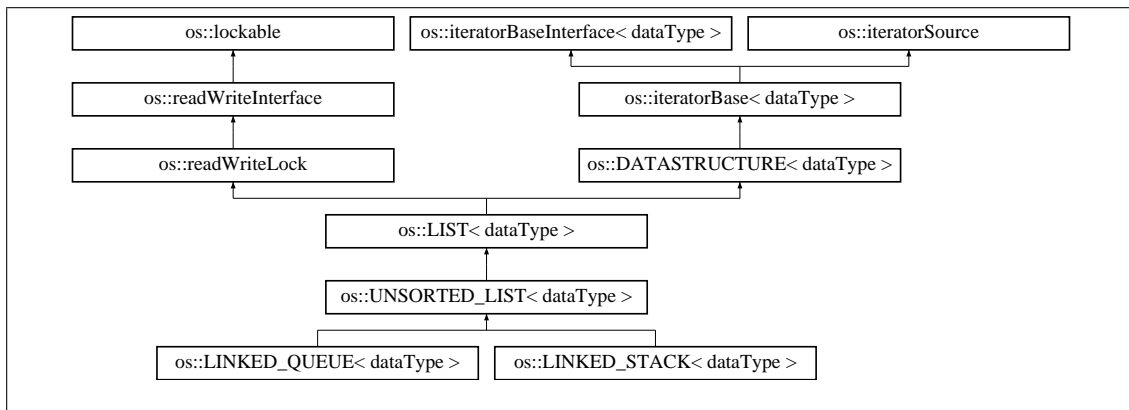
std::recursive_mutex os::threadLock::_mtx [mutable], [private]

Mutable recursive mutex.

12.36 os::UNSORTED_LIST< dataType > Class Template Reference

Unsorted list A basic list which remains unsorted unless the sort function is called on it.

Inheritance diagram for os::UNSORTED_LIST< dataType >:



Public Member Functions

- **bool insert** (const dataType &x) final
Insert element into the list.

Static Public Attributes

- static const bool **ITERABLE** = true
Lists are iterable.
- static const bool **RANDOM_ACCESS** = false
Lists do not allow random access.

Additional Inherited Members

12.36.1 Detailed Description

```
template<class dataType>
class os::UNSORTED_LIST< dataType >
```

Unsorted list A basic list which remains unsorted unless the sort function is called on it.

12.36.2 Member Function Documentation

```
template<class dataType > bool os::UNSORTED_LIST< dataType >::insert ( const dataType & x )  
[inline], [final], [virtual]
```

Insert element into the list.

Parameters

in	x	Data type to be inserted
----	---	--------------------------

Returns

true if inserted, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 159).

12.36.3 Member Data Documentation

```
template<class dataType > const bool os::UNSORTED_LIST< dataType >::ITERABLE = true  
[static]
```

Lists are iterable.

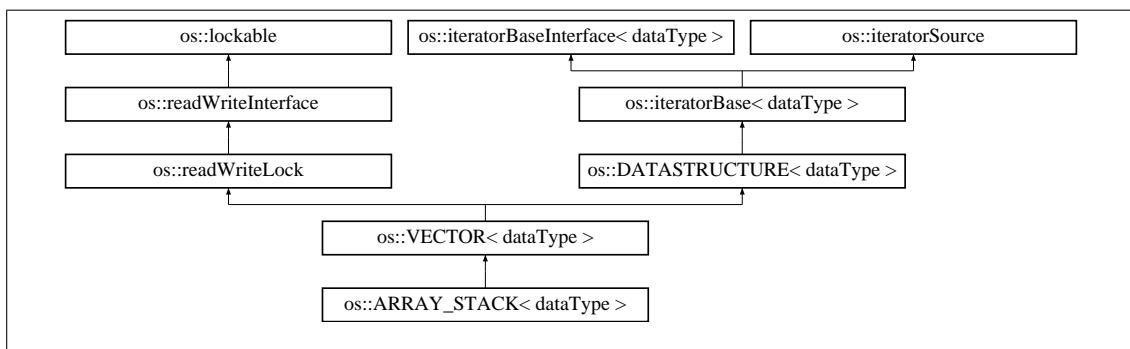
```
template<class dataType > const bool os::UNSORTED_LIST< dataType >::RANDOM_ACCESS =  
false [static]
```

Lists do not allow random access.

12.37 os::VECTOR< dataType > Class Template Reference

Template vector definition.

Inheritance diagram for **os::VECTOR**< **dataType** >:



Public Member Functions

- **VECTOR** (size_t sz=0)
Default constructor.
- virtual **~VECTOR** ()
Vector destructor Destroys all objects held in the array. Note that in the case the vector is holding references, the pointers will be destroyed instead of the objects.
- bool **insert** (const dataType &x) final
Insert item into the vector.
- bool **remove** (const dataType &x) final
Remove item from the data-structure.
- bool **find** (const dataType &x) const final
Searches for an item.
- dataType & **access** (const dataType &x) final
Mutable item access.
- const dataType & **access** (const dataType &x) const final
Immutable item access.
- dataType & **at** (size_t i) final throw (descriptiveException)
Access the vector by index.
- const dataType & **at** (size_t i) const final throw (descriptiveException)
Access the vector by index.
- void **sort** (int(*sort_comparison)(const dataType &, const dataType &)=&defaultCompare< dataType >)
Sorts the vector.
- size_t **size** () const final
Access size of the vector.
- bool **iterable** () const final
Returns if the relevant node type is iterable.
- bool **randomAccess** () const final
Returns if the relevant node type can be accessed randomly.

Static Public Attributes

- static const bool **ITERABLE** = true
Vectors are iterable.
- static const bool **RANDOM_ACCESS** = true
Vectors allow random access.

Protected Member Functions

- **smart_ptr< nodeFrame< dataType > > getFirstNode** () final
Access to first node.
- **smart_ptr< nodeFrame< dataType > > getLastNode** () final
Access to last node.

- **const smart_ptr< nodeFrame< dataType > > getFirstNodeConst ()** const final
Constant access to first node.
- **const smart_ptr< nodeFrame< dataType > > getLastNodeConst ()** const final
Constant access to last node.
- **smart_ptr< nodeFrame< dataType > > searchNode** (const smart_ptr< dataType > dt) final
Search for a node.
- **const smart_ptr< nodeFrame< dataType > > searchNodeConst** (const smart_ptr< dataType > dt) const final
Const search for a node.

Private Attributes

- **dataType * _array**
Pointer to array of data.
- **size_t _arraySize**
Size of the available memory.
- **size_t vecSize**
Number of elements in the vector.

Additional Inherited Members

12.37.1 Detailed Description

```
template<class dataType>
class os::VECTOR< dataType >
```

Template vector definition.

Note that there are 6 different versions of this class defined, allowing for multiple pointer and thread-safety definitions.

12.37.2 Constructor & Destructor Documentation

```
template<class dataType > os::VECTOR< dataType >::VECTOR ( size_t sz = 0 ) [inline]
```

Default constructor.

This constructor builds the vector with a certain size. Note that the vector will always be of, at least, size 16.

Parameters

in	sz	Target size of vector
----	----	-----------------------


```
template<class dataType > virtual os::VECTOR< dataType >::~VECTOR ( ) [inline],  
[virtual]
```

Vector destructor Destroys all objects held in the array. Note that in the case the vector is holding references, the pointers will be destroyed instead of the objects.

12.37.3 Member Function Documentation

```
template<class dataType > dataType& os::VECTOR< dataType >::access ( const dataType & x )  
[inline], [final], [virtual]
```

Mutable item access.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Returns

Mutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 157).

```
template<class dataType > const dataType& os::VECTOR< dataType >::access ( const dataType  
& x ) const [inline], [final], [virtual]
```

Immutable item access.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Returns

Immutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 157).

```
template<class dataType > dataType& os::VECTOR< dataType >::at ( size_t i ) throw  
descriptiveException) [inline], [final], [virtual]
```

Access the vector by index.

Parameters

in	<i>i</i>	Index of the array
----	----------	--------------------

Returns

Reference to ith element of the vector

Reimplemented from **os::DATASTRUCTURE**< **dataType** > (p. 158).

```
template<class dataType > const dataType& os::VECTOR< dataType >::at ( size_t i ) const throw
descriptiveException) [inline], [final], [virtual]
```

Access the vector by index.

Parameters

in	<i>i</i>	Index of the array
-----------	----------	--------------------

Returns

Immutable reference to ith element of the vector

Reimplemented from **os::DATASTRUCTURE**< **dataType** > (p. 158).

```
template<class dataType > bool os::VECTOR< dataType >::find ( const dataType & x ) const
[inline], [final], [virtual]
```

Searches for an item.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references.

Returns

True if found, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 158).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::VECTOR< dataType
>::getFirstNode ( ) [final], [protected], [virtual]
```

Access to first node.

Returns

First node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 198).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::VECTOR< dataType
>::getFirstNodeConst ( ) const [final], [protected], [virtual]
```

Constant access to first node.

Returns

Immutable first node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 198).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::VECTOR< dataType  
>::getLastNode ( ) [final], [protected], [virtual]
```

Access to last node.

Returns

Last node in the structure

Implements **os::iteratorBase< dataType >** (p. 198).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::VECTOR< dataType  
>::getLastNodeConst ( ) const [final], [protected], [virtual]
```

Constant access to last node.

Returns

Immutable last node in the structure

Implements **os::iteratorBase< dataType >** (p. 199).

```
template<class dataType > bool os::VECTOR< dataType >::insert ( const dataType & x )  
[inline], [final], [virtual]
```

Insert item into the vector.

Inserts an item at the end of the vector. Note that different forms of the vector accept pointers instead of object references.

Returns

True

Implements **os::DATASTRUCTURE< dataType >** (p. 159).

```
template<class dataType > bool os::VECTOR< dataType >::iterable ( ) const [inline],  
[final], [virtual]
```

Returns if the relevant node type is iterable.

Returns

true

Reimplemented from **os::iteratorSource** (p. 205).

```
template<class dataType > bool os::VECTOR< dataType >::randomAccess ( ) const [inline],  
[final], [virtual]
```

Returns if the relevant node type can be accessed randomly.

Returns

true

Reimplemented from **os::iteratorSource** (p. 205).

```
template<class dataType > bool os::VECTOR< dataType >::remove ( const dataType & x )
[inline], [final], [virtual]
```

Remove item from the data-structure.

Each data-structure must re-define this function. Note that different forms of the datastructure accept pointers instead of object references.

Returns

True if removed, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 160).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::VECTOR< dataType
>::searchNode ( const smart_ptr< dataType > dt ) [final], [protected], [virtual]
```

Search for a node.

Parameters

in	<i>dt</i>	Pointer to search for
-----------	-----------	-----------------------

Returns

Mutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 199).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::VECTOR< dataType
>::searchNodeConst ( const smart_ptr< dataType > dt ) const [final], [protected],
[virtual]
```

Const search for a node.

Parameters

in	<i>dt</i>	Pointer to search for
-----------	-----------	-----------------------

Returns

Immutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 200).

```
template<class dataType > size_t os::VECTOR< dataType >::size ( ) const [inline], [final],
[virtual]
```

Access size of the vector.

Returns

Number of elements in the vector

Implements **os::DATASTRUCTURE**< **dataType** > (p. 160).

```
template<class dataType > void os::VECTOR< dataType >::sort ( int(*) (const dataType &, const
dataType &) sort_comparison = &defaultCompare<dataType> ) [inline]
```

Sorts the vector.

Parameters

in	<i>sort_comparison</i>	Comparison function
----	------------------------	---------------------

Returns

void

12.37.4 Member Data Documentation

```
template<class dataType > dataType* os::VECTOR< dataType >::_array [private]
```

Pointer to array of data.

```
template<class dataType > size_t os::VECTOR< dataType >::_arraySize [private]
```

Size of the available memory.

```
template<class dataType > const bool os::VECTOR< dataType >::ITERABLE = true [static]
```

Vectors are iterable.

```
template<class dataType > const bool os::VECTOR< dataType >::RANDOM_ACCESS = true
[static]
```

Vectors allow random access.

```
template<class dataType > size_t os::VECTOR< dataType >::vecSize [private]
```

Number of elements in the vector.

12.38 **os::vector2d**< dataType > Class Template Reference

2-dimensional vector

Public Member Functions

- **vector2d** ()
Default constructor.
- **vector2d** (dataType xv, dataType yv)
Value constructor.
- **vector2d** (const **vector2d**< dataType > &vec)
Copy constructor.

- **vector2d**< dataType > & **operator=** (const **vector2d**< dataType > &vec)
Equality constructor.
- **vector2d**< dataType > & **operator()** (const dataType &X, const dataType &Y)
Value setter.
- virtual ~**vector2d** () throw ()
Virtual destructor s Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.*
- dataType **length** () const
Return length of the vector.
- **vector2d**< dataType > & **scaleSelf** (dataType target=1)
Scales this vector.
- **vector2d**< dataType > **scale** (dataType target=1) const
Return a scaled vector.
- int **compare** (const **vector2d**< dataType > &vec) const
- bool **operator==** (const **vector2d**< dataType > &vec) const
Equality comparison operator.
- bool **operator!=** (const **vector2d**< dataType > &vec) const
Not-equals comparison operator.
- bool **operator<** (const **vector2d**< dataType > &vec) const
Less-than comparison operator.
- bool **operator<=** (const **vector2d**< dataType > &vec) const
Less-than or equals to comparison operator.
- bool **operator>** (const **vector2d**< dataType > &vec) const
Less-than comparison operator.
- bool **operator>=** (const **vector2d**< dataType > &vec) const
- **vector2d**< dataType > & **addSelf** (const **vector2d**< dataType > &vec)
Add vector to self.
- **vector2d**< dataType > **add** (const **vector2d**< dataType > &vec) const
Add two vectors.
- **vector2d**< dataType > **operator+** (const **vector2d**< dataType > &vec) const
Add two vectors.
- **vector2d**< dataType > & **operator+=** (const **vector2d**< dataType > &vec)
Add vector to self.
- **vector2d**< dataType > **operator++** ()
Increment.
- **vector2d**< dataType > & **operator++** (int dummy)
Increment.
- **vector2d**< dataType > **operator-** () const
Invert vector.
- **vector2d**< dataType > & **subtractSelf** (const **vector2d**< dataType > &vec)
Subtract vector from self.
- **vector2d**< dataType > **subtract** (const **vector2d**< dataType > &vec) const
Subtract two vectors.

- **vector2d**< dataType > **operator-** (const **vector2d**< dataType > &vec) const
Subtracts two vectors.
- **vector2d**< dataType > & **operator-=** (const **vector2d**< dataType > &vec)
Subtracts vector from self.
- **vector2d**< dataType > **operator--** ()
Decrement.
- **vector2d**< dataType > & **operator--** (int dummy)
Decrement.
- dataType **dotProduct** (const **vector2d**< dataType > &vec) const
Dot-product.
- **vector2d**< dataType > **rotate** (const **vector2d**< dataType > &vec) const
Rotates a point around 0, 0.
- **vector2d**< dataType > **rotateSelf** (const **vector2d**< dataType > &vec)
Rotates self around 0, 0.

Public Attributes

- dataType **x**
X axis vector component.
- dataType **y**
Y axis vector component.

12.38.1 Detailed Description

```
template<class dataType>
class os::vector2d< dataType >
```

2-dimensional vector

This template class contains the functions and operators needed to preform arithmetic on a 2 dimensional vector

12.38.2 Constructor & Destructor Documentation

```
template<class dataType> os::vector2d< dataType >::vector2d ( ) [inline]
```

Default constructor.

Constructs a 2 dimensional vector with x and y as 0.

```
template<class dataType> os::vector2d< dataType >::vector2d ( dataType xv, dataType yv )
[inline]
```

Value constructor.

Constructs a 2 dimensional vector with a x and a y value.

Parameters

in	xv	Value of x dimension
in	yv	Value of y dimension

```
template<class dataType> os::vector2d< dataType >::vector2d ( const vector2d< dataType > &
vec ) [inline]
```

Copy constructor.

Constructs a 2 dimensional vector from a 2 dimensional vector

Parameters

in	vec	Vector to be copied
----	-----	---------------------

```
template<class dataType> virtual os::vector2d< dataType >::~~vector2d ( ) throw ) [inline],
[virtual]
```

Virtual destructor s* Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.38.3 Member Function Documentation

```
template<class dataType> vector2d<dataType> os::vector2d< dataType >::add ( const
vector2d< dataType > & vec ) const [inline]
```

Add two vectors.

Adds the provided vector to the current vector and returns a new vector. This function is essentially the function version of the '+' operator.

Parameters

in	vec	Reference to vector to be added
----	-----	---------------------------------

Returns

Result of the vector addition

```
template<class dataType> vector2d<dataType>& os::vector2d< dataType >::addSelf ( const
vector2d< dataType > & vec ) [inline]
```

Add vector to self.

Adds the provided vector to the current vector. This function is essentially the function version of the '+=' operator.

Parameters

in	vec	Reference to vector to be added
----	-----	---------------------------------

Returns

Reference to self

```
template<class dataType> int os::vector2d< dataType >::compare ( const vector2d< dataType >
& vec ) const [inline]
```

Compares two vectors

This function compares two vectors for equality. It does not change either vector. This function returns 1 if this object is greater than the object reference received, 0 if the two are equal and -1 if the received reference is greater than the object.

Parameters

in	vec	Reference to object compared against
----	-----	--------------------------------------

Returns

1 if greater than, 0 if equal to, -1 if less than

```
template<class dataType> dataType os::vector2d< dataType >::dotProduct ( const vector2d<
dataType > & vec ) const [inline]
```

Dot-product.

Calculates the scalar dot-product. Note that this function does not return a vector, but rather, returns a scalar.

Parameters

in	vec	Reference to vector
----	-----	---------------------

Returns

Scalar dot product

```
template<class dataType> dataType os::vector2d< dataType >::length ( ) const [inline]
```

Return length of the vector.

Returns $\sqrt{x^2+y^2}$, or the length of the vector.

Returns

Length of the vector

```
template<class dataType> bool os::vector2d< dataType >::operator!= ( const vector2d<
dataType > & vec ) const [inline]
```

Not-equals comparison operator.

Parameters

in	vec	Reference to object compared against
----	-----	--------------------------------------

Returns

true if vectors are not equal

```
template<class dataType> vector2d<dataType>& os::vector2d< dataType >::operator() ( const  
dataType & X, const dataType & Y ) [inline]
```

Value setter.

Sets the values of a 2 dimensional vector with a x and a y value.

Parameters

in	X	Value of x dimension
in	Y	Value of y dimension

Returns

Reference to this vector

```
template<class dataType> vector2d<dataType> os::vector2d< dataType >::operator+ ( const  
vector2d< dataType > & vec ) const [inline]
```

Add two vectors.

Parameters

in	vec	Reference to vector to be added
----	-----	---------------------------------

Returns

vector2d<dataType>::add(vec)

```
template<class dataType> vector2d<dataType> os::vector2d< dataType >::operator++ ( )  
[inline]
```

Increment.

Increments this vector by the unit vector of the same direction and then returns a reference to this vector.

Returns

Reference to self

```
template<class dataType> vector2d<dataType>& os::vector2d< dataType >::operator++ ( int dummy ) [inline]
```

Increment.

Copies this vector then increments this vector by the unit vector of the same direction and then returns the original copy.

Parameters

in	<i>dummy</i>	Parameter required to define operator
----	--------------	---------------------------------------

Returns

Original copy

```
template<class dataType> vector2d<dataType>& os::vector2d< dataType >::operator+= ( const vector2d< dataType > & vec ) [inline]
```

Add vector to self.

Parameters

in	<i>vec</i>	Reference to vector to be added
----	------------	---------------------------------

Returns

vector3d<dataType>::addSelf(*vec*)

```
template<class dataType> vector2d<dataType> os::vector2d< dataType >::operator- ( ) const [inline]
```

Invert vector.

Constructs a new vector with an inverted x and inverted y.

Returns

Inverted vector

```
template<class dataType> vector2d<dataType> os::vector2d< dataType >::operator- ( const vector2d< dataType > & vec ) const [inline]
```

Subtracts two vectors.

Parameters

in	<i>vec</i>	Reference to vector to be subtracted
----	------------	--------------------------------------

Returns

`vector2d<dataType>::subtract(vec)`

```
template<class dataType> vector2d<dataType> os::vector2d< dataType >::operator-- ( )  
[inline]
```

Decrement.

Decrements this vector by the unit vector of the same direction and then returns a reference to this vector.

Returns

Reference to self

```
template<class dataType> vector2d<dataType>& os::vector2d< dataType >::operator-- ( int  
dummy ) [inline]
```

Decrement.

Copies this vector then decrements this vector by the unit vector of the same direction and then returns the original copy.

Parameters

in	<i>dummy</i>	Parameter required to define operator
----	--------------	---------------------------------------

Returns

Original copy

```
template<class dataType> vector2d<dataType>& os::vector2d< dataType >::operator-= ( const  
vector2d< dataType > & vec ) [inline]
```

Subtracts vector from self.

Parameters

in	<i>vec</i>	Reference to vector to be subtracted
----	------------	--------------------------------------

Returns

`vector3d<dataType>::subtractSelf(vec)`

```
template<class dataType> bool os::vector2d< dataType >::operator< ( const vector2d< dataType  
> & vec ) const [inline]
```

Less-than comparison operator.

Parameters

in	<i>vec</i>	Reference to object compared against
----	------------	--------------------------------------

Returns

true if this is less than vec

```
template<class dataType> bool os::vector2d< dataType >::operator<= ( const vector2d<
dataType > & vec ) const [inline]
```

Less-than or equals to comparison operator.

Parameters

in	vec	Reference to object compared against
----	-----	--------------------------------------

Returns

true if this is less than vec

```
template<class dataType> vector2d<dataType>& os::vector2d< dataType >::operator= ( const
vector2d< dataType > & vec ) [inline]
```

Equality constructor.

Set the values of a 2 dimensional vector from a another 2 dimensional vector

Parameters

in	vec	Vector to be copied
----	-----	---------------------

Returns

Reference to this vector

```
template<class dataType> bool os::vector2d< dataType >::operator== ( const vector2d<
dataType > & vec ) const [inline]
```

Equality comparison operator.

Parameters

in	vec	Reference to object compared against
----	-----	--------------------------------------

Returns

true if vectors are equal

```
template<class dataType> bool os::vector2d< dataType >::operator> ( const vector2d< dataType
> & vec ) const [inline]
```

Less-than comparison operator.

Parameters

in	vec	Reference to object compared against
----	-----	--------------------------------------

Returns

true if this is less than vec

```
template<class dataType> bool os::vector2d< dataType >::operator>= ( const vector2d<
dataType > & vec ) const [inline]
```

```
template<class dataType> vector2d<dataType> os::vector2d< dataType >::rotate ( const
vector2d< dataType > & vec ) const [inline]
```

Rotates a point around 0, 0.

Parameters

in	vec	Vector representing an angle
----	-----	------------------------------

Returns

Rotated point

```
template<class dataType> vector2d<dataType> os::vector2d< dataType >::rotateSelf ( const
vector2d< dataType > & vec ) [inline]
```

Rotates self around 0, 0.

Parameters

in	vec	Vector representing an angle
----	-----	------------------------------

Returns

Rotated point

```
template<class dataType> vector2d<dataType> os::vector2d< dataType >::scale ( dataType
target = 1 ) const [inline]
```

Return a scaled vector.

Returns a vector scaled to the given target length. This operation, by default, will scale to a distance of 1 (the unit vector)

Parameters

in	target	Vector length to be scaled to
----	--------	-------------------------------

Returns

The scaled vector

```
template<class dataType> vector2d<dataType>& os::vector2d< dataType >::scaleSelf (
dataType target = 1 ) [inline]
```

Scales this vector.

Scales this vector to the given target length. This operation, by default, will scale to a distance of 1 (the unit vector)

Parameters

in	<i>target</i>	Vector length to be scaled to
----	---------------	-------------------------------

Returns

Reference to this

```
template<class dataType> vector2d<dataType> os::vector2d< dataType >::subtract ( const
vector2d< dataType > & vec ) const [inline]
```

Subtract two vectors.

Subtracts the provided vector from the current vector and returns a new vector. This function is essentially the function version of the '-' operator.

Parameters

in	<i>vec</i>	Reference to vector to be subtracted
----	------------	--------------------------------------

Returns

Result of the vector subtraction

```
template<class dataType> vector2d<dataType>& os::vector2d< dataType >::subtractSelf ( const
vector2d< dataType > & vec ) [inline]
```

Subtract vector from self.

Subtracts the provided vector from the current vector. This function is essentially the function version of the '-=' operator.

Parameters

in	<i>vec</i>	Reference to vector to be subtracted
----	------------	--------------------------------------

Returns

Reference to self

12.38.4 Member Data Documentation

template<class dataType> dataType **os::vector2d**< dataType >::x

X axis vector component.

template<class dataType> dataType **os::vector2d**< dataType >::y

Y axis vector component.

12.39 os::vector3d< dataType > Class Template Reference

3-dimensional vector

Public Member Functions

- **vector3d** ()
Default constructor.
- **vector3d** (dataType xv, dataType yv, dataType zv=0)
Value constructor.
- **vector3d** (const **vector3d**< dataType > &vec)
Copy constructor.
- **vector3d** (const **vector2d**< dataType > &vec)
Copy constructor.
- **vector3d**< dataType > & **operator=** (const **vector3d**< dataType > &vec)
Equality constructor.
- **vector3d**< dataType > & **operator()** (const dataType &X, const dataType &Y, const dataType &Z)
Value setter.
- virtual ~**vector3d** () throw ()
Virtual destructor s. Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.*
- dataType **length** () const
Return length of the vector.
- **vector3d**< dataType > & **scaleSelf** (dataType target=1)
Scales this vector.
- **vector3d**< dataType > **scale** (dataType target=1) const
Return a scaled vector.
- int **compare** (const **vector3d** &vec) const
- bool **operator==** (const **vector3d**< dataType > &vec) const
Equality comparison operator.
- bool **operator!=** (const **vector3d**< dataType > &vec) const

- *Not-equals comparison operator.*
- **bool operator<** (const **vector3d**< dataType > &vec) const
Less-than comparison operator.
- **bool operator<=** (const **vector3d**< dataType > &vec) const
Less-than or equal to comparison operator.
- **bool operator>** (const **vector3d**< dataType > &vec) const
Greater-than comparison operator.
- **bool operator>=** (const **vector3d**< dataType > &vec) const
Greater-than or equal to comparison operator.
- **vector3d**< dataType > & **addSelf** (const **vector3d**< dataType > &vec)
Add vector to self.
- **vector3d**< dataType > **add** (const **vector3d**< dataType > &vec) const
Add two vectors.
- **vector3d**< dataType > **operator+** (const **vector3d**< dataType > &vec) const
Add two vectors.
- **vector3d**< dataType > & **operator+=** (const **vector3d**< dataType > &vec)
Add vector to self.
- **vector3d**< dataType > **operator++** ()
Increment.
- **vector3d**< dataType > & **operator++** (int dummy)
Increment.
- **vector3d**< dataType > **operator-** () const
Invert vector.
- **vector3d**< dataType > & **subtractSelf** (const **vector3d**< dataType > &vec)
Subtract vector from self.
- **vector3d**< dataType > **subtract** (const **vector3d**< dataType > &vec) const
Subtract two vectors.
- **vector3d**< dataType > **operator-** (const **vector3d**< dataType > &vec) const
Subtracts two vectors.
- **vector3d**< dataType > & **operator-=** (const **vector3d**< dataType > &vec)
Subtracts vector from self.
- **vector3d**< dataType > **operator--** ()
Decrement.
- **vector3d**< dataType > & **operator--** (int dummy)
Decrement.
- dataType **dotProduct** (const **vector3d**< dataType > &vec) const
Dot-product.
- **vector3d**< dataType > **crossProduct** (const **vector3d**< dataType > &vec) const
Cross-product.
- **vector3d**< dataType > & **crossSelf** (const **vector3d**< dataType > &vec)
Cross-product to self.
- **vector3d**< dataType > **operator*** (const **vector3d**< dataType > &vec) const
Cross-product.
- **vector3d**< dataType > & **operator*=** (const **vector3d**< dataType > &vec)
Self cross-product.

Public Attributes

- **dataType x**
X axis vector component.
- **dataType y**
Y axis vector component.
- **dataType z**
Z axis vector component.

12.39.1 Detailed Description

```
template<class dataType>
class os::vector3d< dataType >
```

3-dimensional vector

This template class contains the functions and operators needed to perform arithmetic on a 3 dimensional vector

12.39.2 Constructor & Destructor Documentation

```
template<class dataType> os::vector3d< dataType >::vector3d ( ) [inline]
```

Default constructor.

Constructs a 3 dimensional vector with x, y and z as 0.

```
template<class dataType> os::vector3d< dataType >::vector3d ( dataType xv, dataType yv,
dataType zv = 0 ) [inline]
```

Value constructor.

Constructs a 3 dimensional vector with x, y and z values. Z, by default, is initialized as 0.

Parameters

in	xv	Value of x dimension
in	yv	Value of y dimension
in	zv	Value of z dimension

```
template<class dataType> os::vector3d< dataType >::vector3d ( const vector3d< dataType > &
vec ) [inline]
```

Copy constructor.

Constructs a 3 dimensional vector from another 3 dimensional vector

Parameters

in	vec	Vector to be copied
----	-----	---------------------

Returns

Reference to this vector

```
template<class dataType> os::vector3d< dataType >::vector3d ( const vector2d< dataType > &
vec ) [inline]
```

Copy constructor.

Constructs a 3 dimensional vector from a 2 dimensional vector

Parameters

in	vec	Vector to be copied
----	-----	---------------------

Returns

Reference to this vector

```
template<class dataType> virtual os::vector3d< dataType >::~vector3d ( ) throw ) [inline],
[virtual]
```

Virtual destructor s* Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called.

12.39.3 Member Function Documentation

```
template<class dataType> vector3d<dataType> os::vector3d< dataType >::add ( const
vector3d< dataType > & vec ) const [inline]
```

Add two vectors.

Adds the provided vector to the current vector and returns a new vector. This function is essentially the function version of the '+' operator.

Parameters

in	vec	Reference to vector to be added
----	-----	---------------------------------

Returns

Result of the vector addition

```
template<class dataType> vector3d<dataType>& os::vector3d< dataType >::addSelf ( const
vector3d< dataType > & vec ) [inline]
```

Add vector to self.

Adds the provided vector to the current vector. This function is essentially the function version of the '+=' operator.

Parameters

in	vec	Reference to vector to be added
----	-----	---------------------------------

Returns

Reference to self

```
template<class dataType> int os::vector3d< dataType >::compare ( const vector3d< dataType > & vec ) const [inline]
```

Compares two vectors

This function compares two vectors for equality. It does not change either vector. This function returns 1 if this object is greater than the object reference received, 0 if the two are equal and -1 if the received reference is greater than the object.

Parameters

in	vec	Reference to object compared against
----	-----	--------------------------------------

Returns

1 if greater than, 0 if equal to, -1 if less than

```
template<class dataType> vector3d<dataType> os::vector3d< dataType >::crossProduct ( const vector3d< dataType > & vec ) const [inline]
```

Cross-product.

Perform the cross-product computation on this vector and the vector argument provided. Unlike the dot-product, the cross product returns a vector.

Parameters

in	vec	Reference to vector to be computed
----	-----	------------------------------------

Returns

Result of the cross-product

```
template<class dataType> vector3d<dataType>& os::vector3d< dataType >::crossSelf ( const vector3d< dataType > & vec ) [inline]
```

Cross-product to self.

Perform the cross-product computation on this vector and the vector argument provided. Binds the result to this and returns a reference to this vector.

Parameters

in	vec	Reference to vector to be computed
----	-----	------------------------------------

Returns

Reference to self

```
template<class dataType> dataType os::vector3d< dataType >::dotProduct ( const vector3d<
dataType > & vec ) const [inline]
```

Dot-product.

Calculates the scalar dot-product. Note that this function does not return a vector, but rather, returns a scalar.

Parameters

in	vec	Reference to vector
----	-----	---------------------

Returns

Scalar dot product

```
template<class dataType> dataType os::vector3d< dataType >::length ( ) const [inline]
```

Return length of the vector.

Returns $\sqrt{x^2+y^2+z^2}$, or the length of the vector.

Returns

Length of the vector

```
template<class dataType> bool os::vector3d< dataType >::operator!= ( const vector3d<
dataType > & vec ) const [inline]
```

Not-equals comparison operator.

Parameters

in	vec	Reference to object compared against
----	-----	--------------------------------------

Returns

true if vectors are not equal

```
template<class dataType> vector3d<dataType>& os::vector3d< dataType >::operator() ( const
dataType & X, const dataType & Y, const dataType & Z ) [inline]
```

Value setter.

Sets values of a 3 dimensional vector with x, y and z values.

Parameters

in	X	Value of x dimension
----	---	----------------------

Parameters

in	Y	Value of y dimension
in	Z	Value of z dimension

Returns

Reference to this vector

```
template<class dataType> vector3d<dataType> os::vector3d< dataType >::operator* ( const vector3d< dataType > & vec ) const [inline]
```

Cross-product.

Parameters

in	vec	Reference to vector to be computed with
----	-----	---

Returns

vector3d<dataType>::crossProduct(vec)

```
template<class dataType> vector3d<dataType>& os::vector3d< dataType >::operator*=( const vector3d< dataType > & vec ) [inline]
```

Self cross-product.

Parameters

in	vec	Reference to vector to be computed with
----	-----	---

Returns

vector3d<dataType>::crossSelf(vec)

```
template<class dataType> vector3d<dataType> os::vector3d< dataType >::operator+ ( const vector3d< dataType > & vec ) const [inline]
```

Add two vectors.

Parameters

in	vec	Reference to vector to be added
----	-----	---------------------------------

Returns

vector3d<dataType>::add(vec)

```
template<class dataType> vector3d<dataType> os::vector3d< dataType >::operator++ ( )  
[inline]
```

Increment.

Increments this vector by the unit vector of the same direction and then returns a reference to this vector.

Returns

Reference to self

```
template<class dataType> vector3d<dataType>& os::vector3d< dataType >::operator++ ( int  
dummy ) [inline]
```

Increment.

Copies this vector then increments this vector by the unit vector of the same direction and then returns the original copy.

Parameters

in	<i>dummy</i>	Parameter required to define operator
----	--------------	---------------------------------------

Returns

Original copy

```
template<class dataType> vector3d<dataType>& os::vector3d< dataType >::operator+= ( const  
vector3d< dataType > & vec ) [inline]
```

Add vector to self.

Parameters

in	<i>vec</i>	Reference to vector to be added
----	------------	---------------------------------

Returns

vector3d<dataType>::addSelf(vec)

```
template<class dataType> vector3d<dataType> os::vector3d< dataType >::operator- ( ) const  
[inline]
```

Invert vector.

Constructs a new vector with an inverted x, inverted y and inverted z.

Returns

Inverted vector

```
template<class dataType> vector3d<dataType> os::vector3d< dataType >::operator- ( const vector3d< dataType > & vec ) const [inline]
```

Subtracts two vectors.

Parameters

in	vec	Reference to vector to be subtracted
----	-----	--------------------------------------

Returns

vector3d<dataType>::subtract(vec)

```
template<class dataType> vector3d<dataType> os::vector3d< dataType >::operator-- ( ) [inline]
```

Decrement.

Decrements this vector by the unit vector of the same direction and then returns a reference to this vector.

Returns

Reference to self

```
template<class dataType> vector3d<dataType>& os::vector3d< dataType >::operator-- ( int dummy ) [inline]
```

Decrement.

Copies this vector then decrements this vector by the unit vector of the same direction and then returns the original copy.

Parameters

in	<i>dummy</i>	Parameter required to define operator
----	--------------	---------------------------------------

Returns

Original copy

```
template<class dataType> vector3d<dataType>& os::vector3d< dataType >::operator-= ( const vector3d< dataType > & vec ) [inline]
```

Subtracts vector from self.

Parameters

in	vec	Reference to vector to be subtracted
----	-----	--------------------------------------

Returns

`vector3d<dataType>::subtractSelf(vec)`

```
template<class dataType> bool os::vector3d< dataType >::operator< ( const vector3d< dataType  
> & vec ) const [inline]
```

Less-than comparison operator.

Parameters

in	vec	Reference to object compared against
----	-----	--------------------------------------

Returns

true if this is less than vec

```
template<class dataType> bool os::vector3d< dataType >::operator<= ( const vector3d<  
dataType > & vec ) const [inline]
```

Less-than or equal to comparison operator.

Parameters

in	vec	Reference to object compared against
----	-----	--------------------------------------

Returns

true if this is less than or equal to vec

```
template<class dataType> vector3d<dataType>& os::vector3d< dataType >::operator= ( const  
vector3d< dataType > & vec ) [inline]
```

Equality constructor.

Set the values of a 3 dimensional vector from a another 3 dimensional vector

Parameters

in	vec	Vector to be copied
----	-----	---------------------

Returns

Reference to this vector

```
template<class dataType> bool os::vector3d< dataType >::operator== ( const vector3d<  
dataType > & vec ) const [inline]
```

Equality comparison operator.

Parameters

in	vec	Reference to object compared against
----	-----	--------------------------------------

Returns

true if vectors are equal

```
template<class dataType> bool os::vector3d< dataType >::operator> ( const vector3d< dataType  
> & vec ) const [inline]
```

Greater-than comparison operator.

Parameters

in	vec	Reference to object compared against
----	-----	--------------------------------------

Returns

true if this is greater than vec

```
template<class dataType> bool os::vector3d< dataType >::operator>= ( const vector3d<  
dataType > & vec ) const [inline]
```

Greater-than or equal to comparison operator.

Parameters

in	vec	Reference to object compared against
----	-----	--------------------------------------

Returns

true if this is greater than or equal to vec

```
template<class dataType> vector3d<dataType> os::vector3d< dataType >::scale ( dataType  
target = 1 ) const [inline]
```

Return a scaled vector.

Returns a vector scaled to the given target length. This operation, by default, will scale to a distance of 1 (the unit vector)

Parameters

in	<i>target</i>	Vector length to be scaled to
----	---------------	-------------------------------

Returns

The scaled vector

```
template<class dataType> vector3d<dataType>& os::vector3d< dataType >::scaleSelf (
dataType target = 1 ) [inline]
```

Scales this vector.

Scales this vector to the given target length. This operation, by default, will scale to a distance of 1 (the unit vector)

Parameters

in	<i>target</i>	Vector length to be scaled to
----	---------------	-------------------------------

Returns

Reference to this

```
template<class dataType> vector3d<dataType> os::vector3d< dataType >::subtract ( const
vector3d< dataType > & vec ) const [inline]
```

Subtract two vectors.

Subtracts the provided vector to the current vector and returns a new vector. This function is essentially the function version of the '-' operator.

Parameters

in	<i>vec</i>	Reference to vector to be subtracted
----	------------	--------------------------------------

Returns

Result of the vector subtraction

```
template<class dataType> vector3d<dataType>& os::vector3d< dataType >::subtractSelf ( const
vector3d< dataType > & vec ) [inline]
```

Subtract vector from self.

Subtracts the provided vector from the current vector. This function is essentially the function version of the '-=' operator.

Parameters

in	<i>vec</i>	Reference to vector to be subtracted
----	------------	--------------------------------------

Returns

Reference to self

12.39.4 Member Data Documentation

template<class dataType> dataType **os::vector3d**< dataType >::x

X axis vector component.

template<class dataType> dataType **os::vector3d**< dataType >::y

Y axis vector component.

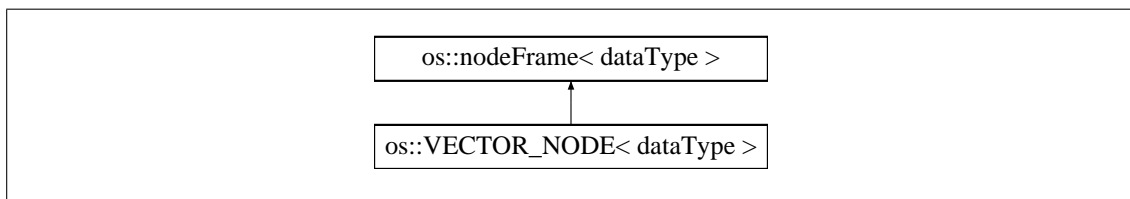
template<class dataType> dataType **os::vector3d**< dataType >::z

Z axis vector component.

12.40 os::VECTOR_NODE< dataType > Class Template Reference

Vector node.

Inheritance diagram for os::VECTOR_NODE< dataType >:



Public Member Functions

- **~VECTOR_NODE** ()
Destructor Class is not designed to be inherited from.
- **smart_ptr**< dataType > **get** () final throw ()
Returns a pointer.
- const **smart_ptr**< dataType > **constGet** () const final throw ()
Returns a const pointer.
- dataType & **operator*** () final throw (descriptiveException)
De-reference.
- const dataType & **operator*** () const final throw (descriptiveException)
De-reference.
- bool **valid** () const final
Valid data query Checks if the provided data is valid.
- bool **iterable** () const final
Returns if the node is iterable.

- **bool randomAccess ()** const final
Returns if the node can be accessed randomly.
- **smart_ptr< nodeFrame< dataType > > getNext ()** final throw (descriptiveException)
Returns the next vector frame.
- **const smart_ptr< nodeFrame< dataType > > getNextConst ()** const final throw (descriptiveException)
Returns the next vector frame.
- **smart_ptr< nodeFrame< dataType > > getPrev ()** final throw (descriptiveException)
Returns the previous vector frame.
- **const smart_ptr< nodeFrame< dataType > > getPrevConst ()** const final throw (descriptiveException)
Returns the previous vector frame.
- **smart_ptr< nodeFrame< dataType > > access (long offset)** final throw (descriptiveException)
Access node by index.
- **const smart_ptr< nodeFrame< dataType > > constAccess (long offset)** const final throw (descriptiveException)
Access node by index.
- **void remove ()** final throw (descriptiveException)
Remove this node from the vector.

Static Public Attributes

- **static const bool ITERABLE = true**
Vector frames are iterable.
- **static const bool RANDOM_ACCESS = true**
Vector frames allow random-access.

Private Member Functions

- **VECTOR_NODE (DRIVER_CLASS *src, size_t pos)**
Private constructor This constructor is not designed to be accessed by anything other than the VECTOR (p. 277) class derivatives.

Private Attributes

- **size_t position**
Current position of the vector iterator.

12.40.1 Detailed Description

```
template<class dataType>
class os::VECTOR_NODE< dataType >
```

Vector node.

Used by the iterator to iterate through a vector.

12.40.2 Constructor & Destructor Documentation

```
template<class dataType > os::VECTOR_NODE< dataType >::VECTOR_NODE (  
DRIVER_CLASS * src, size_t pos ) [inline], [private]
```

Private constructor This constructor is not designed to be accessed by anything other than the **VECTOR** (p. 277) class derivatives.

```
template<class dataType > os::VECTOR_NODE< dataType >::~VECTOR_NODE ( ) [inline]
```

Destructor Class is not designed to be inherited from.

12.40.3 Member Function Documentation

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::VECTOR_NODE< dataType  
>::access ( long offset ) throw descriptiveException) [inline], [final], [virtual]
```

Access node by index.

Access a node offset from the current node by some value. If a node cannot be randomly accessed, an exception will be thrown.

Returns

Offset node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 236).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::VECTOR_NODE<  
dataType >::constAccess ( long offset ) const throw descriptiveException) [inline], [final],  
[virtual]
```

Access node by index.

Access a node offset from the current node by some value. If a node cannot be randomly accessed, an exception will be thrown.

Returns

Offset node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 236).

```
template<class dataType > const smart_ptr<dataType> os::VECTOR_NODE< dataType  
>::constGet ( ) const throw ) [inline], [final], [virtual]
```

Returns a const pointer.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

Implements **os::nodeFrame**< **dataType** > (p. 236).

```
template<class dataType > smart_ptr<dataType> os::VECTOR_NODE< dataType >::get ( )  
throw ) [inline], [final], [virtual]
```

Returns a pointer.

Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

Implements **os::nodeFrame**< **dataType** > (p. 237).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::VECTOR_NODE< dataType  
>::getNext ( ) throw descriptiveException) [inline], [final], [virtual]
```

Returns the next vector frame.

Returns

Next node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 237).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::VECTOR_NODE<  
dataType >::getNextConst ( ) const throw descriptiveException) [inline], [final],  
[virtual]
```

Returns the next vector frame.

Returns

Next node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 238).

```
template<class dataType > smart_ptr<nodeFrame<dataType> > os::VECTOR_NODE< dataType  
>::getPrev ( ) throw descriptiveException) [inline], [final], [virtual]
```

Returns the previous vector frame.

Returns

Previous node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 238).

```
template<class dataType > const smart_ptr<nodeFrame<dataType> > os::VECTOR_NODE<  
dataType >::getPrevConst ( ) const throw descriptiveException) [inline], [final],  
[virtual]
```

Returns the previous vector frame.

Returns

Previous node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 238).

```
template<class dataType > bool os::VECTOR_NODE< dataType >::iterable ( ) const [inline],  
[final], [virtual]
```

Returns if the node is iterable.

Returns

object::ITERABLE

Reimplemented from **os::nodeFrame**< **dataType** > (p. 239).

```
template<class dataType > dataType& os::VECTOR_NODE< dataType >::operator* ( ) throw  
descriptiveException) [inline], [final], [virtual]
```

De-reference.

Returns a reference to the contained object, the reference can be modified.

Returns

Contained object

Implements **os::nodeFrame**< **dataType** > (p. 239).

```
template<class dataType > const dataType& os::VECTOR_NODE< dataType >::operator* ( )  
const throw descriptiveException) [inline], [final], [virtual]
```

De-reference.

Returns a const reference to the contained object, the reference cannot be modified.

Returns

Contained object

Implements **os::nodeFrame**< **dataType** > (p. 240).

```
template<class dataType > bool os::VECTOR_NODE< dataType >::randomAccess ( ) const  
[inline], [final], [virtual]
```

Returns if the node can be accessed randomly.

Returns

object::RANDOM_ACCESS

Reimplemented from **os::nodeFrame**< **dataType** > (p. 240).

```
template<class dataType > void os::VECTOR_NODE< dataType >::remove ( ) throw  
descriptiveException) [inline], [final], [virtual]
```

Remove this node from the vector.

Returns

void

Reimplemented from **os::nodeFrame**< **dataType** > (p. 240).


```
template<class dataType > bool os::VECTOR_NODE< dataType >::valid ( ) const [inline],  
[final], [virtual]
```

Valid data query Checks if the provided data is valid.

Returns

 true if valid, else, false

Reimplemented from **os::nodeFrame**< **dataType** > (p. 241).

12.40.4 Member Data Documentation

```
template<class dataType > const bool os::VECTOR_NODE< dataType >::ITERABLE = true  
[static]
```

Vector frames are iterable.

```
template<class dataType > size_t os::VECTOR_NODE< dataType >::position [private]
```

Current position of the vector iterator.

```
template<class dataType > const bool os::VECTOR_NODE< dataType >::RANDOM_ACCESS =  
true [static]
```

Vector frames allow random-access.