

Unit Test Documentation

Adrian Bedard

Jonathan Bedard

July 3, 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	5
3.1	DatastructuresTest.h File Reference	5
3.1.1	Detailed Description	5
3.2	DatastructuresTest.cpp File Reference	5
3.2.1	Detailed Description	5
3.3	masterTestHolder.h File Reference	6
3.3.1	Detailed Description	6
3.4	masterTestHolder.cpp File Reference	6
3.4.1	Detailed Description	6
3.5	singleTest.h File Reference	7
3.5.1	Detailed Description	7
3.6	singleTest.cpp File Reference	7
3.6.1	Detailed Description	8
3.7	TestSuite.h File Reference	8
3.7.1	Detailed Description	8
3.8	TestSuite.cpp File Reference	8
3.8.1	Detailed Description	8
3.9	UnitTest.h File Reference	9
3.9.1	Detailed Description	9
3.10	UnitTest.cpp File Reference	9
3.10.1	Detailed Description	9
3.11	UnitTestLog.h File Reference	10
3.12	UnitTestExceptions.h File Reference	10
3.12.1	Detailed Description	11
4	Class Index	12
4.1	Class List	12

5	Namespace Documentation	13
5.1	test Namespace Reference	13
5.1.1	Typedef Documentation	14
5.1.2	Function Documentation	14
5.1.3	Variable Documentation	16
6	Class Documentation	17
6.1	test::generalTestException Class Reference	17
6.1.1	Detailed Description	18
6.1.2	Constructor & Destructor Documentation	18
6.1.3	Member Function Documentation	18
6.1.4	Member Data Documentation	19
6.2	test::libraryTests Class Reference	19
6.2.1	Detailed Description	20
6.2.2	Constructor & Destructor Documentation	21
6.2.3	Member Function Documentation	22
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
6.5	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	31
6.6	test::singleTest Class Reference	31
6.6.1	Detailed Description	32
6.6.2	Constructor & Destructor Documentation	32
6.6.3	Member Function Documentation	33
6.6.4	Member Data Documentation	34
6.7	test::testSuite Class Reference	34
6.7.1	Constructor & Destructor Documentation	35
6.7.2	Member Function Documentation	36
6.7.3	Member Data Documentation	40
6.8	test::unknownException Class Reference	40
6.8.1	Detailed Description	40
6.8.2	Constructor & Destructor Documentation	41
II	Datastructures Library	42
7	Introduction	43
7.1	Unit Testing	43
7.2	Namespace os	43

8	File Index	44
8.1	File List	44
9	File Documentation	46
9.1	Datastructures.h File Reference	46
9.1.1	Detailed Description	46
9.2	abstractSorting.h File Reference	46
9.2.1	Detailed Description	47
9.3	ads.h File Reference	47
9.3.1	Detailed Description	48
9.4	asyncAVL.h File Reference	48
9.4.1	Detailed Description	48
9.5	AVL.h File Reference	49
9.5.1	Detailed Description	49
9.6	eventDriver.h File Reference	49
9.6.1	Detailed Description	50
9.7	eventDriver.cpp File Reference	50
9.7.1	Detailed Description	50
9.8	list.h File Reference	50
9.8.1	Detailed Description	51
9.9	matrix.h File Reference	51
9.9.1	Detailed Description	55
9.9.2	Function Documentation	55
9.10	osLogger.h File Reference	64
9.10.1	Detailed Description	65
9.11	osLogger.cpp File Reference	65
9.11.1	Detailed Description	65
9.12	osVectors.h File Reference	65
9.12.1	Detailed Description	67
9.13	set.h File Reference	67
9.13.1	Detailed Description	67
9.14	smartPointer.h File Reference	68
9.14.1	Detailed Description	71
9.14.2	Function Documentation	72
9.15	staticConstantPrinter.h File Reference	75
9.15.1	Detailed Description	75
9.16	staticConstantPrinter.cpp File Reference	76
9.16.1	Detailed Description	76
10	Class Index	77
10.1	Class List	77
11	Namespace Documentation	80
11.1	os Namespace Reference	80
11.1.1	Typedef Documentation	85
11.1.2	Enumeration Type Documentation	87
11.1.3	Function Documentation	88
11.1.4	Variable Documentation	92

12 Class Documentation	94
12.1 os::adnode< dataType > Class Template Reference	94
12.1.1 Detailed Description	95
12.1.2 Constructor & Destructor Documentation	95
12.1.3 Member Function Documentation	95
12.1.4 Member Data Documentation	96
12.2 os::ads< dataType > Class Template Reference	97
12.2.1 Detailed Description	98
12.2.2 Constructor & Destructor Documentation	98
12.2.3 Member Function Documentation	98
12.2.4 Member Data Documentation	101
12.3 os::asyncAVLNode< dataType > Class Template Reference	101
12.3.1 Detailed Description	102
12.3.2 Constructor & Destructor Documentation	103
12.3.3 Member Function Documentation	103
12.3.4 Friends And Related Function Documentation	106
12.3.5 Member Data Documentation	106
12.4 os::asyncAVLTree< dataType > Class Template Reference	107
12.4.1 Detailed Description	109
12.4.2 Constructor & Destructor Documentation	109
12.4.3 Member Function Documentation	109
12.4.4 Friends And Related Function Documentation	114
12.4.5 Member Data Documentation	115
12.5 os::AVLNode< dataType > Class Template Reference	115
12.5.1 Detailed Description	116
12.5.2 Constructor & Destructor Documentation	117
12.5.3 Member Function Documentation	117
12.5.4 Friends And Related Function Documentation	120
12.5.5 Member Data Documentation	120
12.6 os::AVLTree< dataType > Class Template Reference	121
12.6.1 Detailed Description	122
12.6.2 Constructor & Destructor Documentation	122
12.6.3 Member Function Documentation	123
12.6.4 Member Data Documentation	128
12.7 os::constantPrinter Class Reference	128
12.7.1 Detailed Description	129
12.7.2 Constructor & Destructor Documentation	129
12.7.3 Member Function Documentation	129
12.7.4 Member Data Documentation	132
12.8 os::eventReceiver< senderType > Class Template Reference	132
12.8.1 Detailed Description	133
12.8.2 Constructor & Destructor Documentation	133
12.8.3 Member Function Documentation	133
12.8.4 Friends And Related Function Documentation	134
12.8.5 Member Data Documentation	135
12.9 os::eventSender< receiverType > Class Template Reference	135
12.9.1 Detailed Description	136
12.9.2 Constructor & Destructor Documentation	136
12.9.3 Member Function Documentation	136

12.9.4 Friends And Related Function Documentation	138
12.9.5 Member Data Documentation	138
12.10 <code>bs::indirectMatrix< dataType ></code> Class Template Reference	139
12.10.1 Detailed Description	140
12.10.2 Constructor & Destructor Documentation	140
12.10.3 Member Function Documentation	142
12.10.4 Friends And Related Function Documentation	144
12.10.5 Member Data Documentation	144
12.11 <code>bs::matrix< dataType ></code> Class Template Reference	144
12.11.1 Detailed Description	146
12.11.2 Constructor & Destructor Documentation	146
12.11.3 Member Function Documentation	147
12.11.4 Friends And Related Function Documentation	150
12.11.5 Member Data Documentation	150
12.12 <code>bs::ptrComp</code> Class Reference	150
12.12.1 Detailed Description	151
12.12.2 Constructor & Destructor Documentation	151
12.12.3 Member Function Documentation	151
12.13 <code>bs::smart_ptr< dataType ></code> Class Template Reference	151
12.13.1 Detailed Description	154
12.13.2 Constructor & Destructor Documentation	154
12.13.3 Member Function Documentation	156
12.13.4 Member Data Documentation	162
12.14 <code>bs::smartSet< dataType ></code> Class Template Reference	163
12.14.1 Detailed Description	164
12.14.2 Constructor & Destructor Documentation	164
12.14.3 Member Function Documentation	165
12.14.4 Member Data Documentation	167
12.15 <code>bs::unsortedList< dataType ></code> Class Template Reference	167
12.15.1 Detailed Description	168
12.15.2 Constructor & Destructor Documentation	168
12.15.3 Member Function Documentation	169
12.15.4 Member Data Documentation	171
12.16 <code>bs::unsortedListNode< dataType ></code> Class Template Reference	171
12.16.1 Detailed Description	172
12.16.2 Constructor & Destructor Documentation	172
12.16.3 Member Function Documentation	172
12.16.4 Friends And Related Function Documentation	173
12.16.5 Member Data Documentation	173
12.17 <code>bs::vector2d< dataType ></code> Class Template Reference	174
12.17.1 Detailed Description	176
12.17.2 Constructor & Destructor Documentation	176
12.17.3 Member Function Documentation	176
12.17.4 Member Data Documentation	184
12.18 <code>bs::vector3d< dataType ></code> Class Template Reference	184
12.18.1 Detailed Description	187
12.18.2 Constructor & Destructor Documentation	187
12.18.3 Member Function Documentation	188
12.18.4 Member Data Documentation	197

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	5
DatastructuresTest.h	
Datastructures library test	5
defaultTestInit.cpp	
Default UnitTest initializer function	??
masterTestHolder.cpp	
Library tests, masterTestHolder singleton implementations	6
masterTestHolder.h	
Library tests, masterTestHolder singleton	6
singleTest.cpp	
Single test class implementation	7
singleTest.h	
Single test class	7
TestSuite.cpp	
Single test class	8
TestSuite.h	
Single test class	8
UnitTest.cpp	
Unit Test logging and global functions	9
UnitTest.h	
Unit Test header file	9
UnitTestExceptions.h	
Common exceptions thrown by unit tests	10
UnitTestLog.h	
Unit Test header file	10
UnitTestMain.cpp	
UnitTest entry point	??

Chapter 3

File Documentation

3.1 DatastructuresTest.h File Reference

Datastructures library test.

3.1.1 Detailed Description

Datastructures library test.

Author

Jonathan Bedard

Date

2/4/2016

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.

3.2 DatastructuresTest.cpp File Reference

Datastructures library test implementation.

3.2.1 Detailed Description

Datastructures library test implementation.

Author

Jonathan Bedard

Date

7/2/2016

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.

3.3 masterTestHolder.h File Reference

Library tests, masterTestHolder singleton.

Classes

- class **test::libraryTests**
Library test group.
- class **test::masterTestHolder**
Unit Test singleton.

Namespaces

- **test**

3.3.1 Detailed Description

Library tests, masterTestHolder singleton.

Jonathan Bedard

Date

5/14/2016

Bug No known bugs.

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

3.4 masterTestHolder.cpp File Reference

Library tests, masterTestHolder singleton implementations.

3.4.1 Detailed Description

Library tests, masterTestHolder singleton implementations.

Jonathan Bedard

Date

5/14/2016

Bug No known bugs.

This file contains implementations for the library test base class and **test::masterTestHolder** (p. 26) singleton class. Consult **masterTestHolder.h** (p. 6) for details.

3.5 singleTest.h File Reference

Single test class.

Classes

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

Namespaces

- **test**

Typedefs

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

3.5.1 Detailed Description

Single test class.

Jonathan Bedard

Date

5/31/2016

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.

3.6 singleTest.cpp File Reference

Single test class implementation.

3.6.1 Detailed Description

Single test class implementation.

Jonathan Bedard

Date

5/31/2016

Bug No known bugs.

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

3.7 TestSuite.h File Reference

Single test class.

Classes

- class **test::testSuite**

Namespaces

- **test**

3.7.1 Detailed Description

Single test class.

Jonathan Bedard

Date

5/14/2016

Bug No known bugs.

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

3.8 TestSuite.cpp File Reference

Single test class.

3.8.1 Detailed Description

Single test class.

Jonathan Bedard

Date

6/3/2016

Bug No known bugs.

This file contains declarations for a test suite. Consult **testSuite.h** (p. 8) for details.

3.9 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.

3.9.1 Detailed Description

Unit Test header file.

Author

Jonathan Bedard

Date

5/14/2016

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.

3.10 UnitTest.cpp File Reference

Unit Test logging and global functions.

3.10.1 Detailed Description

Unit Test logging and global functions.

Author

Jonathan Bedard

Date

6/25/2016

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.

3.11 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.

3.12 UnitTestExceptions.h File Reference

Common exceptions thrown by unit tests.

Classes

- class **test::generalTestException**
Base class for test exceptions.
- class **test::unknownException**
Unknown exception class.
- class **test::nullFunctionException**
NULL function exception class.

Namespaces

- **test**

Functions

- void **test::throwGeneralTestException** (const std::string &description, const std::string &location)
Creates and throws a test exception.
- void **test::throwUnknownException** (const std::string &location)
Creates and throws an unknown exception.
- void **test::throwNullFunctionException** (const std::string &location)
Creates and throws a null function exception.

3.12.1 Detailed Description

Common exceptions thrown by unit tests.

Jonathan Bedard

Date

6/25/2016

Bug No known bugs.

This file contains a number of common test exceptions used by unit tests. All of these classes extend std::exception.

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	17
test::libraryTests	
Library test group	19
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	31
test::testSuite	34
test::unknownException	
Unknown exception class	40

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**
- 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.
- void **throwGeneralTestException** (const std::string &description, const std::string &location)
Creates and throws a test exception.
- void **throwUnknownException** (const std::string &location)
Creates and throws an unknown exception.
- void **throwNullFunctionException** (const std::string &location)
Creates and throws a null function exception.
- std::ostream & **testout_func** ()
Standard out object for test namespace.
- std::ostream & **testerr_func** ()
Standard error object for test namespace.

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. 31).

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.

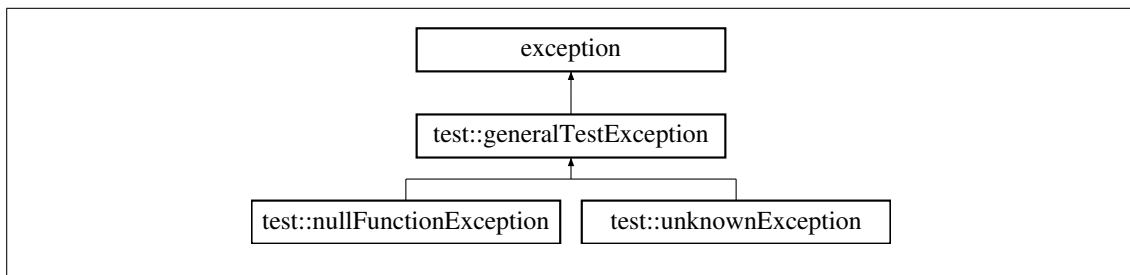
Chapter 6

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.
- virtual const char * **what** () const throw ()
std::exception overload
- const std::string & **getLocation** () const
Location description.
- const std::string & **getString** () const
Error description.

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. 19) 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. 19)

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

Error description.

Returns

`test::generalTestException::_error` (p. 19)

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

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. 19) and **test::generalTestException::location** (p. 19).

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.
- bool **operator==** (const **libraryTests** <) const
Equality comparison.
- bool **operator!=** (const **libraryTests** <) const
Not-equals comparison.
- bool **operator>** (const **libraryTests** <) const
Greater-than comparison.
- bool **operator<** (const **libraryTests** <) const
Less-than comparison.
- bool **operator>=** (const **libraryTests** <) const
Greater-than or equal to comparison.
- bool **operator<=** (const **libraryTests** <) const
Less-than or equal to comparison.

Private Attributes

- std::string **libName**
Name of library to be tested.
- os::smartSet< **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	In	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::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

bool test::libraryTests::operator!= (const **libraryTests** & lt) const [inline]

Not-equals comparison.

Compares two test::libraryTest based on the library name. If the two names are not-equal, the library tests are not-equal.

Parameters

in	lt	Reference to test::libraryTest to be compared against
----	----	---

Returns

this->libName!=lt.libName

bool test::libraryTests::operator< (const **libraryTests** & lt) const [inline]

Less-than comparison.

Compares two test::libraryTest based on the library name. If the name of this object is less than the name of the reference object, return true.

Parameters

in	lt	Reference to test::libraryTest to be compared against
----	----	---

Returns

this->libName<lt.libName

bool test::libraryTests::operator<= (const **libraryTests** & lt) const [inline]

Less-than or equal to comparison.

Compares two `test::libraryTest` based on the library name. If the name of this object is less than or equal to the name of the reference object, return true.

Parameters

<code>in</code>	<code>lt</code>	Reference to <code>test::libraryTest</code> to be compared against
-----------------	-----------------	--

Returns

`this->libName<=lt.libName`

```
bool test::libraryTests::operator==( const libraryTests & lt ) const [inline]
```

Equality comparison.

Compares two `test::libraryTest` based on the library name. If the two names are equal, the library tests are equal.

Parameters

<code>in</code>	<code>lt</code>	Reference to <code>test::libraryTest</code> to be compared against
-----------------	-----------------	--

Returns

`this->libName==lt.libName`

```
bool test::libraryTests::operator> ( const libraryTests & lt ) const [inline]
```

Greater-than comparison.

Compares two `test::libraryTest` based on the library name. If the name of this object is greater than the name of the reference object, return true.

Parameters

<code>in</code>	<code>lt</code>	Reference to <code>test::libraryTest</code> to be compared against
-----------------	-----------------	--

Returns

`this->libName>lt.libName`

```
bool test::libraryTests::operator>= ( const libraryTests & lt ) const [inline]
```

Greater-than or equal to comparison.

Compares two `test::libraryTest` based on the library name. If the name of this object is greater than or equal to the name of the reference object, return true.

Parameters

<code>in</code>	<code>lt</code>	Reference to <code>test::libraryTest</code> to be compared against
-----------------	-----------------	--

Returns

`this->libName>=lt.libName`

`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

<code>in</code>	<code>suite</code>	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

<code>in</code>	<code>suite</code>	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::smartSet<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 os::smart_ptr< **masterTestHolder** > **singleton** ()
Singleton access.

Private Member Functions

- **masterTestHolder** ()
Private constructor.

Private Attributes

- `os::smartSet< 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. 19) 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. 19) 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 os::smart_ptr<**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::smartSet<**libraryTests**> test::masterTestHolder::libraryList [private]

Set of library tests.

int test::masterTestHolder::libsCompleted [private]

Number of libraries successfully completed.

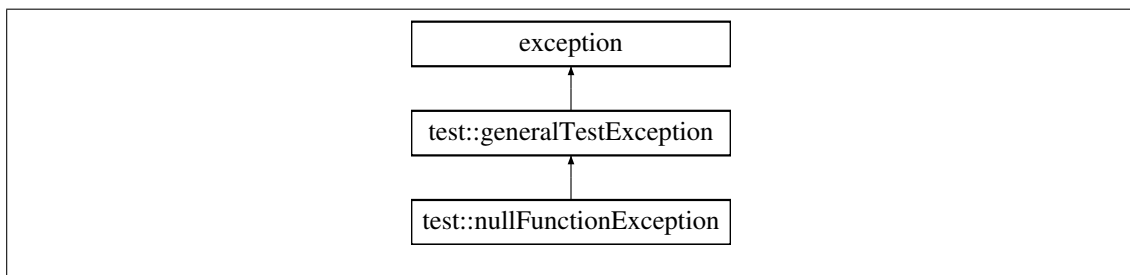
int test::masterTestHolder::libsRun [private]

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.
- virtual **~nullFunctionException** () throw ()
Virtual destructor.

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. 17) with the provided location and the static string for a NULL function exception.

Parameters

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

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

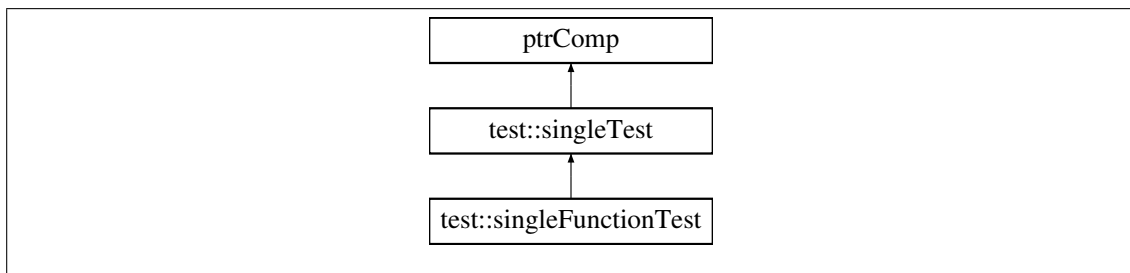
Virtual destructor.

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

6.5 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.

- virtual **~singleFunctionTest** ()

Virtual destructor.

- void **test** () 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. 31) 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

`virtual test::singleFunctionTest::~~singleFunctionTest () [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.

6.5.3 Member Function Documentation

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

Call unit test function.

Calls the function bound to this class in the constructor pointed to by **test::singleFunctionTest::func** (p. 31). 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

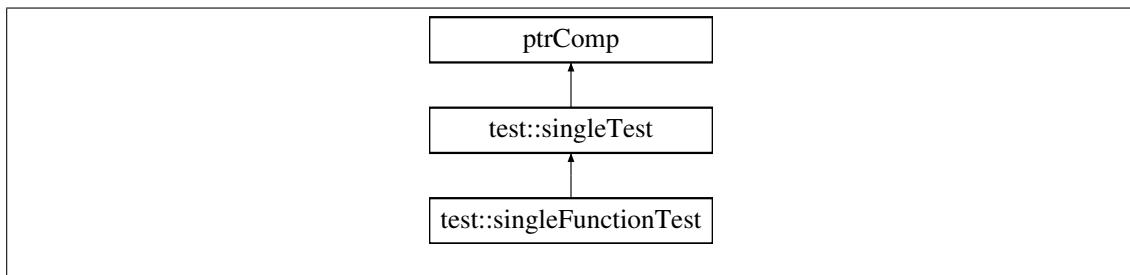
testFunction `test::singleFunctionTest::func [private]`

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	tn	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

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.
- bool **operator==** (const **testSuite** <) const
Equality comparison.
- bool **operator!=** (const **testSuite** <) const
Not-equals comparison.
- bool **operator>** (const **testSuite** <) const
Greater-than comparison.
- bool **operator<** (const **testSuite** <) const
Less-than comparison.
- bool **operator>=** (const **testSuite** <) const
Greater-than or equal to comparison.
- bool **operator<=** (const **testSuite** <) const
Less-than or equal to comparison.

Private Attributes

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

6.7.1 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.

6.7.2 Member Function Documentation

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

Number of tests attempted to run.

Returns

test::testSuite::testsRun (p. 40)

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

Number of tests successfully completed.

Returns

test::testSuite::testsCompleted (p. 40)

`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

bool test::testSuite::operator!= (const **testSuite** & lt) const [inline]

Not-equals comparison.

Compares two **test::testSuite** (p. 34) based on the library name. If the two names are not-equal, the suites are not-equal.

Parameters

in	lt	Reference to test::testSuite (p. 34) to be compared against
----	----	--

Returns

this->suiteName!=lt.suiteName

bool test::testSuite::operator< (const **testSuite** & lt) const [inline]

Less-than comparison.

Compares two **test::testSuite** (p. 34) based on the library name. If the name of this object is less than the name of the reference object, return true.

Parameters

in	lt	Reference to test::testSuite (p. 34) to be compared against
----	----	--

Returns

this->suiteName<lt.suiteName

bool test::testSuite::operator<= (const **testSuite** & lt) const [inline]

Less-than or equal to comparison.

Compares two **test::testSuite** (p. 34) based on the library name. If the name of this object is less than or equal to the name of the reference object, return true.

Parameters

in	lt	Reference to test::testSuite (p. 34) to be compared against
----	----	--

Returns

`this->suiteName<=lt.suiteName`

`bool test::testSuite::operator== (const testSuite & lt) const [inline]`

Equality comparison.

Compares two **test::testSuite** (p. 34) based on the suite name. If the two names are equal, the suites are equal.

Parameters

in	<i>lt</i>	Reference to test::testSuite (p. 34) to be compared against
-----------	-----------	--

Returns

`this->suiteName==lt.suiteName`

`bool test::testSuite::operator> (const testSuite & lt) const [inline]`

Greater-than comparison.

Compares two **test::testSuite** (p. 34) based on the library name. If the name of this object is greater than the name of the reference object, return true.

Parameters

in	<i>lt</i>	Reference to test::testSuite (p. 34) to be compared against
-----------	-----------	--

Returns

`this->suiteName>lt.suiteName`

`bool test::testSuite::operator>= (const testSuite & lt) const [inline]`

Greater-than or equal to comparison.

Compares two **test::testSuite** (p. 34) based on the library name. If the name of this object is greater than or equal to the name of the reference object, return true.

Parameters

in	<i>lt</i>	Reference to test::testSuite (p. 34) to be compared against
-----------	-----------	--

Returns

`this->suiteName>=lt.suiteName`

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

Add test to the set.

Adds a **test::singleTest** (p. 31) 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. 14) to the set of tests to be tested. Constructs a **test::singleTest** (p. 31) 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. 31) 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. 31) and logs the results.

Returns

void

6.7.3 Member Data Documentation

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

Name of test suite.

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

Set of tests.

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

Number of tests successfully completed.

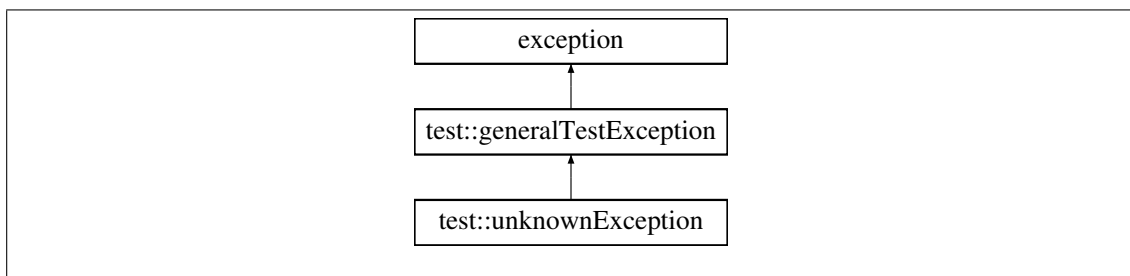
`int test::testSuite::testsRun [private]`

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.
- virtual ~**unknownException** () throw ()
Virtual destructor.

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. 17) with the provided location and the static string for an unknown exception.

Parameters

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

`virtual test::unknownException::~~unknownException () 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.

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	47
adsFrame.h	Include headers with macro management	??
arrayStack.h	Defines a stack with an array	??
asyncAVL.h	Asynchronous AVL tree	48
AVL.h	AVL tree	49
basicStructures.h	Include structure headers with macro management	??
Datastructures.h	Master Datastructures header file	46
descriptiveException.h	Basic descriptive exception	??
eventDriver.h	Event sender and receiver	49
iterator.h	Defines an iterator	??
linkedQueue.h	Defines a queue with a linked list	??
linkedStack.h	Defines a stack with a linked list	??
list.h	Doubly Linked List	50
lock.cpp	Implementation of constant locks	??

lock.h	Constant locking support	??
macroAVL.h	Defines a generalized AVL tree	??
macroBoundedQueue.h	Defines a bounded queue	??
macroDatastructuresInterface.h	Defines the interface used by datastructures	??
macroHash.h	Defines an iterable hash table	??
macroList.h	Defines a basic linked list	??
macros.h	Support macro declarations	??
macroVector.h	Defines an expandable array	??
matrix.h	Matrix templates	51
nodeFrame.h	Declares a framework for nodes	??
nodeMacroHeader.h	Node macro header	??
osLogger.cpp	Logging for os namespace, implementation	65
osLogger.h	Logging for os namespace	64
osVectors.h	Vector templates	65
set.h	Smart Set	67
simpleHash.h	Constant locking support	??
smartPointer.h	Template declaration of os::smart_ptr (p. 151)	68
specializedStructures.h	Include specialized structure headers with macro management	??
staticConstantPrinter.cpp	Constant printing support, implementation	76
staticConstantPrinter.h	Constant printing support	75

Chapter 9

File Documentation

9.1 Datastructures.h File Reference

Master Datastructures header file.

9.1.1 Detailed Description

Master Datastructures header file.

Author

Jonathan Bedard

Date

6/15/2016

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.

9.2 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 int length, int(*sort_comparison)(const dataType`
`&, const dataType &)=&defaultCompare)`
Template quick-sort.
- `template<class dataType >`
`void os::pointerQuicksort (smart_ptr< smart_ptr< dataType > > arr, unsigned int length,`
`int(*sort_comparison)(const smart_ptr< dataType > &, const smart_ptr< dataType > &)=&pointer`
`Compare)`
Template for quick-sort, pointer version.

9.2.1 Detailed Description

Template for sorting arrays.

Author

Jonathan Bedard

Date

6/4/2016

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.

9.3 ads.h File Reference

Abstract datastructure interface.

Classes

- class **os::ptrComp**
Pointer compare interface.
- class **os::adnode< dataType >**
Abstract data-node.
- class **os::ads< dataType >**
Abstract datastructure.

Namespaces

- **os**

9.3.1 Detailed Description

Abstract datastructure interface.

Author

Jonathan Bedard

Date

5/9/2016

Bug No known bugs.

This file contains definitions of a set of class interfaces used by abstract datastructures and classes interfacing with abstract datastructures.

9.4 asyncAVL.h File Reference

Asynchronous AVL tree.

Classes

- class **os::asyncAVLNode< dataType >**
Node for usage in an asynchronous AVL tree.
- class **os::asyncAVLTree< dataType >**
Asynchronous balanced binary search tree.

Namespaces

- **os**

9.4.1 Detailed Description

Asynchronous AVL tree.

Author

Jonathan Bedard

Date

5/9/2016

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.

9.5 AVL.h File Reference

AVL tree.

Classes

- class **os::AVLNode**< **dataType** >
Node for usage in an AVL tree.
- class **os::AVLTree**< **dataType** >
Balanced binary search tree.

Namespaces

- **os**

9.5.1 Detailed Description

AVL tree.

Author

Jonathan Bedard

Date

2/12/2016

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 not thread safe.

9.6 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**

9.6.1 Detailed Description

Event sender and receiver.

Author

Jonathan Bedard

Date

6/28/2016

Bug No known bugs.

Both **os::eventReceiver** (p. 132) and **os::eventSender** (p. 135) are experimental classes and have not been tested or utilized.

9.7 eventDriver.cpp File Reference

Event driver implementation.

9.7.1 Detailed Description

Event driver implementation.

Author

Jonathan Bedard

Date

2/28/2016

Bug No known bugs.

This file implements **os::eventLock** (p. ??) for **os::eventSender** (p. 135) and **os::eventReceiver** (p. 132). These are experimental class and not yet used or tested

9.8 list.h File Reference

Doubly Linked List.

Classes

- class **os::unsortedListNode**< **dataType** >
Node for usage in a linked list.
- class **os::unsortedList**< **dataType** >
Unsorted linked list.

Namespaces

- **os**

9.8.1 Detailed Description

Doubly Linked List.

Author

Jonathan Bedard

Date

2/1/2016

Bug No known bugs.

This file contains a template class definition of a linked list and its nodes. This list has insertion, find and delete of $O(n)$. The linked list provided is doubly linked, allowing for forward and backward traversal. This list is not thread safe.

9.9 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.

9.9.1 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.

9.9.2 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

9.10 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.

9.10.1 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.

9.11 osLogger.cpp File Reference

Logging for os namespace, implementation.

9.11.1 Detailed Description

Logging for os namespace, implementation.
Jonathan Bedard

Date

2/15/2016

Bug No known bugs.

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

9.12 osVectors.h File Reference

Vector templates.

Classes

- class `os::vector2d< dataType >`
2-dimensional vector
- class `os::vector3d< dataType >`
3-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
- 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

9.12.1 Detailed Description

Vector templates.

Author

Jonathan Bedard

Date

5/16/2016

Bug No known bugs.

This file contains two template classes defining vector objects. 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.

9.13 set.h File Reference

Smart Set.

Classes

- class **os::smartSet**< **dataType** >
Smart set abstract data-structures.

Namespaces

- **os**

Enumerations

- enum **os::setTypes** { **os::def_set** =0, **os::small_set**, **os::sorted_set** }
Index of abstract data-structures.

9.13.1 Detailed Description

Smart Set.

Author

Jonathan Bedard

Date

2/12/2016

Bug No known bugs.

This file contains a template class defining a "smart set." A smart set wraps other forms of abstract data structures, allowing applications to define abstract data-structures by numbered indexes.

9.14 smartPointer.h File Reference

Template declaration of **os::smart_ptr** (p. 151).

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. 151).*

Functions

- template<class targ , class src >
 smart_ptr< targ > **os::cast** (const **os::smart_ptr**< src > &conv) throw ()
 os::smart_ptr (p. 151) 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 ()

- [illegible]

- [illegible]

- `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 ()`
- `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 ()`

9.14.1 Detailed Description

Template declaration of `os::smart_ptr` (p. 151).

Author

Jonathan Bedard

Date

6/30/2016

Bug No known bugs.

This file contains a template declaration of `os::smart_ptr` (p. 151) and supporting constants and functions. Note that because `os::smart_ptr` (p. 151) is a template class, the implementation of `os::smart_ptr` (p. 151) occurs here as well.

9.14.2 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]

```

9.15 staticConstantPrinter.h File Reference

Constant printing support.

Classes

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

Namespaces

- **os**

9.15.1 Detailed Description

Constant printing support.

Author

Jonathan Bedard

Date

5/31/2016

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.

9.16 staticConstantPrinter.cpp File Reference

Constant printing support, implementation.

9.16.1 Detailed Description

Constant printing support, implementation.

Author

Jonathan Bedard

Date

5/31/2016

Bug No known bugs.

This file implements **os::constantPrinter** (p. 128). Consult **staticConstantPrinter.h** (p. 75) for detailed documentation.

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	94
os::ads< dataType >	
Abstract datastructure	97
os::ARRAY_STACK< dataType >	??
os::asyncAVLNode< dataType >	
Node for usage in an asynchronous AVL tree	101
os::asyncAVLTree< dataType >	
Asynchronous balanced binary search tree	107
os::AVL_NODE< dataType >	
AVL tree node Node used in an AVL tree. This node allows both iteration and random access	??
os::AVL_TREE< dataType >	
Template AVL Tree definition	??
os::AVLNode< dataType >	
Node for usage in an AVL tree	115
os::AVLTree< dataType >	
Balanced binary search tree	121
os::basicLock	
Basic lock Wraps the std::mutex class allowing it to be accessed in a constant way	??
os::BOUNDED_QUEUE< dataType >	
Template BOUNDED_QUEUE (p. ??) definition	??
os::BOUNDED_QUEUE_NODE< dataType >	
BOUNDED_QUEUE (p. ??) node	??
os::constantPrinter	
Prints constant arrays to files	128
os::constIterator< dataType >	
Generalized constant iterator	??
os::DATASTRUCTURE< dataType >	
Object node definition	??

os::descriptiveException	
Basic exception with description	??
os::errorPointer	
Error pointer class	??
os::eventReceiver< senderType >	
Class which enables event receiving	132
os::eventSender< receiverType >	
Class which enables event sending	135
os::HASH< dataType >	
Iterable hash table	??
os::HASH_NODE< dataType >	
Hash node	??
os::indirectMatrix< dataType >	
Indirect matrix	139
os::iterator< dataType >	
Generalized iterator	??
os::iteratorBase< dataType >	
Iterator source	??
os::iteratorSource	
Base class for iterator source	??
os::LINKED_QUEUE< dataType >	
Queue built on-top of a list	??
os::LINKED_STACK< dataType >	
Stack built on-top of a list	??
os::LIST< dataType >	
List framework Template for a linked list, insertion is fully defined in subsequent extensions of this class	??
os::LIST_NODE< dataType >	
List node Node used in linked lists to store each element	??
os::lockable	
An interface defining a class which can be locked	??
os::matrix< dataType >	
Raw matrix	144
os::NODE< dataType >	
Object node definition	??
os::nodeFrame< dataType >	
NodeFrame interface	??
os::ptrComp	
Pointer compare interface	150
os::readWriteInterface	
Read/write interface	??
os::readWriteLock	
Read/write lock	??
os::simpleHash< dataType >	
Basic hash function	??
os::smart_ptr< dataType >	
Reference counted pointer	151
os::smartSet< dataType >	
Smart set abstract data-structures	163

os::SORTED_LIST< dataType >	
Sorted list A basic list which remains unsorted unless the sort function is called on it	??
os::threadCounter	
Thread counter	??
os::threadLock	
Basic 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	??
os::UNSORTED_LIST< dataType >	
Unsorted list A basic list which remains unsorted unless the sort function is called on it	??
os::unsortedList< dataType >	
Unsorted linked list	167
os::unsortedListNode< dataType >	
Node for usage in a linked list	171
os::VECTOR< dataType >	
Template vector definition	??
os::vector2d< dataType >	
2-dimensional vector	174
os::vector3d< dataType >	
3-dimensional vector	184
os::VECTOR_NODE< dataType >	
Vector node	??

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. ??) definition.*
- class **BOUNDED_QUEUE_NODE**
***BOUNDED_QUEUE** (p. ??) 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 **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. ??) 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**
 - Basic 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 **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
- typedef void(* **void_rec**) (void *)
Deletion function typedef.

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. 151).*

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 int length, int(*sort_comparison)(const dataType &, const dataType &)=&defaultCompare)`
Template quick-sort.
- `template<class dataType >`
`void pointerQuicksort (smart_ptr< smart_ptr< dataType > > arr, unsigned int length, int(*sort_←
_comparison)(const smart_ptr< dataType > &, const smart_ptr< dataType > &)=&pointer←
Compare)`
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. 151) 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. 87) function pointer typedef is used by **os::smart_ptr** (p. 151) when it is of type **os::shared_type_dynamic_delete** (p. 88) 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. 151).

Defines types of **os::smart_ptr** (p. 151). These types are used to define the deletion behaviour of the pointer.

Enumerator

null_type No type. **os::null_type** (p. 87) pointers are the default type of **os::smart_ptr** (p. 151). Any **os::smart_ptr** (p. 151) of type **os::null_type** (p. 87) can be guaranteed to hold a Null pointer.

raw_type Raw pointer. **os::raw_type** (p. 87) pointers are the default type of **os::smart_ptr** (p. 151) when instantiated with a standard pointer. Any **os::smart_ptr** (p. 151) of type **os::raw_type** (p. 87) is not responsible for the deletion of its pointer and makes no guarantees as to the availability of its pointer.

shared_type Reference counted pointer. **os::shared_type** (p. 87) pointers must be instantiated from an **os::smart_ptr** (p. 151) of this type or explicitly through **os::smart_ptr** (p. 151) constructor arguments. **os::shared_type** (p. 87) pointers will automatically delete the pointer contained within the object when the reference count of the **os::smart_ptr** (p. 151) reaches 0.

shared_type_array Reference counted array. Similar in usage and instantiation to **os::raw_type** (p. 87). **os::smart_ptr** (p. 151) of type **os::shared_type_array** (p. 88) 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. 87). **os::smart_ptr** (p. 151) of type **os::shared_type_dynamic_delete** (p. 88) 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. 151) 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. 151)

```
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

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

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

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

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

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

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

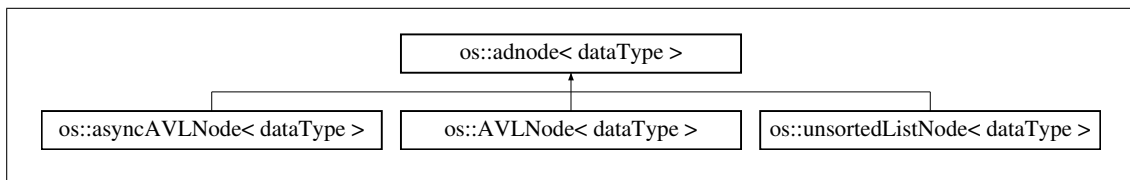
Chapter 12

Class Documentation

12.1 os::adnode< dataType > Class Template Reference

Abstract data-node.

Inheritance diagram for os::adnode< dataType >:



Public Member Functions

- **adnode** (**smart_ptr**< dataType > d)
Abstract data-node constructor.
- virtual **~adnode** ()
Virtual destructor.
- int **compare** (**smart_ptr**< **adnode**< dataType > > inp, bool rawComp=false)
Compares two abstract data-nodes.
- **smart_ptr**< dataType > & **getData** ()
Return a reference to the data pointer.
- **smart_ptr**< dataType > & **operator*** ()
Return a reference to the data pointer.
- virtual **smart_ptr**< **adnode**< dataType > > **getNext** ()
Find the next node.
- virtual **smart_ptr**< **adnode**< dataType > > **getPrev** ()
Find the previous node.

Protected Attributes

- **smart_ptr**< dataType > **data**

Data pointer.

12.1.1 Detailed Description

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

Abstract data-node.

A generalized node class used for linked lists, trees, queues and various other abstract data structures. Primarily, this structure is focused on providing access to the node data and allowing traversal of the data-structure.

12.1.2 Constructor & Destructor Documentation

```
template<class dataType > os::adnode< dataType >::adnode ( smart_ptr< dataType > d )
[inline]
```

Abstract data-node constructor.

An abstract data-node is meaningless without a pointer to it's dataType. The constructor requires this pointer to initialize the node.

Parameters

in	<i>d</i>	Data to be bound to the node
----	----------	------------------------------

```
template<class dataType > virtual os::adnode< dataType >::~adnode ( ) [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.

12.1.3 Member Function Documentation

```
template<class dataType > int os::adnode< dataType >::compare ( smart_ptr< adnode<
dataType > > inp, bool rawComp = false ) [inline]
```

Compares two abstract data-nodes.

Abstract data nodes use the comparison functions defined by their data pointers to determine their comparison

Parameters

in	<i>inp</i>	Data-node being compared with
----	------------	-------------------------------

Returns

1, 0, -1 (Greater than, equal to, less than)

```
template<class dataType > smart_ptr<dataType>& os::adnode< dataType >::getData ( )  
[inline]
```

Return a reference to the data pointer.

Returns

adnode<**dataType**>::data (p. 96)

```
template<class dataType > virtual smart_ptr<adnode<dataType> > os::adnode< dataType  
>::getNext ( ) [inline], [virtual]
```

Find the next node.

This functions attempts to search for the next node in the structure. By default, or if this node either cannot be found or does not exist, a NULL pointer is returned.

Returns

Pointer to the next node in the structure

Reimplemented in **os::asyncAVLNode**< **dataType** > (p. 103), **os::AVLNode**< **dataType** > (p. 117), and **os::unsortedListNode**< **dataType** > (p. 172).

```
template<class dataType > virtual smart_ptr<adnode<dataType> > os::adnode< dataType  
>::getPrev ( ) [inline], [virtual]
```

Find the previous node.

This functions attempts to search for the previous node in the structure. By default, or if this node either cannot be found or does not exist, a NULL pointer is returned.

Returns

Pointer to the previous node in the structure

Reimplemented in **os::AVLNode**< **dataType** > (p. 118), **os::asyncAVLNode**< **dataType** > (p. 104), and **os::unsortedListNode**< **dataType** > (p. 173).

```
template<class dataType > smart_ptr<dataType>& os::adnode< dataType >::operator* ( )  
[inline]
```

Return a reference to the data pointer.

Returns

adnode<**dataType**>::data (p. 96)

12.1.4 Member Data Documentation

```
template<class dataType > smart_ptr<dataType> os::adnode< dataType >::data [protected]
```

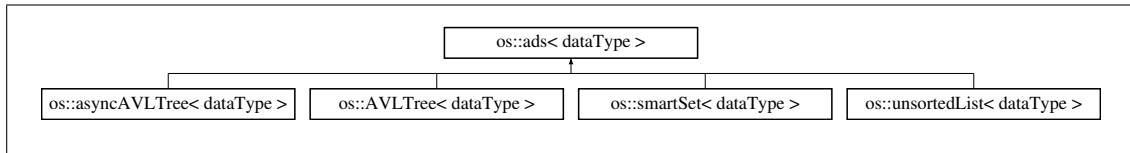
Data pointer.

A pointer to the data being held by the node. This is used to compare nodes as well.

12.2 os::ads< dataType > Class Template Reference

Abstract datastructure.

Inheritance diagram for os::ads< dataType >:



Public Member Functions

- **ads ()**
Default constructor.
- **virtual ~ads ()**
Virtual destructor.
- **virtual bool insert (smart_ptr< dataType > x)**
Inserts a data pointer.
- **virtual unsigned int size () const**
Returns the number of elements in the datastructure.
- **virtual smart_ptr< adnode< dataType > > find (smart_ptr< dataType > x)**
Finds a matching node.
- **virtual bool findDelete (smart_ptr< dataType > x)**
Finds a matching node and removes it.
- **virtual smart_ptr< adnode< dataType > > getFirst ()**
Returns the first node.
- **virtual smart_ptr< adnode< dataType > > getLast ()**
Returns the last node.
- **virtual bool insert (smart_ptr< ads< dataType > > x)**
Inserts an entire datastructure.
- **bool rawInsert (smart_ptr< dataType > x)**
Inserts a data pointer.
- **bool rawCompare () const**
Return state of raw compare.
- **void setRawCompare (bool rwcmp)**
Set raw-compare.

Protected Attributes

- **bool _rawCompare**
Allows for raw compare data-structures.

12.2.1 Detailed Description

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

Abstract datastructure.

A generalized datastructure class which acts as an interface for all datastructures classes. If not extended, the abstract datastructures class is useless.

12.2.2 Constructor & Destructor Documentation

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

Default constructor.

This constructor does nothing, as there are no objects to initialize.

```
template<class dataType > virtual os::ads< dataType >::~ads ( ) [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.

12.2.3 Member Function Documentation

```
template<class dataType > virtual smart_ptr<adnode<dataType> > os::ads< dataType >::find (
smart_ptr< dataType > x ) [inline], [virtual]
```

Finds a matching node.

Finds a pointer to an object of type "dataType" given a comparison pointer. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType> >)`. Each datastructure which inherits from this class will re-implement this function.

[in] x dataType pointer to be compared against

Returns

The found node if applicable, else NULL

Reimplemented in `os::AVLTree< dataType >` (p. 124), `os::asyncAVLTree< dataType >` (p. 111), `os::unsortedList< dataType >` (p. 169), and `os::smartSet< dataType >` (p. 165).

```
template<class dataType > virtual bool os::ads< dataType >::findDelete ( smart_ptr< dataType >
x ) [inline], [virtual]
```

Finds a matching node and removes it.

Finds a pointer to an object of type "dataType" given a comparison pointer. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType> >)`. Each datastructure which inherits from this class will re-implement this function. After finding a node, it will be removed from the datastructure.

[in] x dataType pointer to be compared against

Returns

true if the node was found and deleted, else false

Reimplemented in **os::AVLTree< dataType >** (p. 125), **os::asyncAVLTree< dataType >** (p. 112), **os::unsortedList< dataType >** (p. 169), and **os::smartSet< dataType >** (p. 165).

```
template<class dataType > virtual smart_ptr<adnode<dataType> > os::ads< dataType >::getFirst  
( ) [inline], [virtual]
```

Returns the first node.

Each datastructure has a different definition of what defines "first." By default, this function returns NULL. Datastructures which inherit from this class must re-implement this function.

Returns

The first node, if it exists

Reimplemented in **os::asyncAVLTree< dataType >** (p. 113), **os::AVLTree< dataType >** (p. 126), **os::unsortedList< dataType >** (p. 169), and **os::smartSet< dataType >** (p. 165).

```
template<class dataType > virtual smart_ptr<adnode<dataType> > os::ads< dataType >::getLast  
( ) [inline], [virtual]
```

Returns the last node.

Each datastructure has a different definition of what defines "last." By default, this function returns NULL. Datastructures which inherit from this class must re-implement this function.

Returns

The last node, if it exists

Reimplemented in **os::asyncAVLTree< dataType >** (p. 113), **os::AVLTree< dataType >** (p. 126), **os::unsortedList< dataType >** (p. 170), and **os::smartSet< dataType >** (p. 166).

```
template<class dataType > virtual bool os::ads< dataType >::insert ( smart_ptr< dataType > x )  
[inline], [virtual]
```

Inserts a data pointer.

Inserts a pointer to an object of type "dataType." Each datastructure which inherits from this class will re-implement this function

[in] x dataType pointer to be inserted

Returns

true if successful, false if failed

Reimplemented in **os::AVLTree< dataType >** (p. 127), **os::asyncAVLTree< dataType >** (p. 114), **os::unsortedList< dataType >** (p. 170), and **os::smartSet< dataType >** (p. 166).


```
template<class dataType > virtual bool os::ads< dataType >::insert ( smart_ptr< ads< dataType  
> > x ) [inline], [virtual]
```

Inserts an entire datastructure.

This function may be redefined to speed-up insertion. Currently, this function will be $O(n * \text{insertionTime})$ where n is the number of elements in x
[in] x datastructure of type dataType to be inserted

Returns

true if successful, false if failed

Reimplemented in **os::AVLTree**< dataType > (p. 127), **os::asyncAVLTree**< dataType > (p. 113),
os::unsortedList< dataType > (p. 170), and **os::smartSet**< dataType > (p. 166).

```
template<class dataType > bool os::ads< dataType >::rawCompare ( ) const [inline]
```

Return state of raw compare.

Returns

_rawCompare

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

Inserts a data pointer.

Inserts a pointer to an object of type "dataType." This function disabiguates certain calls to insert.
[in] x dataType pointer to be inserted

Returns

true if successful, false if failed

```
template<class dataType > void os::ads< dataType >::setRawCompare ( bool rwcmp )  
[inline]
```

Set raw-compare.

Parameters

in	<i>rwcmp</i>	Value of raw compare to set
----	--------------	-----------------------------

Returns

void

```
template<class dataType > virtual unsigned int os::ads< dataType >::size ( ) const [inline],  
[virtual]
```

Returns the number of elements in the datastructure.

This function must be re-implemented by all classes which inherit from this class. By default, this function returns 0.

Returns

number of elements as an unsigned integer

Reimplemented in **os::asyncAVLTree< dataType >** (p. 114), **os::AVLTree< dataType >** (p. 127), **os::unsortedList< dataType >** (p. 170), and **os::smartSet< dataType >** (p. 167).

12.2.4 Member Data Documentation

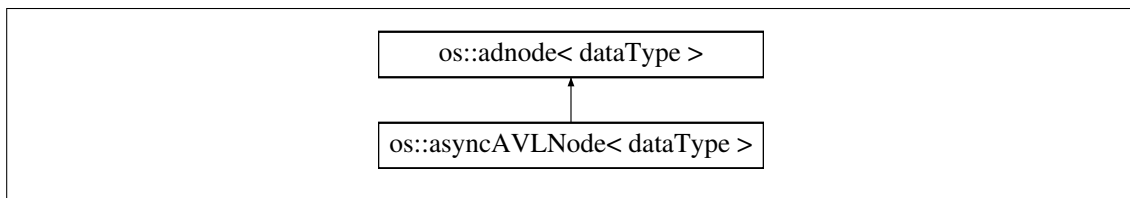
template<class dataType > bool **os::ads< dataType >::_rawCompare** [protected]

Allows for raw compare data-structures.

12.3 os::asyncAVLNode< dataType > Class Template Reference

Node for usage in an asynchronous AVL tree.

Inheritance diagram for os::asyncAVLNode< dataType >:



Public Member Functions

- **asyncAVLNode (smart_ptr< dataType > d, asyncAVLTree< dataType > *master)**
Abstract data-node constructor.
- virtual **~asyncAVLNode ()**
Virtual destructor.
- **smart_ptr< adnode< dataType > > getNext ()**
Find the next node.
- **smart_ptr< adnode< dataType > > getPrev ()**
Find the previous node.

Protected Member Functions

- **smart_ptr< asyncAVLNode< dataType > > getParent ()**
Returns the parent node.
- **smart_ptr< asyncAVLNode< dataType > > getChild (int x)**
Returns a child by index.
- int **getHeight () const**
Returns the height of the sub-tree.
- void **setHeight ()**
Sets the height of the sub-tree.

- void **setChild** (**smart_ptr**< **asyncAVLNode**< dataType > > c, bool _rawCompare)
Add a child to this node.
- void **setParent** (**smart_ptr**< **asyncAVLNode**< dataType > > p, **smart_ptr**< **asyncAVLNode**< dataType > > self_pointer, bool _rawCompare)
Sets the parent node.
- void **removeChild** (**smart_ptr**< **asyncAVLNode**< dataType > > c, bool _rawCompare)
Remove a child from this node.
- void **removeChild** (int pos)
Remove a child from this node.
- void **removeParent** ()
Remove the parent node.
- void **remove** ()
Remove all children and parents.

Protected Attributes

- **smart_ptr**< **asyncAVLNode**< dataType > > **parent**
Parent node one level up in the tree.
- **smart_ptr**< **asyncAVLNode**< dataType > > **child1**
Left child one level down in the tree.
- **smart_ptr**< **asyncAVLNode**< dataType > > **child2**
Right child one level down in the tree.
- int **height**
The height of the tree.
- **asyncAVLTree**< dataType > * **masterTree**
Reference to source tree.

Friends

- class **asyncAVLTree**< dataType >
AVL Tree must know details of node implementation.

12.3.1 Detailed Description

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

Node for usage in an asynchronous AVL tree.

The AVL node class implements a number of functions unique to an AVL tree. This node has knowledge of the structure of the AVL tree through its parent and children.

12.3.2 Constructor & Destructor Documentation

```
template<class dataType > os::asyncAVLNode< dataType >::asyncAVLNode ( smart_ptr<
dataType > d, asyncAVLTree< dataType > * master ) [inline]
```

Abstract data-node constructor.

An AVL node is meaningless without a pointer to it's dataType. The constructor requires this pointer to initialize the node. Parent and children nodes are, by default, initialized to 0.

Parameters

in	<i>d</i>	Data to be bound to the node
----	----------	------------------------------

```
template<class dataType > virtual os::asyncAVLNode< dataType >::~~asyncAVLNode ( )
[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.

12.3.3 Member Function Documentation

```
template<class dataType > smart_ptr<asyncAVLNode<dataType> > os::asyncAVLNode<
dataType >::getChild ( int x ) [inline], [protected]
```

Returns a child by index.

Returns child node by index. 0 indicates the left child, **asyncAVLNode**<dataType>::**child1** (p. 106). 1 indicates the right child, **asyncAVLNode**<dataType>::**child2** (p. 106). All other indices will return NULL.

Returns

os::asyncAVLNode<dataType>::**child1** (p. 106) for x==0, **asyncAVLNode**<dataType>::**child2** (p. 106) for x==1

```
template<class dataType > int os::asyncAVLNode< dataType >::getHeight ( ) const [inline],
[protected]
```

Returns the height of the sub-tree.

Returns

os::asyncAVLNode<dataType>::**height** (p. 106)

```
template<class dataType > smart_ptr<adnode<dataType> > os::asyncAVLNode< dataType
>::getNext ( ) [virtual]
```

Find the next node.

This functions attempts to search for the next node in the structure. This trips the traverse flag of the current node and traverses the tree looking for the next node.

Returns

Pointer to the next node in the structure

Reimplemented from **os::adnode< dataType >** (p. 96).

```
template<class dataType > smart_ptr<asyncAVLNode<dataType> > os::asyncAVLNode<  
dataType >::getParent ( ) [inline], [protected]
```

Returns the parent node.

Returns

os::asyncAVLNode<dataType>::parent (p. 107)

```
template<class dataType > smart_ptr<adnode<dataType> > os::asyncAVLNode< dataType  
>::getPrev ( ) [virtual]
```

Find the previous node.

This functions attempts to search for the previous node in the structure. This trips the traverse flag of the current node and traverses the tree looking for the previous node.

Returns

Pointer to the previous node in the structure

Reimplemented from **os::adnode< dataType >** (p. 96).

```
template<class dataType > void os::asyncAVLNode< dataType >::remove ( ) [inline],  
[protected]
```

Remove all children and parents.

This function is important because nodes are of type **os::smart_ptr** (p. 151), 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 > void os::asyncAVLNode< dataType >::removeChild ( smart_ptr<  
asyncAVLNode< dataType > > c, bool_rawCompare ) [inline], [protected]
```

Remove a child from this node.

Checks **os::asyncAVLNode<dataType>::child1** (p. 106) and **os::asyncAVLNode<dataType>::child2** (p. 106) for equality with the the node received as a parameter.

Parameters

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

Returns

void

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

Remove a child from this node.

Remove **os::asyncAVLNode**<dataType>::child1 (p. 106) if position is 0 and **os::asyncAVLNode**<dataType>::child2 (p. 106) if position is 1.

Parameters

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

Returns

void

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

Remove the parent node.

Returns

void

```
template<class dataType > void os::asyncAVLNode< dataType >::setChild ( smart_ptr<  
asyncAVLNode< dataType > > c, bool _rawCompare ) [inline], [protected]
```

Add a child to this node.

Set **os::asyncAVLNode**<dataType>::child1 (p. 106) or **os::asyncAVLNode**<dataType>::child2 (p. 106) 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::asyncAVLNode< dataType >::setHeight ( ) [inline],  
[protected]
```

Sets the height of the sub-tree.

Uses the height of the sub-tree of the node's children to calculate the height of the sub-tree of this node.

Returns

void

```
template<class dataType > void os::asyncAVLNode< dataType >::setParent ( smart_ptr<  
asyncAVLNode< dataType > > p, smart_ptr< asyncAVLNode< dataType > > self_pointer, bool  
_rawCompare ) [inline], [protected]
```

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

12.3.4 Friends And Related Function Documentation

```
template<class dataType > friend class asyncAVLTree< dataType > [friend]
```

AVL Tree must know details of node implementation.

Since the AVL node implements many of the unique functions of the AVL tree, the tree must be aware of the private members of it's nodes.

12.3.5 Member Data Documentation

```
template<class dataType > smart_ptr<asyncAVLNode<dataType> > os::asyncAVLNode<  
dataType >::child1 [protected]
```

Left child one level down in the tree.

```
template<class dataType > smart_ptr<asyncAVLNode<dataType> > os::asyncAVLNode<  
dataType >::child2 [protected]
```

Right child one level down in the tree.

```
template<class dataType > int os::asyncAVLNode< dataType >::height [protected]
```

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 **asyncAVLNode**<**dataType**>::setHeight() (p. 105) resets the height based on the height of the node's children.

```
template<class dataType > asyncAVLTree<dataType>* os::asyncAVLNode< dataType  
>::masterTree [protected]
```

Reference to source tree.

This reference to the source tree is used when incrementing or decrementing the node, locking the tree temporarily.

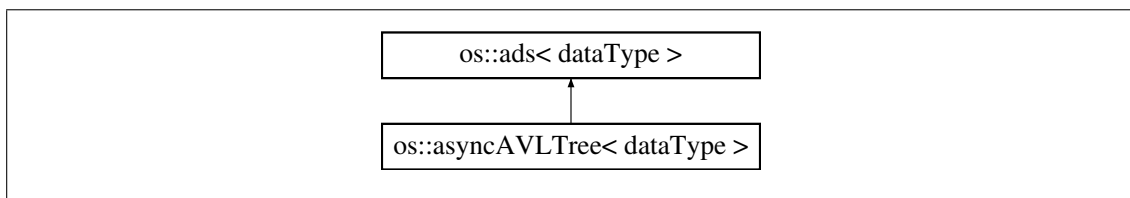
```
template<class dataType > smart_ptr<asyncAVLNode<dataType> > os::asyncAVLNode<  
dataType >::parent [protected]
```

Parent node one level up in the tree.

12.4 os::asyncAVLTree< dataType > Class Template Reference

Asynchronous balanced binary search tree.

Inheritance diagram for os::asyncAVLTree< dataType >:



Public Member Functions

- **asyncAVLTree** ()
Default constructor.
- virtual **~asyncAVLTree** ()
Virtual destructor.
- bool **insert** (**smart_ptr**< **ads**< dataType > > x)
Inserts an os::ads<dataType>
- bool **insert** (**smart_ptr**< dataType > x)
Inserts a data node.
- **smart_ptr**< **asyncAVLNode**< dataType > > **getRoot** ()
Return the root of the tree.
- **smart_ptr**< **adnode**< dataType > > **find** (**smart_ptr**< dataType > x)
Finds a matching node.
- **smart_ptr**< **adnode**< dataType > > **find** (**smart_ptr**< **adnode**< dataType > > x)
Finds by adnode node.
- **smart_ptr**< **asyncAVLNode**< dataType > > **find** (**smart_ptr**< **asyncAVLNode**< dataType > > x)
Finds by asyncAVLNode (p. 101) node.
- bool **findDelete** (**smart_ptr**< dataType > x)
Finds and delete a matching node.

- **bool findDelete** (long x)
Finds and delete a matching node.
- **bool findDelete** (**smart_ptr**< **asyncAVLNode**< dataType > > x)
Finds and delete by node.
- virtual unsigned int **size** () const
Finds and delete a matching node.
- **smart_ptr**< **adnode**< dataType > > **getFirst** ()
Returns the first node.
- **smart_ptr**< **adnode**< dataType > > **getLast** ()
Returns the last node.

Protected Member Functions

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

Protected Attributes

- **smart_ptr**< **asyncAVLNode**< dataType > > **root**
Root node of the tree.
- unsigned int **numElements**
Number of elements in the tree.
- std::mutex **mtx**
Mutex to ensure synchronous access.

Friends

- class **asyncAVLNode**< dataType >
AVL Node must have access to mutex.

12.4.1 Detailed Description

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

Asynchronous balanced binary search tree.

The AVL Tree rigorously balances a binary search tree. As a template class, it can hold any kind of dataType so long as the data type implements basic comparison functions.

12.4.2 Constructor & Destructor Documentation

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

Default constructor.

Sets the number of elements to 0 and the root to NULL.

```
template<class dataType> virtual os::asyncAVLTree< dataType >::~asyncAVLTree ( )
[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. The AVL tree must explicitly force deletion through the **asyncAVLNode<dataType>::remove()** (p. 104) function.

12.4.3 Member Function Documentation

```
template<class dataType> bool os::asyncAVLTree< dataType >::balance ( smart_ptr<
asyncAVLNode< dataType > > x ) [inline], [protected]
```

Balances a single node.

Parameters

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

Returns

true if the node is already balanced, else, false

```
template<class dataType> bool os::asyncAVLTree< dataType >::balanceDelete ( smart_ptr<
asyncAVLNode< dataType > > x, bool _rawCompare ) [inline], [protected]
```

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::asyncAVLTree< dataType >::balanceUp ( smart_ptr<  
asyncAVLNode< dataType > > x ) [inline], [protected]
```

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::asyncAVLTree< dataType >::checkBalance ( smart_ptr<  
asyncAVLNode< dataType > > x ) [inline], [protected]
```

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::asyncAVLTree< dataType >::doubleRotation ( smart_ptr<  
asyncAVLNode< dataType > > r, int dir ) [inline], [protected]
```

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> smart_ptr<adnode<dataType> > os::asyncAVLTree< dataType  
>::find ( smart_ptr< dataType > x ) [inline], [virtual]
```

Finds a matching node.

Finds a pointer to an object of type "dataType" given a comparison pointer. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType> >)`. This function takes $O(\log(n))$ where n is the number of elements in the tree.

[in] x dataType pointer to be compared against

Returns

true if the node was found, else false

Reimplemented from `os::ads< dataType >` (p. 98).

```
template<class dataType> smart_ptr<adnode<dataType> > os::asyncAVLTree< dataType  
>::find ( smart_ptr< adnode< dataType > > x ) [inline]
```

Finds by adnode node.

Finds a pointer to an object of type "dataType" given a comparison pointer to a node. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType> >)`. This function takes $O(\log(n))$ where n is the number of elements in the tree and will re-balance the tree

[in] x `os::adnode<dataType>` pointer to be compared against

Returns

true if the node was found and deleted, else false

```
template<class dataType> smart_ptr<asyncAVLNode<dataType> > os::asyncAVLTree<  
dataType >::find ( smart_ptr< asyncAVLNode< dataType > > x ) [inline]
```

Finds by `asyncAVLNode` (p. 101) node.

Finds a pointer to an object of type "dataType" given a comparison pointer to a node. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType> >)`. This function takes $O(\log(n))$ where n is the number of elements in the tree and will re-balance the tree

[in] x `os::asyncAVLNode<dataType>` pointer to be compared against

Returns

true if the node was found and deleted, else false

```
template<class dataType> smart_ptr<asyncAVLNode<dataType> > os::asyncAVLTree<  
dataType >::findBottom ( smart_ptr< asyncAVLNode< dataType > > x, int dir ) [inline],  
[protected]
```

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> bool os::asyncAVLTree< dataType >::findDelete ( smart_ptr<
dataType > x ) [inline], [virtual]
```

Finds and delete a matching node.

Finds a pointer to an object of type "dataType" given a comparison pointer and removes it. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType>> >)`. This function takes $O(\log(n))$ where n is the number of elements in the tree and will re-balance the tree

[in] x dataType pointer to be compared against

Returns

true if the node was found and deleted, else false

Reimplemented from **os::ads**< **dataType** > (p. 98).

```
template<class dataType> bool os::asyncAVLTree< dataType >::findDelete ( long x ) [inline]
```

Finds and delete a matching node.

Finds a pointer to an object of type "dataType" given a comparison pointer and removes it. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType>> >)`. This function takes $O(\log(n))$ where n is the number of elements in the tree and will re-balance the tree

[in] x dataType pointer to be compared against

Returns

true if the node was found and deleted, else false

```
template<class dataType> bool os::asyncAVLTree< dataType >::findDelete ( smart_ptr<
asyncAVLNode< dataType > > x ) [inline]
```

Finds and delete by node.

Finds a pointer to an object of type "dataType" given a comparison pointer to a node and removes it. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType>> >)`. This function takes $O(\log(n))$ where n is the number of elements in the tree and will re-balance the tree

[in] x `os::asyncAVLNode<dataType>` pointer to be compared against

Returns

true if the node was found and deleted, else false

```
template<class dataType> smart_ptr<adnode<dataType> > os::asyncAVLTree< dataType
>::getFirst( ) [inline], [virtual]
```

Returns the first node.

For the AVL tree, the first node is defined as the child at index 1. Note that while an `os::adnode<dataType>` is returned, the true type of the pointer returned is `os::asyncAVLNode<dataType>`. This function is $O(\log(n))$.

Returns

The first node, if it exists

Reimplemented from `os::ads< dataType >` (p. 99).

```
template<class dataType> smart_ptr<adnode<dataType> > os::asyncAVLTree< dataType
>::getLast( ) [inline], [virtual]
```

Returns the last node.

For the AVL tree, the last node is defined as the child at index 0. Note that while an `os::adnode<dataType>` is returned, the true type of the pointer returned is `os::asyncAVLNode<dataType>`. This function is $O(\log(n))$.

Returns

The last node, if it exists

Reimplemented from `os::ads< dataType >` (p. 99).

```
template<class dataType> smart_ptr<asyncAVLNode<dataType> > os::asyncAVLTree<
dataType >::getRoot( ) [inline]
```

Return the root of the tree.

Returns

`os::asyncAVLTree<dataType>::root` (p. 115)

```
template<class dataType> bool os::asyncAVLTree< dataType >::insert ( smart_ptr< ads<
dataType > > x ) [inline], [virtual]
```

Inserts an `os::ads<dataType>`

Inserts every element in a given abstract datastructure into this tree. Adopts the insertion function of `os::ads<dataType>`

[in] x pointer to `os::ads<dataType>`

Returns

true if successful, false if failed

Reimplemented from `os::ads< dataType >` (p. 100).

```
template<class dataType> bool os::asyncAVLTree< dataType >::insert ( smart_ptr< dataType >
x ) [inline], [virtual]
```

Inserts a data node.

Inserts a pointer to an object of type "dataType." This insertion will place the node into the binary tree and balance the tree. This function takes $O(\log(n))$ where n is the number of elements in the tree.

[in] x dataType pointer to be inserted

Returns

true if successful, false if failed

Reimplemented from **os::ads**< **dataType** > (p. 99).

```
template<class dataType> bool os::asyncAVLTree< dataType >::singleRotation ( smart_ptr<
asyncAVLNode< dataType > > r, int dir ) [inline], [protected]
```

Rotates a node.

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> virtual unsigned int os::asyncAVLTree< dataType >::size ( ) const
[inline], [virtual]
```

Finds and delete a matching node.

Returns

os::asyncAVLTree<**dataType**>::numElements (p. 115)

Reimplemented from **os::ads**< **dataType** > (p. 100).

12.4.4 Friends And Related Function Documentation

```
template<class dataType> friend class asyncAVLNode< dataType > [friend]
```

AVL Node must have access to mutex.

When the **AVLNode** (p. 115) finds the next element or finds the previous element, it must lock the mutex to prevent insertion and deletion into the tree.

12.4.5 Member Data Documentation

template<class dataType> std::mutex **os::asyncAVLTree**< dataType >::mtx [protected]

Mutex to ensure synchronous access.

template<class dataType> unsigned int **os::asyncAVLTree**< dataType >::numElements [protected]

Number of elements in the tree.

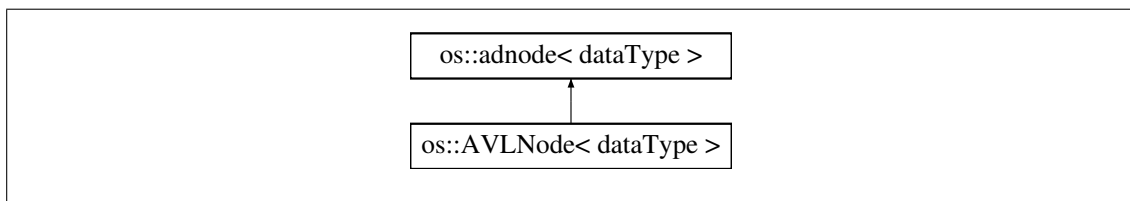
template<class dataType> **smart_ptr**<**asyncAVLNode**<dataType> > **os::asyncAVLTree**< dataType >::root [protected]

Root node of the tree.

12.5 os::AVLNode< dataType > Class Template Reference

Node for usage in an AVL tree.

Inheritance diagram for os::AVLNode< dataType >:



Public Member Functions

- **AVLNode** (**smart_ptr**< dataType > d)
Abstract data-node constructor.
- virtual **~AVLNode** ()
Virtual destructor.
- **smart_ptr**< **adnode**< dataType > > **getNext** ()
Find the next node.
- **smart_ptr**< **adnode**< dataType > > **getPrev** ()
Find the previous node.

Protected Member Functions

- **smart_ptr**< **AVLNode**< dataType > > **getParent** ()
Returns the parent node.
- **smart_ptr**< **AVLNode**< dataType > > **getChild** (int x)
Returns a child by index.
- int **getHeight** () const

- Returns the height of the sub-tree.*
- void **setHeight** ()
Sets the height of the sub-tree.
- void **setChild** (smart_ptr< AVLNode< dataType > > c)
Add a child to this node.
- void **setParent** (smart_ptr< AVLNode< dataType > > p, smart_ptr< AVLNode< dataType > > self_pointer)
Sets the parent node.
- void **removeChild** (smart_ptr< AVLNode< 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 **remove** ()
Remove all children and parents.

Protected Attributes

- smart_ptr< AVLNode< dataType > > **parent**
Parent node one level up in the tree.
- smart_ptr< AVLNode< dataType > > **child1**
Left child one level down in the tree.
- smart_ptr< AVLNode< dataType > > **child2**
Right child one level down in the tree.
- int **height**
The height of the tree.

Friends

- class **AVLTree< dataType >**
AVL Tree must know details of node implementation.

12.5.1 Detailed Description

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

Node for usage in an AVL tree.

The AVL node class implements a number of functions unique to an AVL tree. This node has knowledge of the structure of the AVL tree through its parent and children.

12.5.2 Constructor & Destructor Documentation

```
template<class dataType > os::AVLNode< dataType >::AVLNode ( smart_ptr< dataType > d )  
[inline]
```

Abstract data-node constructor.

An AVL node is meaningless without a pointer to it's dataType. The constructor requires this pointer to initialize the node. Parent and children nodes are, by default, initialized to 0.

Parameters

in	d	Data to be bound to the node
----	---	------------------------------

```
template<class dataType > virtual os::AVLNode< dataType >::~AVLNode ( ) [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.

12.5.3 Member Function Documentation

```
template<class dataType > smart_ptr<AVLNode<dataType> > os::AVLNode< dataType  
>::getChild ( int x ) [inline], [protected]
```

Returns a child by index.

Returns child node by index. 0 indicates the left child, **AVLNode**<dataType>::child1 (p. 120). 1 indicates the right child, **AVLNode**<dataType>::child2 (p. 120). All other indices will return NULL.

Returns

os::AVLNode<dataType>::child1 (p. 120) for x==0, **AVLNode**<dataType>::child2 (p. 120) for x==1

```
template<class dataType > int os::AVLNode< dataType >::getHeight ( ) const [inline],  
[protected]
```

Returns the height of the sub-tree.

Returns

os::AVLNode<dataType>::height (p. 120)

```
template<class dataType > smart_ptr<adnode<dataType> > os::AVLNode< dataType >::getNext  
( ) [inline], [virtual]
```

Find the next node.

This functions attempts to search for the next node in the structure. This trips the traverse flag of the current node and traverses the tree looking for the next node.

Returns

Pointer to the next node in the structure

Reimplemented from **os::adnode< dataType >** (p. 96).

```
template<class dataType > smart_ptr<AVLNode<dataType> > os::AVLNode< dataType  
>::getParent ( ) [inline], [protected]
```

Returns the parent node.

Returns

os::AVLNode<dataType>::parent (p. 120)

```
template<class dataType > smart_ptr<adnode<dataType> > os::AVLNode< dataType >::getPrev  
( ) [inline], [virtual]
```

Find the previous node.

This functions attempts to search for the previous node in the structure. This trips the traverse flag of the current node and traverses the tree looking for the previous node.

Returns

Pointer to the previous node in the structure

Reimplemented from **os::adnode< dataType >** (p. 96).

```
template<class dataType > void os::AVLNode< dataType >::remove ( ) [inline],  
[protected]
```

Remove all children and parents.

This function is important because nodes are of type **os::smart_ptr** (p. 151), 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 > void os::AVLNode< dataType >::removeChild ( smart_ptr<  
AVLNode< dataType > > c ) [inline], [protected]
```

Remove a child from this node.

Checks **os::AVLNode<dataType>::child1** (p. 120) and **os::AVLNode<dataType>::child2** (p. 120) for equality with the the node received as a parameter.

Parameters

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

Returns

void

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

Remove a child from this node.

Remove **os::AVLNode<dataType>::child1** (p. 120) if position is 0 and **os::AVLNode<dataType>::child2** (p. 120) if position is 1.

Parameters

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

Returns

void

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

Remove the parent node.

Returns

void

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

Add a child to this node.

Set **os::AVLNode<dataType>::child1** (p. 120) or **os::AVLNode<dataType>::child2** (p. 120) 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::AVLNode< dataType >::setHeight ( ) [inline],  
[protected]
```

Sets the height of the sub-tree.

Uses the height of the sub-tree of the node's children to calculate the height of the sub-tree of this node.

Returns

void

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

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

12.5.4 Friends And Related Function Documentation

```
template<class dataType > friend class AVLTree< dataType > [friend]
```

AVL Tree must know details of node implementation.

Since the AVL node implements many of the unique functions of the AVL tree, the tree must be aware of the private members of it's nodes.

12.5.5 Member Data Documentation

```
template<class dataType > smart_ptr<AVLNode<dataType> > os::AVLNode< dataType
>::child1 [protected]
```

Left child one level down in the tree.

```
template<class dataType > smart_ptr<AVLNode<dataType> > os::AVLNode< dataType
>::child2 [protected]
```

Right child one level down in the tree.

```
template<class dataType > int os::AVLNode< dataType >::height [protected]
```

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 **AVLNode**<**dataType**>::setHeight() (p. 119) resets the height based on the height of the node's children.

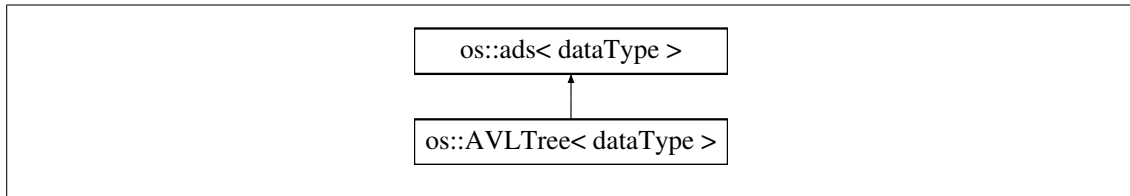
```
template<class dataType > smart_ptr<AVLNode<dataType> > os::AVLNode< dataType
>::parent [protected]
```

Parent node one level up in the tree.

12.6 os::AVLTree< dataType > Class Template Reference

Balanced binary search tree.

Inheritance diagram for os::AVLTree< dataType >:



Public Member Functions

- **AVLTree ()**
Default constructor.
- virtual **~AVLTree ()**
Virtual destructor.
- bool **insert (smart_ptr< ads< dataType > > x)**
Inserts an os::ads<dataType>
- bool **insert (smart_ptr< dataType > x)**
Inserts a data node.
- smart_ptr< AVLNode< dataType > > **getRoot ()**
Return the root of the tree.
- smart_ptr< adnode< dataType > > **find (smart_ptr< dataType > x)**
Finds a matching node.
- smart_ptr< adnode< dataType > > **find (smart_ptr< adnode< dataType > > x)**
Finds by adnode node.
- smart_ptr< AVLNode< dataType > > **find (smart_ptr< AVLNode< dataType > > x)**
Finds by AVLNode (p. 115) node.
- bool **findDelete (smart_ptr< dataType > x)**
Finds and delete a matching node.
- bool **findDelete (smart_ptr< AVLNode< dataType > > x)**
Finds and delete by node.
- virtual unsigned int **size () const**
Finds and delete a matching node.
- smart_ptr< adnode< dataType > > **getFirst ()**
Returns the first node.
- smart_ptr< adnode< dataType > > **getLast ()**
Returns the last node.

Protected Member Functions

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

Protected Attributes

- **smart_ptr< AVLNode< dataType > > root**
Root node of the tree.
- **unsigned int numElements**
Number of elements in the tree.

12.6.1 Detailed Description

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

Balanced binary search tree.

The AVL Tree rigorously balances a binary search tree. As a template class, it can hold any kind of dataType so long as the data type implements basic comparison functions.

12.6.2 Constructor & Destructor Documentation

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

Default constructor.

Sets the number of elements to 0 and the root to NULL.

```
template<class dataType > virtual os::AVLTree< dataType >::~~AVLTree ( ) [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. The AVL tree must explicitly force deletion through the **AVLNode<dataType>::remove()** (p. 118) function.

12.6.3 Member Function Documentation

```
template<class dataType > bool os::AVLTree< dataType >::balance ( smart_ptr< AVLNode<
dataType > > x ) [inline], [protected]
```

Balances a single node.

Parameters

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

Returns

true if the node is already balanced, else, false

```
template<class dataType > bool os::AVLTree< dataType >::balanceDelete ( smart_ptr<
AVLNode< dataType > > x ) [inline], [protected]
```

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::AVLTree< dataType >::balanceUp ( smart_ptr< AVLNode<
dataType > > x ) [inline], [protected]
```

Balances this node and ancestor nodes.

Balances the current node then orders its 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::AVLTree< dataType >::checkBalance ( smart_ptr<
AVLNode< dataType > > x ) [inline], [protected]
```

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::AVLTree< dataType >::doubleRotation ( smart_ptr<
AVLNode< dataType > > r, int dir ) [inline], [protected]
```

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 > smart_ptr<adnode<dataType> > os::AVLTree< dataType >::find (
smart_ptr< dataType > x ) [inline], [virtual]
```

Finds a matching node.

Finds a pointer to an object of type "dataType" given a comparison pointer. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType> >)`. This function takes $O(\log(n))$ where n is the number of elements in the tree.

[in] x dataType pointer to be compared against

Returns

true if the node was found, else false

Reimplemented from `os::ads< dataType >` (p. 98).

```
template<class dataType > smart_ptr<adnode<dataType> > os::AVLTree< dataType >::find (
smart_ptr< adnode< dataType > > x ) [inline]
```

Finds by adnode node.

Finds a pointer to an object of type "dataType" given a comparison pointer to a node. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType> >)`. This function takes $O(\log(n))$ where n is the number of elements in the tree and will re-balance the tree

[in] x `os::adnode<dataType>` pointer to be compared against

Returns

true if the node was found and deleted, else false

```
template<class dataType > smart_ptr<AVLNode<dataType> > os::AVLTree< dataType >::find (
smart_ptr< AVLNode< dataType > > x ) [inline]
```

Finds by **AVLNode** (p. 115) node.

Finds a pointer to an object of type "dataType" given a comparison pointer to a node. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType> >)`. This function takes $O(\log(n))$ where n is the number of elements in the tree and will re-balance the tree

[in] x `os::AVLNode<dataType>` pointer to be compared against

Returns

true if the node was found and deleted, else false

```
template<class dataType > smart_ptr<AVLNode<dataType> > os::AVLTree< dataType
>::findBottom ( smart_ptr< AVLNode< dataType > > x, int dir ) [inline], [protected]
```

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	x	Starting node
in	dir	Direction node to search in

Returns

First or last node in sub-tree

```
template<class dataType > bool os::AVLTree< dataType >::findDelete ( smart_ptr< dataType > x
) [inline], [virtual]
```

Finds and delete a matching node.

Finds a pointer to an object of type "dataType" given a comparison pointer and removes it. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType> >)`. This function takes $O(\log(n))$ where n is the number of elements in the tree and will re-balance the tree

[in] x `dataType` pointer to be compared against

Returns

true if the node was found and deleted, else false

Reimplemented from `os::ads< dataType >` (p. 98).

```
template<class dataType > bool os::AVLTree< dataType >::findDelete ( smart_ptr< AVLNode<
dataType > > x ) [inline]
```

Finds and delete by node.

Finds a pointer to an object of type "dataType" given a comparison pointer to a node and removes it. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType> >)`. This function takes $O(\log(n))$ where n is the number of elements in the tree and will re-balance the tree

[in] x `os::AVLNode<dataType>` pointer to be compared against

Returns

true if the node was found and deleted, else false

```
template<class dataType > smart_ptr<adnode<dataType> > os::AVLTree< dataType >::getFirst (
) [inline], [virtual]
```

Returns the first node.

For the AVL tree, the first node is defined as the child at index 1. Note that while an `os::adnode<dataType>` is returned, the true type of the pointer returned is `os::AVLNode<dataType>`. This function is $O(\log(n))$.

Returns

The first node, if it exists

Reimplemented from `os::ads< dataType >` (p. 99).

```
template<class dataType > smart_ptr<adnode<dataType> > os::AVLTree< dataType >::getLast (
) [inline], [virtual]
```

Returns the last node.

For the AVL tree, the last node is defined as the child at index 0. Note that while an `os::adnode<dataType>` is returned, the true type of the pointer returned is `os::AVLNode<dataType>`. This function is $O(\log(n))$.

Returns

The last node, if it exists

Reimplemented from `os::ads< dataType >` (p. 99).

```
template<class dataType > smart_ptr<AVLNode<dataType> > os::AVLTree< dataType
>::getRoot ( ) [inline]
```

Return the root of the tree.

Returns

`os::AVLTree<dataType>::root` (p. 128)

```
template<class dataType > bool os::AVLTree< dataType >::insert ( smart_ptr< ads< dataType > > x ) [inline], [virtual]
```

Inserts an **os::ads**<dataType>

Inserts every element in a given abstract datastructure into this tree. Adopts the insertion function of **os::ads**<dataType>

[in] x pointer to **os::ads**<dataType>

Returns

true if successful, false if failed

Reimplemented from **os::ads**< **dataType** > (p. 100).

```
template<class dataType > bool os::AVLTree< dataType >::insert ( smart_ptr< dataType > x ) [inline], [virtual]
```

Inserts a data node.

Inserts a pointer to an object of type "dataType." This insertion will place the node into the binary tree and balance the tree. This function takes $O(\log(n))$ where n is the number of elements in the tree.

[in] x dataType pointer to be inserted

Returns

true if successful, false if failed

Reimplemented from **os::ads**< **dataType** > (p. 99).

```
template<class dataType > bool os::AVLTree< dataType >::singleRotation ( smart_ptr< AVLNode< dataType > > r, int dir ) [inline], [protected]
```

Rotates a node.

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 > virtual unsigned int os::AVLTree< dataType >::size ( ) const [inline], [virtual]
```

Finds and delete a matching node.

Returns

os::AVLTree<dataType>::numElements (p. 128)

Reimplemented from **os::ads< dataType >** (p. 100).

12.6.4 Member Data Documentation

template<class dataType > unsigned int **os::AVLTree< dataType >::numElements** [protected]

Number of elements in the tree.

template<class dataType > **smart_ptr<AVLNode<dataType> > os::AVLTree< dataType >::root** [protected]

Root node of the tree.

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 int 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 int 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 int 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. 132). 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	str	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. 132)

```
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. 132).

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. 132) 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 int 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::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`

12.8.1 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.

12.8.2 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.

12.8.3 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. 135).

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

12.8.4 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.

12.8.5 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.

12.9 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 int arg, void *data)
Receive event notification, flexible data.
- void **triggerEvent** ()
Sends an event to all receivers.
- void **triggerEvent** (unsigned int 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.

12.9.1 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.

12.9.2 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.

12.9.3 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. 132).

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 int 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. 137) function with each receiver as an argument.

Returns

void

```
template<class receiverType > void os::eventSender< receiverType >::triggerEvent ( unsigned int  
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. 137) function with each receiver as an argument. This event trigger passes data arguments.

Parameters

in	<i>arg</i>	Event flag
in	<i>data</i>	Void pointer to data

Returns

void

12.9.4 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.

12.9.5 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.

12.10 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.

12.10.1 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<dataType>`.

12.10.2 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
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.

12.10.3 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	<i>w</i>	X position
in	<i>h</i>	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. 144)

```
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. 144)

```
template<class dataType> uint32_t os::indirectMatrix< dataType >::height ( ) const throw )  
[inline]
```

Return _height of matrix.

Returns

indirectMatrix<dataType>::_height (p. 144)

```
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	w	X position
in	h	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	m	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

<code>in</code>	<code>m</code>	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. 144)

12.10.4 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.

12.10.5 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`.

12.11 `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.

12.11.1 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>`.

12.11.2 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
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.

12.11.3 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. 150)

```
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. 150)

```
template<class dataType> uint32_t os::matrix< dataType >::height ( ) const throw ) [inline]
```

Return `_height` of matrix.

Returns

matrix<dataType>::_height (p. 150)

```
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	<i>m</i>	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	<i>m</i>	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. 150)

12.11.4 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 and indirect matrix can be constructed from a raw matrix.

12.11.5 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`.

12.12 os::ptrComp Class Reference

Pointer compare interface.

Public Member Functions

- virtual **~ptrComp** ()
Virtual destructor.
- virtual bool **operator==** (const **ptrComp** &l) const
Equality test.
- virtual bool **operator>** (const **ptrComp** &l) const
Greater than test.
- virtual bool **operator<** (const **ptrComp** &l) const

Less than test.

- virtual bool **operator>=** (const **ptrComp** &l) const

Greater than/equal to test.

- virtual bool **operator<=** (const **ptrComp** &l) const

Less than/equal to test.

12.12.1 Detailed Description

Pointer compare interface.

Allows a class which does not define comparison operators to be placed into an abstract data-structure by defining comparison to be address comparison.

12.12.2 Constructor & Destructor Documentation

```
virtual os::ptrComp::~~ptrComp ( ) [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.

12.12.3 Member Function Documentation

```
virtual bool os::ptrComp::operator< ( const ptrComp &l ) const [inline], [virtual]
```

Less than test.

```
virtual bool os::ptrComp::operator<= ( const ptrComp &l ) const [inline], [virtual]
```

Less than/equal to test.

```
virtual bool os::ptrComp::operator== ( const ptrComp &l ) const [inline], [virtual]
```

Equality test.

```
virtual bool os::ptrComp::operator> ( const ptrComp &l ) const [inline], [virtual]
```

Greater than test.

```
virtual bool os::ptrComp::operator>= ( const ptrComp &l ) const [inline], [virtual]
```

Greater than/equal to test.

12.13 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 data↵
Type *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 int i) throw ()`
Array de-reference.
- `const dataType & operator[] (unsigned int 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. 151).
- `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.

12.13.1 Detailed Description

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

Reference counted pointer.

The **os::smart_ptr** (p. 151) template class allows for automatic memory management. **os::smart_ptr** (p. 151)'s have a type defined by **os::smart_pointer_type** (p. 87) which defines the copy and deletion behaviour of the object.

12.13.2 Constructor & Destructor Documentation

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

Default constructor.

Constructs an **os::smart_ptr** (p. 151) of type **os::null_type** (p. 87). 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. 151) 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. 151) from an existing **os::smart_ptr** (p. 151). Will increment the reference count as defined by the received **os::smart_pointer_type** (p. 87).

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. 151) from a raw pointer and a type. This is the most commonly used **os::smart_ptr** (p. 151) constructor, other than the copy constructor. Note that **os::shared_type_dynamic_delete** (p. 88) 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. 151) from a raw pointer and a destruction function. This constructor generates an **os::smart_ptr** (p. 151) of type **os::shared_type_dynamic_delete** (p. 88).

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. 162) 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. 151) 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. 151) from an 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. 151) 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
----	-----------	-----------------------------------

12.13.3 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. 151) from an existing **os::smart_ptr** (p. 151). Will increment the reference count as defined by the received **os::smart_pointer_type** (p. 87).

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. 151) from a dataType pointer. This new **os::smart_ptr** (p. 151) will

be of type **os::raw_type** (p. 87) unless the dataType pointer is NULL, then it will be of type **os::null_type** (p. 87).

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. 151).

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. 163) of the objects does not factor into this comparison.

Parameters

in	<i>c</i>	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	<i>c</i>	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. 151).

Returns

dataType* in constant form, **os::smart_ptr**<dataType>::raw_ptr (p. 162)

```
template<class dataType> dataType* os::smart_ptr< dataType >::get ( ) throw ) [inline]
```

Return data.

Returns the dataType pointer of the **os::smart_ptr** (p. 151).

Returns

dataType* in modifiable form, **os::smart_ptr**<dataType>::raw_ptr (p. 162)

```
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. 151).

Returns

dataType* in constant form, **os::smart_ptr**<dataType>::raw_ptr (p. 162)

```
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. 88) mode)

Returns

os::void_rec (p. 87) **os::smart_ptr**<dataType>::func (p. 162)

```
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. 163)

```
template<class dataType> smart_pointer_type os::smart_ptr< dataType >::getType ( ) const  
throw ) [inline]
```

Return type.

Returns the **os::smart_pointer_type** (p. 87) of the **os::smart_ptr** (p. 151).

Returns

os::smart_pointer_type (p. 87) **os::smart_ptr<dataType>::type** (p. 163)

```
template<class dataType> os::smart_ptr< dataType >::operator bool ( ) const throw )  
[inline]
```

Boolean conversion.

Returns

os::smart_ptr<dataType>::raw_ptr (p. 162) 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. 162) 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. 162) 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. 162) 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. 162) 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. 162) de-referenced

```
template<class dataType> dataType* os::smart_ptr< dataType >::operator-> ( ) throw )  
[inline]
```

Pointer pass.

Returns

os::smart_ptr<dataType>::raw_ptr (p. 162)

```
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. 162)

```
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. 156).

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. 156).

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. 156) 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. 156) 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. 156) 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 int i )  
throw ) [inline]
```

Array de-reference.

Returns

dataType reference of **os::smart_ptr**<dataType>::raw_ptr (p. 162) incremented i de-referenced

```
template<class dataType> const dataType& os::smart_ptr< dataType >::operator[] ( unsigned int  
i ) const throw ) [inline]
```

Constant array de-reference.

Returns

Constant dataType reference of **os::smart_ptr**<dataType>::raw_ptr (p. 162) 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. 151). Decrements the reference counter, if not of **os::raw_type** (p. 87) or **os::null_type** (p. 87), and delete **os::smart_ptr**<dataType>::raw_ptr (p. 162) if needed. Note that if **os::smart_ptr**<dataType>::raw_ptr (p. 162) is deleted, so is **os::smart_ptr**<dataType>::ref_count (p. 163).

Returns

void

12.13.4 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. 151) is of type **os::shared_type** or **os::dynamic_delete** (p. 88).

```
template<class dataType> dataType* os::smart_ptr< dataType >::raw_ptr [private]
```

Pointer to data.

The **os::smart_ptr**<dataType>::raw_ptr (p. 162) holds the pointer to the block of memory to be managed by the **os::smart_ptr** (p. 151). If this pointer is NULL, the **os::smart_ptr** (p. 151) is of type **os::null_type** (p. 87).

```
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. 151). Note that all **os::smart_ptr** (p. 151)'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. 151) is either of type **os::null_type** (p. 87) or **os::raw_type** (p. 87).

```
template<class dataType> smart_pointer_type os::smart_ptr< dataType >::type [private]
```

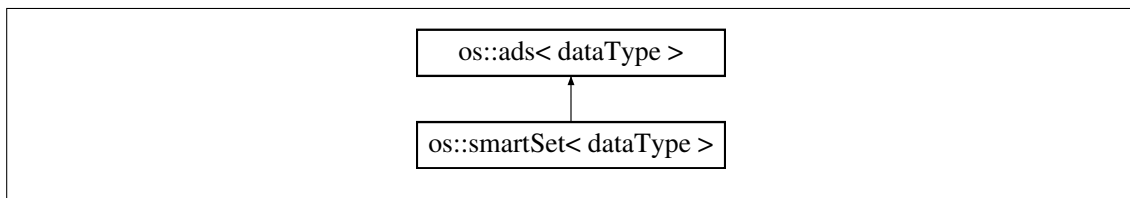
Stores the type.

Defines the type of the **os::smart_ptr** (p. 151). See **os::smart_pointer_type** (p. 87) for details on the available types.

12.14 os::smartSet< dataType > Class Template Reference

Smart set abstract data-structures.

Inheritance diagram for **os::smartSet< dataType >**:



Public Member Functions

- **smartSet** (**setTypes** typ=**def_set**)
Default constructor.
- virtual **~smartSet** ()
Virtual destructor.
- void **rebuild** (**setTypes** typ)
Set set type.
- **setTypes** **getType** () const
Return set type.
- bool **insert** (**smart_ptr**< **ads**< dataType > > x)
Inserts an os::ads<dataType>
- bool **insert** (**smart_ptr**< dataType > x)
Inserts a data node.
- **smart_ptr**< **adnode**< dataType > > **find** (**smart_ptr**< dataType > x)
Finds a matching node.
- bool **findDelete** (**smart_ptr**< dataType > x)
Finds and delete a matching node.

- unsigned int **size** () const
Returns the number of elements in the set.
- **smart_ptr**< **adnode**< dataType > > **getFirst** ()
Return the first element.
- **smart_ptr**< **adnode**< dataType > > **getLast** ()
Return the last element.

Private Member Functions

- void **build** (**setTypes** typ)

Private Attributes

- **setTypes** type
Stores the set type.
- **smart_ptr**< **ads**< dataType > > **current_struct**
Abstract data-structure storing data.

Additional Inherited Members

12.14.1 Detailed Description

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

Smart set abstract data-structures.

Wraps other forms of abstract data structures, allowing applications to define abstract data-structures by numbered indexes.

12.14.2 Constructor & Destructor Documentation

```
template<class dataType > os::smartSet< dataType >::smartSet ( setTypes typ = def_set )
[inline]
```

Default constructor.

This constructor builds the smart set based on a set type. Will call **os::smartSet<dataType>::build** (p. 165).

Parameters

in	typ	Set type, default is os::def_set (p. 87)
----	-----	---

```
template<class dataType > virtual os::smartSet< dataType >::~smartSet ( ) [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.

12.14.3 Member Function Documentation

```
template<class dataType > void os::smartSet< dataType >::build ( setTypes typ ) [inline],  
[private]
```

```
template<class dataType > smart_ptr<adnode<dataType> > os::smartSet< dataType >::find ( smart_ptr< dataType > x ) [inline], [virtual]
```

Finds a matching node.

Finds a pointer to an object of type "dataType" given a comparison pointer. Adopts the find function of the abstract data-structure used for this set type. If no abstract data-structure exists, return false.

[in] x dataType pointer to be compared against

Returns

true if the node was found, else false

Reimplemented from **os::ads**< **dataType** > (p. 98).

```
template<class dataType > bool os::smartSet< dataType >::findDelete ( smart_ptr< dataType > x  
) [inline], [virtual]
```

Finds and delete a matching node.

Finds a pointer to an object of type "dataType" given a comparison pointer and remove it. Adopts the findDelete function of the abstract data-structure used for this set type. If no abstract data-structure exists, return false.

[in] x dataType pointer to be compared against

Returns

true if the node was found, else false

Reimplemented from **os::ads**< **dataType** > (p. 98).

```
template<class dataType > smart_ptr<adnode<dataType> > os::smartSet< dataType >::getFirst  
( ) [inline], [virtual]
```

Return the first element.

Adopts the getFirst function of the abstract data-structure used for this set type. If no abstract data-structure exists, return NULL.

Returns

os::smartSet<**dataType**>::current_struct (p. 167)->getFirst() (p. 165)

Reimplemented from **os::ads**< **dataType** > (p. 99).

```
template<class dataType > smart_ptr<adnode<dataType> > os::smartSet< dataType >::getLast  
( ) [inline], [virtual]
```

Return the last element.

Adopts the getLast function of the abstract data-structure used for this set type. If no abstract data-structure exists, return NULL.

Returns

os::smartSet<**dataType**>::**current_struct** (p. 167)->**getLast()** (p. 166)

Reimplemented from **os::ads**< **dataType** > (p. 99).

```
template<class dataType > setTypes os::smartSet< dataType >::getType ( ) const [inline]
```

Return set type.

Returns

os::smartSet<**dataType**>::**type** (p. 167)

```
template<class dataType > bool os::smartSet< dataType >::insert ( smart_ptr< ads< dataType >  
> x ) [inline], [virtual]
```

Inserts an **os::ads**<**dataType**>

Inserts every element in a given abstract datastructure into this tree. Adopts the insertion function of **os::ads**<**dataType**>

[in] x pointer to **os::ads**<**dataType**>

Returns

true if successful, false if failed

Reimplemented from **os::ads**< **dataType** > (p. 100).

```
template<class dataType > bool os::smartSet< dataType >::insert ( smart_ptr< dataType > x )  
[inline], [virtual]
```

Inserts a data node.

Adopts the insertion function of the abstract data-structure used for this set type. If no abstract data-structure exists, return false.

[in] x dataType pointer to be inserted

Returns

true if successful, false if failed

Reimplemented from **os::ads**< **dataType** > (p. 99).

```
template<class dataType > void os::smartSet< dataType >::rebuild ( setTypes typ ) [inline]
```

Set set type.

Sets the type of the set, rebuilding the set if the requested type and current type do not match.

Parameters

in	type	Set type
----	------	----------

Returns

void

```
template<class dataType > unsigned int os::smartSet< dataType >::size ( ) const [inline],  
[virtual]
```

Returns the number of elements in the set.

Adopts the size function of the abstract data-structure used for this set type. If no abstract data-structure exists, return 0.

Returns

os::smartSet<dataType>::current_struct (p. 167)->**size()** (p. 167)

Reimplemented from **os::ads< dataType >** (p. 100).

12.14.4 Member Data Documentation

```
template<class dataType > smart_ptr<ads<dataType> > os::smartSet< dataType  
>::current_struct [private]
```

Abstract data-structure storing data.

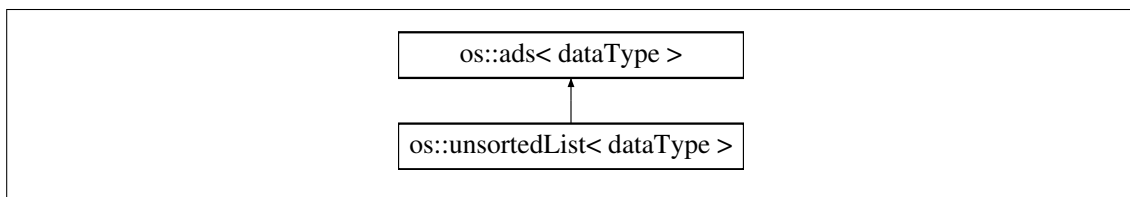
```
template<class dataType > setTypes os::smartSet< dataType >::type [private]
```

Stores the set type.

12.15 os::unsortedList< dataType > Class Template Reference

Unsorted linked list.

Inheritance diagram for **os::unsortedList< dataType >**:



Public Member Functions

- **unorderedList ()**
Default constructor.
- virtual **~unorderedList ()**
Virtual destructor.
- bool **insert (smart_ptr< ads< dataType > > x)**
Inserts an os::ads<dataType>
- bool **insert (smart_ptr< dataType > x)**
Inserts a data node.
- virtual unsigned int **size () const**
Returns the number of elements in the list.
- **smart_ptr< adnode< dataType > > find (smart_ptr< dataType > x)**
Finds a matching node.
- bool **findDelete (smart_ptr< dataType > x)**
Finds and delete a matching node.
- **smart_ptr< adnode< dataType > > getFirst ()**
Return the head.
- **smart_ptr< adnode< dataType > > getLast ()**
Return the tail.

Private Attributes

- **smart_ptr< unorderedListNode< dataType > > head**
Head node.
- **smart_ptr< unorderedListNode< dataType > > tail**
Tail node.
- unsigned int **_size**
Number of elements in the list.

Additional Inherited Members

12.15.1 Detailed Description

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

Unsorted linked list.

The list defined by this class is searchable but unsorted. Insert checks to see if the element being inserted is already contained inside the list. Elements are inserted from the front of the list.

12.15.2 Constructor & Destructor Documentation

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

Default constructor.

Sets the number of elements to 0 and the head and tail to NULL.

```
template<class dataType > virtual os::unsortedList< dataType >::~unsortedList ( ) [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. The list must explicitly force deletion through setting all of the next and previous references of nodes to NULL.

12.15.3 Member Function Documentation

```
template<class dataType > smart_ptr<adnode<dataType> > os::unsortedList< dataType >::find  
( smart_ptr< dataType > x ) [inline], [virtual]
```

Finds a matching node.

Finds a pointer to an object of type "dataType" given a comparison pointer. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType> >)`. This function takes $O(n)$ where n is the number of elements in the list.

[in] x dataType pointer to be compared against

Returns

true if the node was found, else false

Reimplemented from `os::ads< dataType >` (p. 98).

```
template<class dataType > bool os::unsortedList< dataType >::findDelete ( smart_ptr< dataType  
> x ) [inline], [virtual]
```

Finds and delete a matching node.

Finds a pointer to an object of type "dataType" given a comparison pointer and removes it. This comparison function is defined by `os::adnode<dataType>::compare(smart_ptr<adnode<dataType> >)`. This function takes $O(n)$ where n is the number of elements in the list.

[in] x dataType pointer to be compared against

Returns

true if the node was found, else false

Reimplemented from `os::ads< dataType >` (p. 98).

```
template<class dataType > smart_ptr<adnode<dataType> > os::unsortedList< dataType  
>::getFirst ( ) [inline], [virtual]
```

Return the head.

This function is $O(1)$

Returns

`os::unsortedList<dataType>::head` (p. 171)

Reimplemented from `os::ads< dataType >` (p. 99).

```
template<class dataType > smart_ptr<adnode<dataType> > os::unsortedList< dataType  
>::getLast ( ) [inline], [virtual]
```

Return the tail.

This function is O(1).

Returns

os::unsortedList<**dataType**>::tail (p. 171)

Reimplemented from **os::ads**< **dataType** > (p. 99).

```
template<class dataType > bool os::unsortedList< dataType >::insert ( smart_ptr< ads<  
dataType > > x ) [inline], [virtual]
```

Inserts an **os::ads**<**dataType**>

Inserts every element in a given abstract datastructure into this tree. Adopts the insertion function of **os::ads**<**dataType**>

[in] x pointer to **os::ads**<**dataType**>

Returns

true if successful, false if failed

Reimplemented from **os::ads**< **dataType** > (p. 100).

```
template<class dataType > bool os::unsortedList< dataType >::insert ( smart_ptr< dataType > x  
) [inline], [virtual]
```

Inserts a data node.

Inserts a pointer to an object of type "dataType." This insertion will place the node into the list at the beginning. If the node already exists, it will not be inserted. This means that this function must first attempt to find the node being inserted. This function is O(n).

[in] x dataType pointer to be inserted

Returns

true if successful, false if failed

Reimplemented from **os::ads**< **dataType** > (p. 99).

```
template<class dataType > virtual unsigned int os::unsortedList< dataType >::size ( ) const  
[inline], [virtual]
```

Returns the number of elements in the list.

Returns

os::unsortedList<**dataType**>::numElements

Reimplemented from **os::ads**< **dataType** > (p. 100).

12.15.4 Member Data Documentation

template<class dataType > unsigned int **os::unsortedList**< dataType >::_size [private]

Number of elements in the list.

template<class dataType > **smart_ptr**<**unsortedListNode**<dataType> > **os::unsortedList**< dataType >::head [private]

Head node.

Contains a pointer to the head node in the list. If this node is NULL, the list is empty.

template<class dataType > **smart_ptr**<**unsortedListNode**<dataType> > **os::unsortedList**< dataType >::tail [private]

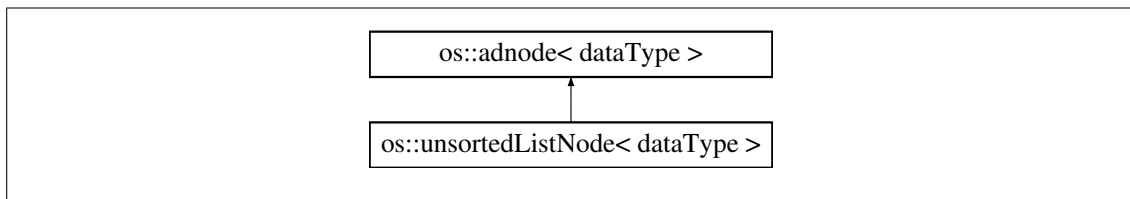
Tail node.

Contains a pointer to the tail node in the list. If this node is NULL, the list is empty.

12.16 os::unsortedListNode< dataType > Class Template Reference

Node for usage in a linked list.

Inheritance diagram for os::unsortedListNode< dataType >:



Public Member Functions

- **unsortedListNode** (**smart_ptr**< dataType > d)
Abstract data-node constructor.
- virtual **~unsortedListNode** ()
Virtual destructor.
- **smart_ptr**< **adnode**< dataType > > **getNext** ()
Return the next node.
- **smart_ptr**< **adnode**< dataType > > **getPrev** ()
Return the previous node.

Protected Member Functions

- void **remove** ()
Remove this node from the list.

Protected Attributes

- **smart_ptr< unsortedListNode< dataType > > prev**
Previous node.
- **smart_ptr< unsortedListNode< dataType > > next**
Next node.

Friends

- class **unsortedList< dataType >**
List aware of it's nodes.

12.16.1 Detailed Description

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

Node for usage in a linked list.

This class is a simple extension of the `os::adnode<dataType>` class. It holds the previous and next node inside of it as well as a pointer to its data. Note that the `os::unsortedList<dataType>` class implements the mechanics of the list.

12.16.2 Constructor & Destructor Documentation

```
template<class dataType > os::unsortedListNode< dataType >::unsortedListNode (
smart_ptr< dataType > d ) [inline]
```

Abstract data-node constructor.

A list node is meaningless without a pointer to it's `dataType`. The constructor requires this pointer to initialize the node. Next and previous nodes are, by default, initialized to zero.

Parameters

in	<i>d</i>	Data to be bound to the node
----	----------	------------------------------

```
template<class dataType > virtual os::unsortedListNode< dataType >::~unsortedListNode ( )
[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 > smart_ptr<adnode<dataType> > os::unsortedListNode< dataType
>::getNext ( ) [inline], [virtual]
```

Return the next node.

Note that **os::unsortedListNode<dataType>::next** (p. 173) is of type **os::unsortedListNode<dataType>**, but this function returns type of **os::adnode<dataType>**. **os::unsortedListNode<dataType>::next** (p. 173) must be case before returning.

Returns

os::unsortedListNode<dataType>::next (p. 173)

Reimplemented from **os::adnode< dataType >** (p. 96).

```
template<class dataType > smart_ptr<adnode<dataType> > os::unsortedListNode< dataType  
>::getPrev ( ) [inline], [virtual]
```

Return the previous node.

Note that **os::unsortedListNode<dataType>::prev** (p. 174) is of type **os::unsortedListNode<dataType>**, but this function returns type of **os::adnode<dataType>**. **os::unsortedListNode<dataType>::prev** (p. 174) must be case before returning.

Returns

os::unsortedListNode<dataType>::prev (p. 174)

Reimplemented from **os::adnode< dataType >** (p. 96).

```
template<class dataType > void os::unsortedListNode< dataType >::remove ( ) [inline],  
[protected]
```

Remove this node from the list.

Removes the references to this node from the next and previous node, if they exists. Sets the previous and next nodes to NULL.

Returns

void

12.16.4 Friends And Related Function Documentation

```
template<class dataType > friend class unsortedList< dataType > [friend]
```

List aware of it's nodes.

The unsorted list must be aware of the inner-workings of its nodes. Only the unsorted list is permitted to access the private members of this class.

12.16.5 Member Data Documentation

```
template<class dataType > smart_ptr<unsortedListNode<dataType> > os::unsortedListNode<  
dataType >::next [protected]
```

Next node.

Contains a pointer to the next node in the list. If this node is the tail of the list, the next node is NULL.

```
template<class dataType > smart_ptr<unsortedListNode<dataType> > os::unsortedListNode<
dataType >::prev [protected]
```

Previous node.

Contains a pointer to the previous node in the list. If this node is the head of the list, the previous node is NULL.

12.17 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.

12.17.1 Detailed Description

```
template<class dataType>
class os::vector2d< dataType >
```

2-dimensional vector

This template class contains the functions and operators needed to perform arithmetic on a 2 dimensional vector

12.17.2 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.

12.17.3 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	vec	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	vec	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	dummy	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	vec	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	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]
```

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	target	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	vec	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	vec	Reference to vector to be subtracted
----	-----	--------------------------------------

Returns

Reference to self

12.17.4 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.

12.18 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.

12.18.1 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

12.18.2 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.

12.18.3 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

12.18.4 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.