Unit Test Documentation

Adrian Bedard Jonathan Bedard

September 5, 2016

Contents

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:

Chapter 3

File Documentation

3.1 DatastructuresTest.cpp File Reference

Datastructures library test implementation.

3.1.1 Detailed Description

Datastructures library test implementation.

Author

Jonathan Bedard

Date

8/6/2016

Bug No known bugs.

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

3.2 DatastructuresTest.h File Reference

Datastructures library test.

3.2.1 Detailed Description

Datastructures library test.

Author

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

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

3.3 defaultTestInit.cpp File Reference

Default UnitTest initializer function.

3.3.1 Detailed Description

Default UnitTest initializer function.

Author

Jonathan Bedard

Date

2/12/2016

Bug No known bugs.

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

3.4 libraryTests.cpp File Reference

Library tests implementations.

3.4.1 Detailed Description

Library tests implementations.
Jonathan Bedard

Date

8/12/2016

Bug No known bugs.

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

3.5 libraryTests.h File Reference

Library tests class.

Classes

- class **test::libraryTests**
  Library test group.

Namespaces

- **test**

3.5.1 Detailed Description

Library tests class.
Jonathan Bedard

Date

8/14/2016

Bug No known bugs.

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

3.6 masterTestHolder.cpp File Reference

masterTestHolder singleton implementations

3.6.1 Detailed Description

masterTestHolder singleton implementations
Jonathan Bedard

Date

8/14/2016

Bug No known bugs.

This file contains implementations for the **test::masterTestHolder** (p. 28) singleton class. Consult **masterTestHolder.h** (p. 8) for details.

3.7 masterTestHolder.h File Reference

masterTestHolder singleton

Classes

- class **test::masterTestHolder**
  Unit Test singleton.

Namespaces

- **test**

3.7.1 Detailed Description

masterTestHolder singleton
Jonathan Bedard

Date

8/14/2016

Bug No known bugs.

This file contains declarations for the **test::masterTestHolder** (p. 28) singleton class. This file represents the top level of the Unit Test driver classes.

3.8 singleFunctionTest.cpp File Reference

3.9 singleFunctionTest.h File Reference

Classes

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

Namespaces

- **test**

Typedefs

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

3.10 singleTest.cpp File Reference

Single test class implementation.

3.10.1 Detailed Description

Single test class implementation.
Jonathan Bedard

Date

8/14/2016

Bug No known bugs.

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

3.11 singleTest.h File Reference

Single test class.

Classes

- class **test::singleTest**

  Single unit test class.

Namespaces

- **test**

3.11.1 Detailed Description

Single test class.
Jonathan Bedard

Date

8/13/2016

Bug No known bugs.

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

3.12 testSuite.cpp File Reference

Single test class.

3.12.1 Detailed Description

Single test class.
Jonathan Bedard

Date

8/14/2016

Bug No known bugs.

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

3.13 testSuite.h File Reference

Single test class.

Classes

- class **test::testSuite**

  Test suite.

Namespaces

- **test**

3.13.1 Detailed Description

Single test class.
Jonathan Bedard

Date

8/13/2016

Bug No known bugs.

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

3.14 UnitTest.cpp File Reference

Unit Test logging and global functions.

3.14.1 Detailed Description

Unit Test logging and global functions.

Author

Jonathan Bedard

Date

8/14/2016

Bug No known bugs.

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

3.15 UnitTest.h File Reference

Unit Test header file.

Namespaces

- **test**

Functions

- void **test::startTests** ()

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

  End tests in error.
- void **test::endTestsSuccess** ()

  End tests successfully.
- void **test::testInit** (int argc=0, char **argv=NULL)

  Test initialization.

3.15.1 Detailed Description

Unit Test header file.

Author

Jonathan Bedard

Date

8/13/2016

Bug No known bugs.

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

3.16 unitTestLog.h File Reference

Logging for test namespace.

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.

3.16.1 Detailed Description

Logging for test namespace.
Jonathan Bedard

Date

8/12/2016

Bug No known bugs.

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

3.17 UnitTestMain.cpp File Reference

UnitTest entry point.

Functions

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

3.17.1 Detailed Description

UnitTest entry point.

Author

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

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

3.17.2 Function Documentation

main()

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

Chapter 4

Class Index

4.1 Class List

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

Chapter 5

Namespace Documentation

5.1 test Namespace Reference

Classes

- class **generalTestException**

    Base class for test exceptions.
- class **libraryTests**

    Library test group.
- class **masterTestHolder**

    Unit Test singleton.
- class **nullFunctionException**

    NULL function exception class.
- class **singleFunctionTest**

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

    Single unit test class.
- class **testSuite**

    Test suite.
- class **unknownException**

    Unknown exception class.

Typedefs

- typedef void($*$ **testFunction**) ()

    Typedef for single test function.

Functions

- void **startTests** ()

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

End tests in error.

- void **endTestsSuccess** ()

  End tests successfully.
- void **testInit** (int argc=0, char **argv=NULL)

  Test initialization.
- std::ostream & **testout_func** ()

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

  Standard error object for test namespace.
- void **throwGeneralTestException** (const std::string &description, const std::string &location)

  Creates and throws a test exception.
- void **throwNullFunctionException** (const std::string &location)

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

  Creates and throws an unknown exception.

Variables

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

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

  Standard error pointer for test namespace.

5.1.1 Typedef Documentation

testFunction

```
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. 34).

Returns

void

5.1.2 Function Documentation

endTestsError()

```
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

endTestsSuccess()

```
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. 28) then another global division block

Returns

void

startTests()

```
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

testerr_func()

```
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."

testInit()

```
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

testout_func()

```
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."

throwGeneralTestException()

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

Creates and throws a test exception.

Parameters

in	*description*	Error description
in	*location*	Source of the error, file and function

Returns

void

throwNullFunctionException()

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

Creates and throws a null function exception.

Parameters

in	*location*	Source of the error, file and function

Returns

void

throwUnknownException()

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

Creates and throws an unknown exception.

Parameters

in	*location*	Source of the error, file and function

Returns

void

5.1.3 Variable Documentation

testerr_ptr

`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.

testout_ptr

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

Standard out pointer for test namespace.

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

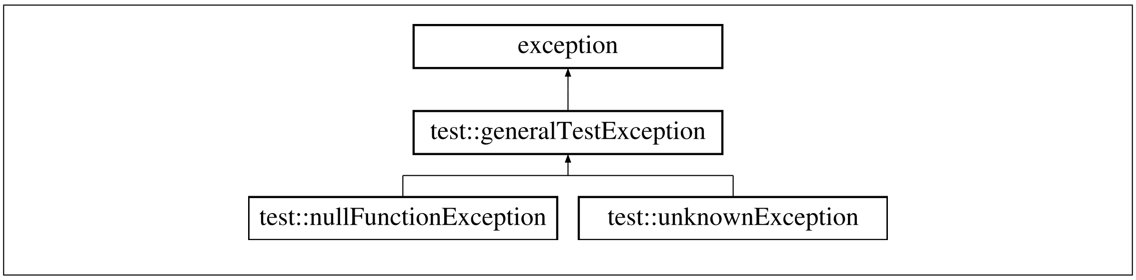
Chapter 6

Class Documentation

6.1 test::generalTestException Class Reference

Base class for test exceptions.

Inheritance diagram for test::generalTestException:



Public Member Functions

- **generalTestException** (const std::string &err, const std::string &loc)

  Construct exception with error and location.
- virtual ~**generalTestException** () throw ()

  Virtual destructor.
- const char ∗ **what** () const final throw ()

  std::exception overload
- const std::string & **getLocation** () const

  Location description.
- const std::string & **getString** () const

  Error description.

Static Public Member Functions

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

  Creates and throws a test exception.

Private Attributes

- std::string **location**

  The location where the error came from.
- std::string **_error**

  A description of the error.
- std::string **total_error**

  Combination of the error and location.

6.1.1 Detailed Description

Base class for test exceptions.

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

6.1.2 Constructor & Destructor Documentation

generalTestException()

```
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. 23) string for use by the "what()" function.

Parameters

in	*err*	Error string
in	*loc*	Location string

~generalTestException()

```
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

getLocation()

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

Location description.

Returns

test::generalTestException::location (p. 22)

getString()

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

Error description.

Returns

test::generalTestException::_error (p. 22)

throwException()

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

Creates and throws a test exception.

Parameters

in	*description*	Error description
in	*location*	Source of the error, file and function

Returns

void

what()

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

std::exception overload

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

Returns

character pointer to the complete description

6.1.4 Member Data Documentation

_error

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

A description of the error.

location

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

The location where the error came from.

total_error

`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. 22) and **test::generalTestException::location** (p. 22).

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.
- size_t **getNumSuites** () const

    Number of suites in the set.
- size_t **getNumSuccess** () const

    Number of suites successfully completed.
- size_t **getNumRun** () const

    Number of suites attempted to run.
- void **pushSuite** (os::smart_ptr< **testSuite** > suite)

    Add suite to the set.
- void **removeSuite** (os::smart_ptr< **testSuite** > suite)

    Remove suite from the set.
- int **compare** (const **libraryTests** &cmp) const

Compares two library test suites.

- **operator size_t** () const

  size_t cast for the library

Private Attributes

- std::string **libName**

  Name of library to be tested.
- os::pointerAVLTree< **testSuite** > **suiteList**

  Set of test suites.
- size_t **suitesCompleted**

  Number of suites successfully completed.
- size_t **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

libraryTests()

```
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	*ln*	Name of library to be tested

~libraryTests()

```
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

compare()

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

Compares two library test suites.

Parameters

in	*cmp*	Element to compare

Returns

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

getNumRun()

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

Number of suites attempted to run.

Returns

test::libraryTests::suitesRun (p. 28)

getNumSuccess()

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

Number of suites successfully completed.

Returns

test::libraryTests::suitesCompleted (p. 27)

getNumSuites()

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

Number of suites in the set.

Returns

test::libraryTests::suiteList.size()

logBegin()

```
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

logEnd()

```
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

onSetup()

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

Runs on shutdown of the group.

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

Returns

void

onTeardown()

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

Runs on teardown of the group.

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

Returns

void

operator size_t()

```
test::libraryTests::operator size_t ( ) const  [inline]
```

size_t cast for the library

Returns

size_t hash of the library

pushSuite()

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

Add suite to the set.

Adds a **test::testSuite** (p. 37) to the set of suites to be tested.

Parameters

in	*suite*	Test suite to be added to set

Returns

void

removeSuite()

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

Remove suite from the set.

Removes a **test::testSuite** (p. 37) from the set of suites to be tested.

Parameters

in	*suite*	Test suite to be removed from the set

Returns

void

runTests()

```
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

libName

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

Name of library to be tested.

suiteList

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

Set of test suites.

suitesCompleted

`size_t test::libraryTests::suitesCompleted [private]`

Number of suites successfully completed.

suitesRun

`size_t 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.
- size_t **getNumLibs** () const

  Number of libraries in the set.
- size_t **getNumSuccess** () const

  Number of libraries successfully completed.
- size_t **getNumRun** () const

  Number of libraries attempted to run.
- void **pushLibrary** (os::smart_ptr< **libraryTests** > lib)

  Add library to the set.
- void **removeLibrary** (os::smart_ptr< **libraryTests** > lib)

  Remove library from the set.

Static Public Member Functions

- static **masterTestHolder** & **singleton** ()

  Singleton access.

Private Member Functions

- **masterTestHolder** ()

  Private constructor.

Private Attributes

- os::pointerAVLTree< **libraryTests** > **libraryList**

  Set of library tests.
- size_t **libsCompleted**

  Number of libraries successfully completed.
- size_t **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. 30) function runs all of the library tests.

6.3.2 Constructor & Destructor Documentation

masterTestHolder()

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

Private constructor.

The **test::masterTestHolder** (p. 28) class is a singleton class. This constructor initializes the number of libraries completed and number of libraries run to 0.

~masterTestHolder()

`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

getNumLibs()

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

Number of libraries in the set.

Returns

test::masterTestHolder::libraryList.size()

getNumRun()

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

Number of libraries attempted to run.

Returns

test::masterTestHolder::libsRun (p. 31)

getNumSuccess()

`size_t test::masterTestHolder::getNumSuccess ( ) const [inline]`

Number of libraries successfully completed.

Returns

test::masterTestHolder::libsCompleted (p. 31)

pushLibrary()

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

Add library to the set.

Adds a **test::libraryTests** (p. 23) to the set of library tests to be tested.

Parameters

in	*lib*	Library test to be added to set

Returns

void

removeLibrary()

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

Remove library from the set.

Removes a **test::libraryTests** (p. 23) from the set of library tests to be tested.

Parameters

in	*lib*	Library test to be removed from the set

Returns

void

runTests()

`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

singleton()

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

Singleton access.

This function constructs the single reference to the **test::masterTestHolder** (p. 28) class if needed. Then, it returns a pointer to this single reference.

Returns

Singleton reference to **test::masterTestHolder** (p. 28)

6.3.4 Member Data Documentation

libraryList

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

Set of library tests.

libsCompleted

`size_t test::masterTestHolder::libsCompleted [private]`

Number of libraries successfully completed.

libsRun

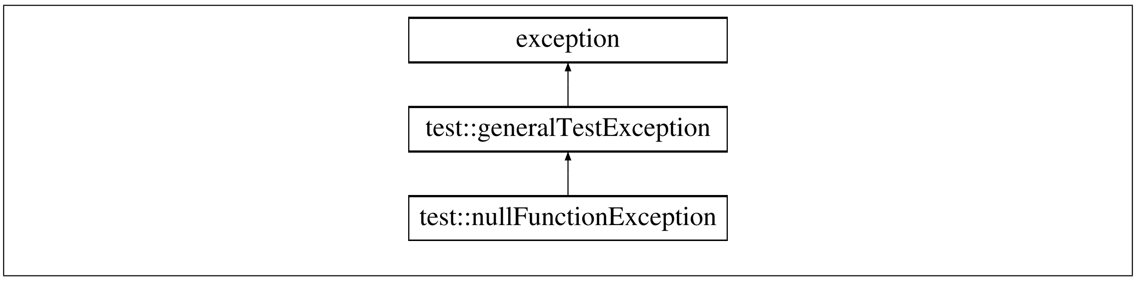
`size_t test::masterTestHolder::libsRun [private]`

Number of libraries attempted to run.

6.4 test::nullFunctionException Class Reference

NULL function exception class.

Inheritance diagram for test::nullFunctionException:



Public Member Functions

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

Static Public Member Functions

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

6.4.1 Detailed Description

NULL function exception class.

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

6.4.2 Constructor & Destructor Documentation

nullFunctionException()

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

Construct exception with location.

Constructs a **test::generalTestException** (p. 20) with the provided location and the static string for a NULL function exception.

Parameters

in	*loc*	Location string

~nullFunctionException()

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

Final destructor.

This class cannot be inherited from, so the the destructor cannot be extended.

6.4.3 Member Function Documentation

throwException()

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

Creates and throws a null function exception.

Parameters

in	*location*	Source of the error, file and function

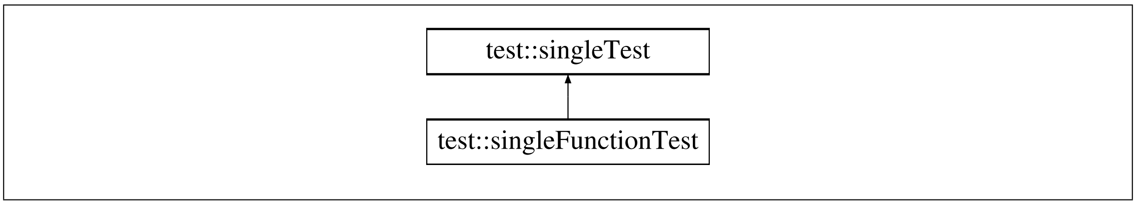
Returns

void

6.5 test::singleFunctionTest Class Reference

Single unit test from function.

Inheritance diagram for test::singleFunctionTest:



Public Member Functions

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

  Single unit test constructor.
- ~**singleFunctionTest** () final

  Final destructor.
- void **test** () final throw (os::errorPointer)

  Call unit test function.

Private Attributes

- **testFunction func**

  Reference to unit test function.

6.5.1 Detailed Description

Single unit test from function.

This class allows a **test::singleTest** (p. 34) to be defined by a single test function.

6.5.2 Constructor & Destructor Documentation

singleFunctionTest()

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

Single unit test constructor.

Parameters

in	*tn*	Name of unit test
in	*f*	Function which defines test

~singleFunctionTest()

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

Final destructor.

This class cannot be inherited from, so the the destructor cannot be extended.

6.5.3 Member Function Documentation

test()

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

Call unit test function.

Calls the function bound to this class in the constructor pointed to by **test::singleFunctionTest↩ ::func** (p. 34). 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. 36).

6.5.4 Member Data Documentation

func

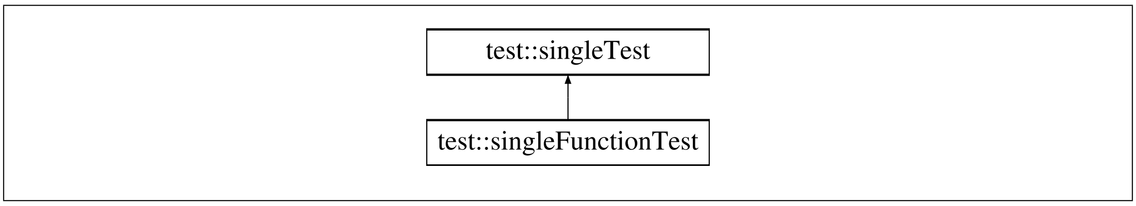
testFunction `test::singleFunctionTest::func [private]`

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

singleTest()

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

Single unit test constructor.

Parameters

in	*tn*	Name of unit test

~singleTest()

```
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

logBegin()

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

Prints out the name of the test.

Returns

void

logEnd()

```
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`	*except*	Exception to be printed, NULL by default

Returns

True if except is NULL

setupTest()

```
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. 37) will catch exceptions in this function if they are thrown.

Returns

void

teardownTest()

```
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. 37) will catch exceptions in this function if they are thrown.

Returns

void

test()

```
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. 37) will catch exceptions in this function if they are thrown.

Returns

void

Reimplemented in **test::singleFunctionTest** (p. 34).

6.6.4 Member Data Documentation

testName

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

Name of unit test.

6.7 test::testSuite Class Reference

Test suite.

Public Member Functions

- **testSuite** (std::string sn)

  Test suite constructor.
- virtual ~**testSuite** ()

  Virtual destructor.
- void **runTests** () throw (os::errorPointer)

  Runs all of the tests.
- virtual void **onSetup** ()

  Runs on shutdown.
- virtual void **onTeardown** ()

  Runs on teardown of the suite.
- void **logBegin** ()

  Logs the beginning of a suite test.
- bool **logEnd** (os::errorPointer except=NULL)

  Logs the end of a suite test.
- size_t **getNumTests** () const

  Number of tests in the set.
- size_t **getNumSuccess** () const

  Number of tests successfully completed.
- size_t **getNumRun** () const

  Number of tests attempted to run.
- void **pushTest** (os::smart_ptr< **singleTest** > tst)

  Add test to the set.
- void **removeTest** (os::smart_ptr< **singleTest** > tst)

  Remove test to the set.
- virtual void **pushTest** (std::string str, **testFunction** tst)

  Add test to the set.

- int **compare** (const **testSuite** &cmp) const

  Compares two library test suites.
- **operator size_t** () const

  size_t cast for the library

Private Attributes

- std::string **suiteName**

  Name of test suite.
- os::pointerUnsortedList< **singleTest** > **testList**

  Set of tests.
- size_t **testsCompleted**

  Number of tests successfully completed.
- size_t **testsRun**

  Number of tests attempted to run.

6.7.1 Detailed Description

Test suite.

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

6.7.2 Constructor & Destructor Documentation

testSuite()

```
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

~testSuite()

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

Virtual destructor.

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

6.7.3 Member Function Documentation

compare()

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

Compares two library test suites.

Parameters

in	*cmp*	Element to compare

Returns

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

getNumRun()

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

Number of tests attempted to run.

Returns

test::testSuite::testsRun (p. 42)

getNumSuccess()

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

Number of tests successfully completed.

Returns

test::testSuite::testsCompleted (p. 42)

getNumTests()

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

Number of tests in the set.

Returns

test::testSuite::testList.size()

logBegin()

```
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

logEnd()

```
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

onSetup()

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

Runs on shutdown.

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

Returns

void

onTeardown()

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

Runs on teardown of the suite.

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

Returns

void

operator size_t()

```
test::testSuite::operator size_t ( ) const  [inline]
```

size_t cast for the library

Returns

size_t hash of the library

pushTest() [1/2]

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

Add test to the set.

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

Parameters

in	*tst*	Test to be added to set

Returns

void

pushTest() [2/2]

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

Add test to the set.

Adds a **test::testFunction** (p. 16) to the set of tests to be tested. Constructs a **test::singleTest** (p. 34) from a function and a test name

Parameters

in	*str*	Test name
in	*tst*	Function which defines test

Returns

void

removeTest()

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

Remove test to the set.

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

Parameters

in	*tst*	Test to be removed from the set

Returns

void

runTests()

```
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. 34) and logs the results.

Returns

void

6.7.4 Member Data Documentation

suiteName

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

Name of test suite.

testList

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

Set of tests.

testsCompleted

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

Number of tests successfully completed.

testsRun

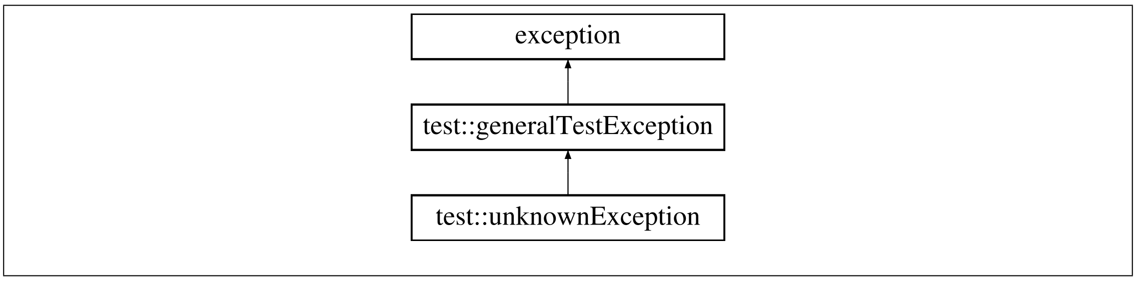
`size_t test::testSuite::testsRun [private]`

Number of tests attempted to run.

6.8 test::unknownException Class Reference

Unknown exception class.

Inheritance diagram for test::unknownException:



Public Member Functions

- **unknownException** (const std::string &loc)

  Construct exception with location.
- ~**unknownException** () final throw ()

  Final destructor.

Static Public Member Functions

- static void **throwException** (const std::string &**location**)

  Creates and throws an unknown exception.

6.8.1 Detailed Description

Unknown exception class.

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

6.8.2 Constructor & Destructor Documentation

unknownException()

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

Construct exception with location.

Constructs a **test::generalTestException** (p. 20) with the provided location and the static string for an unknown exception.

Parameters

in	*loc*	Location string

~unknownException()

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

Final destructor.

This class cannot be inherited from, so the the destructor cannot be extended.

6.8.3 Member Function Documentation

throwException()

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

Creates and throws an unknown exception.

Parameters

in	*location*	Source of the error, file and function

Returns

void

Part II

Datastructures Library

Chapter 7

Introduction

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

7.1 Unit Testing

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

7.2 Namespace os

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

Chapter 8

File Index

8.1 File List

Here is a list of all files with brief descriptions:

Chapter 9

File Documentation

9.1 abstractSorting.h File Reference

Template for sorting arrays.

Namespaces

- **os**

Functions

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

  Basic compare.
- template<class dataType >
  int **os::pointerCompare** (const smart_ptr< dataType > &ptr1, const smart_ptr< dataType > &ptr2) throw ()

  Pointer compare.
- template<class dataType >
  void **os::quicksort** (dataType ∗arr, size_t length, int(∗sort_comparison)(const dataType &, const dataType &)=&defaultCompare)

  Template quick-sort.
- template<class dataType >
  void **os::pointerQuicksort** (smart_ptr< smart_ptr< dataType > > arr, size_t length, int(∗sort↩
  _comparison)(const smart_ptr< dataType > &, const smart_ptr< dataType > &)=&pointer↩
  Compare)

  Template for quick-sort, pointer version.

9.1.1 Detailed Description

Template for sorting arrays.

Author

Jonathan Bedard

Date

7/12/2016

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 thread safe.

9.2 adsFrame.h File Reference

Include headers with macro management.

9.2.1 Detailed Description

Include headers with macro management.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

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

9.3 arrayStack.h File Reference

Defines a stack with an array.

Classes

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

Namespaces

- **os**

9.3.1 Detailed Description

Defines a stack with an array.

Author

Jonathan Bedard

Date

7/17/2016

Bug No known bugs.

This header should only be included from **specializedStructures.h** (p. 96). It defines a number of different types of stacks to be used.

9.4 basicLock.cpp File Reference

Implementation basicLock.

9.4.1 Detailed Description

Implementation basicLock.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Implements the non-recursive lock defined in **basicLock.h** (p. 51).

9.5 basicLock.h File Reference

Non-recursive lock.

Classes

- class **os::basicLock**

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

Namespaces

- **os**

9.5.1 Detailed Description

Non-recursive lock.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Defines a simple, non-recursive, mutex.

9.6 basicStructures.h File Reference

Include structure headers with macro management.

9.6.1 Detailed Description

Include structure headers with macro management.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

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

9.7 constantPrinter.cpp File Reference

Constant printing support, implementation.

9.7.1 Detailed Description

Constant printing support, implementation.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

This file implements **os::constantPrinter** (p. 163). Consult staticConstantPrinter.h for detailed documentation.

9.8 constantPrinter.h File Reference

Constant printing support.

Classes

- class **os::constantPrinter**

    Prints constant arrays to files.

Namespaces

- **os**

9.8.1 Detailed Description

Constant printing support.

Author

Jonathan Bedard

Date

8/25/2016

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.

9.9 Datastructures.h File Reference

Master Datastructures header file.

9.9.1 Detailed Description

Master Datastructures header file.

Author

Jonathan Bedard

Date

8/28/2016

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.

9.10 descriptiveException.h File Reference

Basic descriptive exception.

Classes

- class **os::descriptiveException**

  Basic exception with description.

Namespaces

- **os**

9.10.1 Detailed Description

Basic descriptive exception.

Author

Jonathan Bedard

Date

6/3/2016

Bug No known bugs.

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

9.11 eventDriver.h File Reference

Event sender and receiver.

Classes

- class **os::eventSenderBase**

  Base class for sender events This class is inherited by the generalized sender event class.
- class **os::eventReceiverBase**

  Base class for receiving events This class is inherited by the generalized receiver class.
- class **os::eventSender**< **receiverType** >

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

  Class which enables event receiving.

Namespaces

- **os**

9.11.1 Detailed Description

Event sender and receiver.

Author

Jonathan Bedard

Date

8/19/2016

Bug No known bugs.

Both **os::eventReceiver** (p. 189) and **os::eventSender** (p. 193) are experimental classes and have not been tested or utilized.

9.12 iterator.h File Reference

Defines an iterator.

Classes

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

Namespaces

- **os**

9.12.1 Detailed Description

Defines an iterator.

Author

Jonathan Bedard

Date

8/25/2016

Bug No known bugs.

Defines an iterator which wraps a node for intuitive usage

9.13 linkedQueue.h File Reference

Defines a queue with a linked list.

Classes

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

Namespaces

- **os**

9.13.1 Detailed Description

Defines a queue with a linked list.

Author

Jonathan Bedard

Date

8/6/2016

Bug No known bugs.

This header should only be included from **specializedStructures.h** (p. 96). It defines a number of different types of queues to be used.

9.14 linkedStack.h File Reference

Defines a stack with a linked list.

Classes

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

Namespaces

- **os**

9.14.1 Detailed Description

Defines a stack with a linked list.

Author

Jonathan Bedard

Date

8/6/2016

Bug No known bugs.

This header should only be included from **specializedStructures.h** (p. 96). It defines a number of different types of stacks to be used.

9.15 lockable.h File Reference

Defines the lockable interface.

Classes

- class **os::lockable**

  An interface defining a class which can be locked.

Namespaces

- **os**

9.15.1 Detailed Description

Defines the lockable interface.

Author

Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

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

9.16 Locks.h File Reference

Constant locking support.

9.16.1 Detailed Description

Constant locking support.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Includes the headers for all locking structures.

9.17 macroAVL.h File Reference

Defines a generalized AVL tree.

Classes

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

  AVL tree node Node used in an AVL tree. This node allows both itteration and random access.
- class **os::AVL_TREE**< **dataType** >

  Template AVL Tree definition.

Namespaces

- **os**

9.17.1 Detailed Description

Defines a generalized AVL tree.

Author

Jonathan Bedard

Date

8/25/2016

Bug No known bugs.

This header should only be included from **basicStructures.h** (p. 52). It defines a number of different types of AVL trees to be used.

9.18 macroBoundedQueue.h File Reference

Defines a bounded queue.

Classes

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

  *Template **BOUNDED_QUEUE** (p. 147) definition.*
- class **os::BOUNDED_QUEUE_NODE**< **dataType** >

  ***BOUNDED_QUEUE** (p. 147) node.*

Namespaces

- **os**

9.18.1 Detailed Description

Defines a bounded queue.

Author

Jonathan Bedard

Date

7/17/2016

Bug No known bugs.

This header should only be included from **specializedStructures.h** (p. 96). It defines a number of different types of bounded queues to be used.

9.19 macroDatastructuresInterface.h File Reference

Defines the interface used by datastructures.

Classes

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

  Object node definition.

Namespaces

- **os**

9.19.1 Detailed Description

Defines the interface used by datastructures.

Author

Jonathan Bedard

Date

7/12/2016

Bug No known bugs.

This header should only be included from the **adsFrame.h** (p. 50). It defines a number of different forms of the generalized data-structure template

9.20 macroHash.h File Reference

Defines an iterable hash table.

Classes

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

Namespaces

- **os**

9.20.1 Detailed Description

Defines an iterable hash table.

Author

Jonathan Bedard

Date

8/4/2016

Bug No known bugs.

This header should only be included from **specializedStructures.h** (p. 96). It defines a number of different types of hashes to be used.

9.21 macroList.h File Reference

Defines a basic linked list.

Classes

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

  List node Node used in linked lists to store each element.
- class **os::LIST**< **dataType** >

  List framework Template for a linked list, insertion is fully defined in subsequent extensions of this class.
- class **os::UNSORTED_LIST**< **dataType** >

  Unsorted list A basic list which remains unsorted unless the sort function is called on it.
- class **os::SORTED_LIST**< **dataType** >

  Sorted list A basic list which remains unsorted unless the sort function is called on it.

Namespaces

- **os**

9.21.1 Detailed Description

Defines a basic linked list.

Author

Jonathan Bedard

Date

8/25/2016

Bug No known bugs.

This header should only be included from **basicStructures.h** (p. 52). It defines a number of different types of vectors to be used.

9.22 macros.h File Reference

Support macro declarations.

9.22.1 Detailed Description

Support macro declarations.

Author

Jonathan Bedard

Date

7/24/2016

Bug No known bugs.

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

9.23 macroSet.h File Reference

Defines an iterable set.

Classes

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

  Template AVL Tree definition.

Namespaces

- **os**

9.23.1 Detailed Description

Defines an iterable set.

Author

Jonathan Bedard

Date

8/28/2016

Bug No known bugs.

This header should only be included from **specializedStructures.h** (p. 96). It defines a number of different types of sets to be used, which use other datastructures to define a set.

9.24 macroVector.h File Reference

Defines an expandable array.

Classes

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

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

  Vector node.

Namespaces

- **os**

9.24.1 Detailed Description

Defines an expandable array.

Author

Jonathan Bedard

Date

8/6/2016

Bug No known bugs.

This header should only be included from **basicStructures.h** (p. 52). It defines a number of different types of vectors to be used.

9.25 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**< dataType > &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::indirectMatrix**< 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::indirectMatrix**< 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.

9.25.1 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.

9.25.2 Function Documentation

operator"!=() [1/4]

```
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	*m1*	Raw matrix reference
in	*m2*	Raw matrix reference

Returns

False if exactly equivalent

operator"!=() [2/4]

```
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	*m1*	Indirect matrix reference
in	*m2*	Raw matrix reference

Returns

False if exactly equivalent

operator"!=() [3/4]

```
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	*m1*	Raw matrix reference
in	*m2*	Indirect matrix reference

Returns

False if exactly equivalent

operator"!=() [4/4]

```
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	*m1*	Indirect matrix reference
in	*m2*	Indirect matrix reference

Returns

False if exactly equivalent

operator*() [1/8]

```
template<class dataType >
os::matrix<dataType> operator* (
            const os::matrix< 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	*m1*	Raw matrix reference
in	*m2*	Raw matrix reference

Returns

m1 x m2 (raw matrix)

operator*() [2/8]

```
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	*m1*	Indirect matrix reference
in	*m2*	Raw matrix reference

Returns

m1 x m2 (raw matrix)

operator*() [3/8]

```
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	*m1*	Raw matrix reference
in	*m2*	Indirect matrix reference

Returns

m1 x m2 (raw matrix)

operator*() [4/8]

```
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	*m1*	Indirect matrix reference
in	*m2*	Indirect matrix reference

Returns

m1 x m2 (indirect matrix)

operator*() [5/8]

```
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	*d1*	Scalar data type
in	*m1*	Raw matrix reference

Returns

d1 * m1 (raw matrix)

operator*() [6/8]

```
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	*m1*	Raw matrix reference
in	*d1*	Scalar data type

Returns

d1 * m1 (raw matrix)

operator*() [7/8]

```
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 data↩ Type.

Parameters

in	*d1*	Scalar data type
in	*m1*	Indirect matrix reference

Returns

d1 * m1 (indirect matrix)

operator*() [8/8]

```
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	*m1*	Indirect matrix reference
in	*d1*	Scalar data type

Returns

d1 $*$ m1 (indirect matrix)

operator+() [1/4]

```
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	*m1*	Raw matrix reference
in	*m2*	Raw matrix reference

Returns

m1 + m2 (raw matrix)

operator+() [2/4]

```
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	*m1*	Indirect matrix reference
in	*m2*	Raw matrix reference

Returns

m1 + m2 (raw matrix)

operator+() [3/4]

```
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	*m1*	Raw matrix reference
in	*m2*	Indirect matrix reference

Returns

m1 + m2 (raw matrix)

operator+() [4/4]

```
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	*m1*	Indirect matrix reference
in	*m2*	Indirect matrix reference

Returns

m1 + m2 (indirect matrix)

operator-() [1/4]

```
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	*m1*	Raw matrix reference
in	*m2*	Raw matrix reference

Returns

m1 - m2 (raw matrix)

operator-() [2/4]

```
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	*m1*	Indirect matrix reference
in	*m2*	Raw matrix reference

Returns

m1 - m2 (raw matrix)

operator-() [3/4]

```
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	*m1*	Raw matrix reference
in	*m2*	Indirect matrix reference

Returns

m1 - m2 (raw matrix)

operator-() [4/4]

```
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	*m1*	Indirect matrix reference
in	*m2*	Indirect matrix reference

Returns

m1 - m2 (indirect matrix)

operator/() [1/2]

```
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	*m1*	Raw matrix reference
in	*d1*	Scalar data type

Returns

m1/d (raw matrix)

operator/() [2/2]

```
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	*m1*	Raw matrix reference
in	*d1*	Scalar data type

Returns

m1/d (raw matrix)

operator<<() [1/2]

```
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

	[in/out]	os std::ostream reference
in	*dt*	Raw matrix reference

Returns

std::ostream os

operator<<() [2/2]

```
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

	[in/out]	os std::ostream reference
in	*dt*	Indirect matrix reference

Returns

std::ostream os

operator==() [1/4]

```
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	*m1*	Raw matrix reference
in	*m2*	Raw matrix reference

Returns

True if exactly equivalent

operator==() [2/4]

```
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	*m1*	Indirect matrix reference
in	*m2*	Raw matrix reference

Returns

True if exactly equivalent

operator==() [3/4]

```
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	*m1*	Raw matrix reference
in	*m2*	Indirect matrix reference

Returns

True if exactly equivalent

operator==() [4/4]

```
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	*m1*	Indirect matrix reference
in	*m2*	Indirect matrix reference

Returns

True if exactly equivalent

9.26 nodeFrame.h File Reference

Declares a framework for nodes.

Classes

- class **os::nodeFrame**< **dataType** >
  nodeFrame *(p. 274) interface*

Namespaces

- **os**

9.26.1 Detailed Description

Declares a framework for nodes.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

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

9.27 nodeMacroHeader.h File Reference

Node macro header.

Classes

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

    Object node definition.

Namespaces

- **os**

9.27.1 Detailed Description

Node macro header.

Author

Jonathan Bedard

Date

8/24/2016

Bug No known bugs.

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

9.28 osLogger.cpp File Reference

Logging for os namespace, implementation.

9.28.1 Detailed Description

Logging for os namespace, implementation.
Jonathan Bedard

Date

7/9/2016

Bug No known bugs.

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

9.29 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.

9.29.1 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.

9.30 readWriteInterface.h File Reference

Classes

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

Namespaces

- **os**

9.31 readWriteLock.cpp File Reference

Implements the read/write lock.

9.31.1 Detailed Description

Implements the read/write lock.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Implements the read/write lock as defined in **readWriteLock.h** (p. 81).

9.32 readWriteLock.h File Reference

Declaration of the read/write lock.

Classes

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

Namespaces

- **os**

9.32.1 Detailed Description

Declaration of the read/write lock.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

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

9.33 simpleHash.cpp File Reference

Implements a basic hash function.

9.33.1 Detailed Description

Implements a basic hash function.

Author

Jonathan Bedard

Date

8/12/4/2016

Bug No known bugs.

Implements the function used to hash a set of data.

9.34 simpleHash.h File Reference

Simple hash table template class.

Classes

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

Namespaces

- **os**

Functions

- size_t **os::hashData** (const unsigned char ∗dat, size_t len)
- template<class dataType >
  size_t **os::hashObject** (const dataType &dt)

  Default hash function This hash function assumes that an object's hash is determined by it's raw data.

9.34.1 Detailed Description

Simple hash table template class.

Author

Jonathan Bedard

Date

8/12/2016

Bug No known bugs.

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

9.35 smartPointer.h File Reference

Template declaration of **os::smart_ptr** (p. 311).

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. 311)**.*

Functions

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

  os::smart_ptr *(p. 311) cast function*
- template<class dataType >
  bool **operator==** (const **os::smart_ptr**< dataType > &c1, const **os::smart_ptr**< dataType > &c2) throw ()
- template<class dataType >
  bool **operator==** (const **os::smart_ptr**< dataType > &c1, const dataType ∗c2) throw ()
- template<class dataType >
  bool **operator==** (const dataType ∗c1, const **os::smart_ptr**< dataType > &c2) throw ()
- template<class dataType >
  bool **operator==** (const **os::smart_ptr**< dataType > &c1, const void ∗c2) throw ()
- template<class dataType >
  bool **operator==** (const void ∗c1, const **os::smart_ptr**< dataType > &c2) throw ()
- template<class dataType >
  bool **operator==** (const **os::smart_ptr**< dataType > &c1, const int c2) throw ()
- template<class dataType >
  bool **operator==** (const int c1, const **os::smart_ptr**< dataType > &c2) throw ()
- template<class dataType >
  bool **operator==** (const **os::smart_ptr**< dataType > &c1, const long c2) throw ()
- template<class dataType >
  bool **operator==** (const long c1, const **os::smart_ptr**< dataType > &c2) throw ()
- 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 ()
- 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 ()
- 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 ()
- 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 ()
- 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 ()

9.35.1 Detailed Description

Template declaration of **os::smart_ptr** (p. 311).

Author

Jonathan Bedard

Date

7/12/2016

Bug No known bugs.

This file contains a template declaration of **os::smart_ptr** (p. 311) and supporting constants and functions. Note that because **os::smart_ptr** (p. 311) is a template class, the implimentation of **os::smart_ptr** (p. 311) occurs here as well.

9.35.2 Function Documentation

operator"!=() [1/11]

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

operator"!=() [2/11]

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

operator"!=() [3/11]

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

operator"!=() [4/11]

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

operator"!=() [5/11]

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

operator"!=() [6/11]

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

operator"!=() [7/11]

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

operator"!=() [8/11]

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

operator"!=() [9/11]

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

operator"!=() [10/11]

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

operator"!=() [11/11]

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

operator<() [1/11]

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

operator<() [2/11]

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

operator<() [3/11]

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

operator<() [4/11]

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

operator<() [5/11]

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

operator<() [6/11]

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

operator<() [7/11]

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

operator<() [8/11]

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

operator<() [9/11]

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

operator<() [10/11]

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

operator<() [11/11]

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

operator<=() [1/11]

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

operator<=() [2/11]

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

operator<=() [3/11]

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

operator<=() [4/11]

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

operator<=() [5/11]

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

operator<=() [6/11]

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

operator<=() [7/11]

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

operator<=() [8/11]

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

operator<=() [9/11]

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

operator<=() [10/11]

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

operator<=() [11/11]

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

operator==() [1/11]

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

operator==() [2/11]

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

operator==() [3/11]

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

operator==() [4/11]

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

operator==() [5/11]

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

operator==() [6/11]

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

operator==() [7/11]

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

operator==() [8/11]

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

operator==() [9/11]

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

operator==() [10/11]

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

operator==() [11/11]

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

operator>() [1/11]

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

operator>() [2/11]

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

operator>() [3/11]

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

operator>() [4/11]

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

operator>() [5/11]

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

operator>() [6/11]

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

operator>() [7/11]

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

operator>() [8/11]

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

operator>() [9/11]

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

operator>() [10/11]

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

operator>() [11/11]

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

operator>=() [1/11]

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

operator>=() [2/11]

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

operator>=() [3/11]

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

operator>=() [4/11]

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

operator>=() [5/11]

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

operator>=() [6/11]

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

operator>=() [7/11]

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

operator>=() [8/11]

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

operator>=() [9/11]

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

operator>=() [10/11]

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

operator>=() [11/11]

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

9.36 specializedStructures.h File Reference

Include specialized structure headers with macro management.

9.36.1 Detailed Description

Include specialized structure headers with macro management.

Author

Jonathan Bedard

Date

8/25/2016

Bug No known bugs.

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

9.37 threadCounter.cpp File Reference

Implementation of thread counter.

9.37.1 Detailed Description

Implementation of thread counter.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Implementation of the threadCounter class defined in **threadCounter.h** (p. 97).

9.38 threadCounter.h File Reference

Thread counter declaration.

Classes

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

Namespaces

- **os**

9.38.1 Detailed Description

Thread counter declaration.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

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

9.39 threadLock.cpp File Reference

Implementation of thread locks.

9.39.1 Detailed Description

Implementation of thread locks.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

Implements the recursive thread lock.

9.40 threadLock.h File Reference

Recursive lock.

Classes

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

Namespaces

- **os**

9.40.1 Detailed Description

Recursive lock.

Author

Jonathan Bedard

Date

7/8/2016

Bug No known bugs.

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

9.41 vector2d.h File Reference

Vector templates.

Classes

- class **os::vector2d**< **dataType** >

  2-dimensional vector

Namespaces

- **os**

Typedefs

- typedef vector2d< int8_t > **os::vector2d_8**

  8 bit 2-d vector
- typedef vector2d< uint8_t > **os::vector2d_u8**

  unsigned 8 bit 2-d vector
- typedef vector2d< int16_t > **os::vector2d_16**

  16 bit 2-d vector
- typedef vector2d< uint16_t > **os::vector2d_u16**

  unsigned 16 bit 2-d vector
- typedef vector2d< int32_t > **os::vector2d_32**

  32 bit 2-d vector
- typedef vector2d< uint32_t > **os::vector2d_u32**

  unsigned 32 bit 2-d vector
- typedef vector2d< int64_t > **os::vector2d_64**

  64 bit 2-d vector
- typedef vector2d< uint64_t > **os::vector2d_u64**

  unsigned 64 bit 2-d vector
- typedef vector2d< float > **os::vector2d_f**

  float 2-d vector
- typedef vector2d< double > **os::vector2d_d**

  double 2-d vector

9.41.1 Detailed Description

Vector templates.

Author

Jonathan Bedard

Date

8/31/2016

Bug No known bugs.

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

9.42 vector3d.h File Reference

Vector templates.

Classes

- class **os::vector3d**< **dataType** >

  3-dimensional vector

Namespaces

- **os**

Typedefs

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

  8 bit 3-d vector
- typedef vector3d< uint8_t > **os::vector3d_u8**

  unsigned 8 bit 3-d vector
- typedef vector3d< int16_t > **os::vector3d_16**

  16 bit 3-d vector
- typedef vector3d< uint16_t > **os::vector3d_u16**

  unsigned 16 bit 3-d vector
- typedef vector3d< int32_t > **os::vector3d_32**

  32 bit 3-d vector
- typedef vector3d< uint32_t > **os::vector3d_u32**

  unsigned 32 bit 3-d vector
- typedef vector3d< int64_t > **os::vector3d_64**

  64 bit 3-d vector

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

    unsigned 64 bit 3-d vector
- typedef vector3d< float > **os::vector3d_f**

    float 3-d vector
- typedef vector3d< double > **os::vector3d_d**

    double 3-d vector

9.42.1 Detailed Description

Vector templates.

Author

Jonathan Bedard

Date

8/31/2016

Bug No known bugs.

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

Chapter 10

Class Index

10.1 Class List

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

Chapter 11

Namespace Documentation

11.1 os Namespace Reference

Classes

- class **ARRAY_STACK**
- class **AVL_NODE**

  AVL tree node Node used in an AVL tree. This node allows both itteration and random access.
- class **AVL_TREE**

  Template AVL Tree definition.
- 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. *147)* *definition.*
- class **BOUNDED_QUEUE_NODE**

  BOUNDED_QUEUE (p. *147)* *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 **eventReceiverBase**

  Base class for receiving events This class is inherited by the generalized receiver class.
- class **eventSender**

Class which enables event sending.

- class **eventSenderBase**

  Base class for sender events This class is inherited by the generalized sender event class.
- class **HASH**

  Iterable hash table.
- class **HASH_NODE**

  Hash node.
- class **indirectMatrix**

  Indirect matrix.
- class **iterator**

  Generalized iterator.
- class **iteratorBase**

  Iterator source.
- class **iteratorBaseInterface**

  Iterator base interface.
- class **iteratorSource**

  Base class for iterator source.
- class **LINKED_QUEUE**

  Queue built on-top of a list.
- class **LINKED_STACK**

  Stack built on-top of a list.
- class **LIST**

  List framework Template for a linked list, insertion is fully defined in subsequent extensions of this class.
- class **LIST_NODE**

  List node Node used in linked lists to store each element.
- class **lockable**

  An interface defining a class which can be locked.
- class **matrix**

  Raw matrix.
- class **NODE**

  Object node definition.
- class **nodeFrame**

  ***nodeFrame** (p. 274) interface*
- class **readWriteInterface**

  Read/write interface.
- class **readWriteLock**

  Read/write lock.
- class **SET**

  Template AVL Tree definition.
- class **simpleHash**

  Basic hash function.
- class **smart_ptr**

Reference counted pointer.

- class **SORTED_LIST**

  Sorted list A basic list which remains unsorted unless the sort function is called on it.
- class **threadCounter**

  Thread counter.
- class **threadLock**

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

  Unsorted list A basic list which remains unsorted unless the sort function is called on it.
- class **VECTOR**

  Template vector definition.
- class **vector2d**

  2-dimensional vector
- class **vector3d**

  3-dimensional vector
- class **VECTOR_NODE**

  Vector node.

Typedefs

- typedef void(∗ **void_rec**) (void ∗)

  Deletion function typedef.
- typedef **vector2d**< int8_t > **vector2d_8**

  8 bit 2-d vector
- typedef **vector2d**< uint8_t > **vector2d_u8**

  unsigned 8 bit 2-d vector
- typedef **vector2d**< int16_t > **vector2d_16**

  16 bit 2-d vector
- typedef **vector2d**< uint16_t > **vector2d_u16**

  unsigned 16 bit 2-d vector
- typedef **vector2d**< int32_t > **vector2d_32**

  32 bit 2-d vector
- typedef **vector2d**< uint32_t > **vector2d_u32**

  unsigned 32 bit 2-d vector
- typedef **vector2d**< int64_t > **vector2d_64**

  64 bit 2-d vector
- typedef **vector2d**< uint64_t > **vector2d_u64**

  unsigned 64 bit 2-d vector
- typedef **vector2d**< float > **vector2d_f**

  float 2-d vector
- typedef **vector2d**< double > **vector2d_d**

  double 2-d vector

- typedef **vector3d**< int8_t > **vector3d_8**

  8 bit 3-d vector
- typedef **vector3d**< uint8_t > **vector3d_u8**

  unsigned 8 bit 3-d vector
- typedef **vector3d**< int16_t > **vector3d_16**

  16 bit 3-d vector
- typedef **vector3d**< uint16_t > **vector3d_u16**

  unsigned 16 bit 3-d vector
- typedef **vector3d**< int32_t > **vector3d_32**

  32 bit 3-d vector
- typedef **vector3d**< uint32_t > **vector3d_u32**

  unsigned 32 bit 3-d vector
- typedef **vector3d**< int64_t > **vector3d_64**

  64 bit 3-d vector
- typedef **vector3d**< uint64_t > **vector3d_u64**

  unsigned 64 bit 3-d vector
- typedef **vector3d**< float > **vector3d_f**

  float 3-d vector
- typedef **vector3d**< double > **vector3d_d**

  double 3-d vector

Enumerations

- enum **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. 311).*

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, size_t length, int(∗sort_comparison)(const dataType &, const dataType &)=&**defaultCompare**)

  Template quick-sort.
- template<class dataType >
  void **pointerQuicksort** (**smart_ptr**< **smart_ptr**< dataType > > arr, size_t 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.
- size_t **hashData** (const unsigned char ∗dat, size_t len)
- template<class dataType >
  size_t **hashObject** (const dataType &dt)

  Default hash function This hash function assumes that an object's hash is determined by it's raw data.
- template<class targ , class src >
  smart_ptr< targ > **cast** (const **os::smart_ptr**< src > &conv) throw ()

  os::smart_ptr *(*p. *311) 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

vector2d_16

`typedef` **vector2d**`<int16_t>` **os::vector2d_16**

16 bit 2-d vector

vector2d_32

`typedef` **vector2d**`<int32_t>` **os::vector2d_32**

32 bit 2-d vector

vector2d_64

`typedef` **vector2d**`<int64_t>` **os::vector2d_64**

64 bit 2-d vector

vector2d_8

`typedef` **vector2d**`<int8_t>` **os::vector2d_8**

8 bit 2-d vector

vector2d_d

`typedef` **vector2d**`<double>` **os::vector2d_d**

double 2-d vector

vector2d_f

`typedef` **vector2d**`<float>` **os::vector2d_f**

float 2-d vector

vector2d_u16

`typedef` **vector2d**`<uint16_t>` **os::vector2d_u16**

unsigned 16 bit 2-d vector

vector2d_u32

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

unsigned 32 bit 2-d vector

vector2d_u64

typedef **vector2d**<uint64_t> **os::vector2d_u64**
unsigned 64 bit 2-d vector

vector2d_u8

typedef **vector2d**<uint8_t> **os::vector2d_u8**
unsigned 8 bit 2-d vector

vector3d_16

typedef **vector3d**<int16_t> **os::vector3d_16**
16 bit 3-d vector

vector3d_32

typedef **vector3d**<int32_t> **os::vector3d_32**
32 bit 3-d vector

vector3d_64

typedef **vector3d**<int64_t> **os::vector3d_64**
64 bit 3-d vector

vector3d_8

typedef **vector3d**<int8_t> **os::vector3d_8**
8 bit 3-d vector

vector3d_d

typedef **vector3d**<double> **os::vector3d_d**
double 3-d vector

vector3d_f

typedef **vector3d**<float> **os::vector3d_f**
float 3-d vector

vector3d_u16

typedef **vector3d**<uint16_t> **os::vector3d_u16**
unsigned 16 bit 3-d vector

vector3d_u32

typedef **vector3d**<uint32_t> **os::vector3d_u32**
unsigned 32 bit 3-d vector

vector3d_u64

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

unsigned 64 bit 3-d vector

vector3d_u8

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

unsigned 8 bit 3-d vector

void_rec

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

Deletion function typedef.

The **os::void_rec** (p. 112) function pointer typedef is used by **os::smart_ptr** (p. 311) when it is of type **os::shared_type_dynamic_delete** (p. 113) 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

smart_pointer_type

enum **os::smart_pointer_type**

Enumeration for types of **os::smart_ptr** (p. 311).

Defines types of **os::smart_ptr** (p. 311). These types are used to define the deletion behaviour of the pointer.

Enumerator

null_type	No type. **os::null_type** (p. 112) pointers are the default type of **os::smart_ptr** (p. 311). Any **os::smart_ptr** (p. 311) of type **os::null_type** (p. 112) can be guaranteed to hold a NULL pointer.
raw_type	Raw pointer. **os::raw_type** (p. 112) pointers are the default type of **os::smart_ptr** (p. 311) when instantiated with a standard pointer. Any **os::smart_ptr** (p. 311) of type **os::raw_type** (p. 112) is not responsible for the deletion of it's pointer and makes no guarantees as to the availability of it's pointer.

Enumerator

shared_type	Reference counted pointer. **os::shared_type** (p. 113) pointers must be instantiated from an **os::smart_ptr** (p. 311) of this type or explicitly through **os::smart_ptr** (p. 311) constructor arguments. **os::shared_type** (p. 113) pointers will automatically delete the pointer contained within the object when the reference count of the **os::smart_ptr** (p. 311) reaches 0.
shared_type_array	Reference counted array. Similar in usage and instantiation to **os::raw_type** (p. 112). **os::smart_ptr** (p. 311) of type **os::shared_type_array** (p. 113) 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. 112). **os::smart_ptr** (p. 311) of type **os::shared_type_dynamic_delete** (p. 113) 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

cast()

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

os::smart_ptr (p. 311) 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. 311)

compareSize() [1/4]

```
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

compareSize() [2/4]

```
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	*m1*	Indirect matrix reference
in	*m2*	Raw matrix reference

Returns

True if the matrices are the same size

compareSize() [3/4]

```
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	*m1*	Raw matrix reference
in	*m2*	Indirect matrix reference

Returns

True if the matrices are the same size

compareSize() [4/4]

```
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	*m1*	Indirect matrix reference
in	*m2*	Indirect matrix reference

Returns

True if the matrices are the same size

defaultCompare()

```
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

hashData()

```
size_t os::hashData (
            const unsigned char * dat,
            size_t len )
```

Byte array hash function

Parameters

in	*dat*	Data to be hashed
in	*len*	Size of data to be hashed

Returns

Hash value of data

hashObject()

```
template<class dataType >
size_t os::hashObject (
        const dataType & dt )
```

Default hash function This hash function assumes that an object's hash is determined by it's raw data.

Returns

size_t representing an object's hash

oserr_func()

```
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."

osout_func()

```
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."

pointerCompare()

```
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 arguements 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

pointerQuicksort()

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

Template for quick-sort, pointer version.

Preforms 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

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

Returns

void

quicksort()

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

Template quick-sort.

Preforms 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

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

Returns

void

testCross() [1/4]

```
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	*m1*	Raw matrix reference
in	*m2*	Raw matrix reference

Returns

True if the cross-product is defined

testCross() [2/4]

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

Tests if the cross-product is a legal operation.

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

Parameters

in	*m1*	Indirect matrix reference
in	*m2*	Raw matrix reference

Returns

True if the cross-product is defined

testCross() [3/4]

```
template<class dataType >
bool os::testCross (
```

```
            const matrix< dataType > & m1,
            const indirectMatrix< dataType > & m2 ) throw )
```

Tests if the cross-product is a legal operation.

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

Parameters

in	*m1*	Raw matrix reference
in	*m2*	Indirect matrix reference

Returns

True if the cross-product is defined

testCross() [4/4]

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

Tests if the cross-product is a legal operation.

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

Parameters

in	*m1*	Indirect matrix reference
in	*m2*	Indirect matrix reference

Returns

True if the cross-product is defined

11.1.4 Variable Documentation

oserr_ptr

```
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.

osout_ptr

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

Standard out pointer for os namespace.

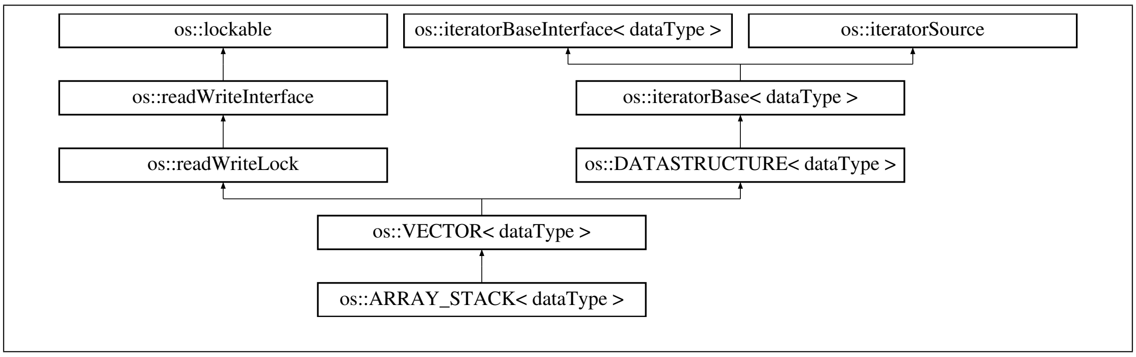
This std::ostream is used as standard out for the os namespace. This pointer can be swapped out to programmatically redirect standard out for the os namespace.

Chapter 12

Class Documentation

12.1 os::ARRAY_STACK< dataType > Class Template Reference

Inheritance diagram for os::ARRAY_STACK< dataType >:



Public Member Functions

- **ARRAY_STACK** ()

  Default constructor Simple constructor which implicitly call the constructor of it's base.
- ~**ARRAY_STACK** () final

  Destroys the stack Uses the destructor function defined by the ***VECTOR*** *(p. 336) class.*
- void **pop** ()

  Remove top element.
- bool **push** (const dataType &x)

  Add element.
- const dataType & **constTop** () const

  Const access top element.
- dataType & **top** ()

  Access top element.
- const dataType & **top** () const

Const access top element.

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

Additional Inherited Members

12.1.1 Constructor & Destructor Documentation

ARRAY_STACK()

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

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

~ARRAY_STACK()

template<class dataType >
os::ARRAY_STACK< dataType >::~**ARRAY_STACK** ( ) [inline], [final]

Destroys the stack Uses the destructor function defined by the **VECTOR** (p. 336) class.

12.1.2 Member Function Documentation

constTop() [1/2]

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

Const access top element.

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

Returns

Immutable top element

constTop() [2/2]

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

pop()

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

Remove top element.

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

Returns

void

push() [1/2]

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

Add element.

Inserts an element at the top of the stack

Parameters

in	*x*	Element to be inserted

Returns

true

push() [2/2]

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

top() [1/4]

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

Access top element.

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

Returns

Mutable top element

top() [2/4]

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

Const access top element.

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

Returns

Immutable top element

top() [3/4]

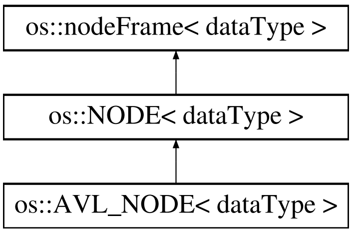
```
template<class dataType >
smart_ptr<dataType> os::ARRAY_STACK< dataType >::top ( ) [inline]
```

top() [4/4]

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

12.2 os::AVL_NODE< dataType > Class Template Reference

AVL tree node Node used in an AVL tree. This node allows both itteration and random access.
Inheritance diagram for os::AVL_NODE< dataType >:



Public Member Functions

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

- const **smart_ptr**< **nodeFrame**< dataType > > **getNextConst** () const final throw (descriptiveException)

  Find the next node.
- const **smart_ptr**< **nodeFrame**< dataType > > **getPrevConst** () const final throw (descriptiveException)

  Find the previous node.
- **smart_ptr**< **nodeFrame**< dataType > > **access** (long offset) final throw (descriptiveException)

  Access node by index.
- const **smart_ptr**< **nodeFrame**< dataType > > **constAccess** (long offset) const final throw (descriptiveException)

  Access node by index.

Static Public Attributes

- static const bool **ITERABLE** = true

  AVL nodes are iterable.
- static const bool **RANDOM_ACCESS** = true

  AVL nodes allow random-access.

Private Member Functions

- void **propigateUp** (int isFirst=0)

  Processes both height and children count This function is designed to move recursively up the tree.
- void **setChild** (**smart_ptr**< **AVL_NODE**< dataType > > c)

  Add a child to this node.
- void **setParent** (**smart_ptr**< **AVL_NODE**< dataType > > p, **smart_ptr**< **AVL_NODE**< dataType > > self_pointer)

  Sets the parent node.
- void **removeChild** (**smart_ptr**< **AVL_NODE**< dataType > > c)

  Remove a child from this node.
- void **removeChild** (int pos)

  Remove a child from this node.
- void **removeParent** ()

  Remove the parent node.
- void **purge** ()

  Remove all children and parents.
- **AVL_NODE** (**AVL_TREE**< dataType > ∗src, const dataType &dt)

  Private constructor.

Private Attributes

- **smart_ptr**< **AVL_NODE**< dataType > > **_parent**

  Parent node one level up in the tree.
- **smart_ptr**< **AVL_NODE**< dataType > > **_child** [2]

  Children nodes.
- size_t **_height**

  The height of the tree.
- size_t **_numChildren**

  Number of children.

Additional Inherited Members

12.2.1 Detailed Description

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

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

12.2.2 Constructor & Destructor Documentation

AVL_NODE()

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

Private constructor.
Only AVL trees can access the node constructor.

Parameters

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

12.2.3 Member Function Documentation

access()

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

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

Returns

Offset node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 276).

child() [1/2]

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

Returns a mutable child by index.

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

Returns

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

child() [2/2]

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

Returns an immutable child by index.

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

Returns

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

constAccess()

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

Access node by index.

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

Returns

Offset node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 276).

getNext()

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

Find the next node.

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

Returns

Pointer to the next node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 277).

getNextConst()

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

Find the next node.

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

Returns

Pointer to the next node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 278).

getPrev()

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

Find the previous node.

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

Returns

Pointer to the previous node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 278).

getPrevConst()

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

Find the previous node.

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

Returns

Pointer to the previous node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 279).

height()

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

Returns the height of the sub-tree.

Returns

os::AVL_NODE<**dataType**>**::_height** (p. 133)

iterable()

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

Returns if the node is iterable.

Returns

object::ITERABLE

Reimplemented from **os::nodeFrame**< **dataType** > (p. 279).

numChildren()

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

Returns the number of children.

Returns

os::AVL_NODE<**dataType**>**::_numChildren** (p. 133)

parent() [1/2]

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

Returns the mutable parent node.

Returns

Modifiable **os::AVL_NODE**<**dataType**>**::_parent** (p. 133)

parent() [2/2]

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

Returns an immutable parent node.

Returns

Constant **os::AVL_NODE<dataType>::_parent** (p. 133)

propigateUp()

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

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

Parameters

in	*isFirst*	0 by default, increments each time

Returns

void

purge()

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

Remove all children and parents.

This function is important because nodes are of type **os::smart_ptr** (p. 311), since there are co-dependencies, failure to run this function on deletion of the tree will cause a memory leak.

Returns

void

randomAccess()

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

Returns if the node can be accessed randomly.

Returns

object::RANDOM_ACCESS

Reimplemented from **os::nodeFrame< dataType >** (p. 281).

remove()

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

Remove this node from the tree.

Returns

void

Reimplemented from **os::nodeFrame**< **dataType** > (p. 281).

removeChild() [1/2]

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

Remove a child from this node.

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

Parameters

in	*c*	Node to be removed

Returns

void

removeChild() [2/2]

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

Remove a child from this node.

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

Parameters

in	*pos*	Node index to be removed

Returns

void

removeParent()

```
template<class dataType >
```

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

Remove the parent node.

Returns

void

setChild()

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

Add a child to this node.

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

Parameters

in	*c*	Node to be inserted

Returns

void

setParent()

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

Sets the parent node.

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

Parameters

in	*p*	Parent node
in	*self_pointer*	Pointer to self, with memory management

Returns

void

12.2.4 Member Data Documentation

_child

`template<class dataType >`

smart_ptr<**AVL_NODE**<dataType> > **os::AVL_NODE**< dataType >::_child[2] [private]

Children nodes.

_height

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

The height of the tree.

This variable is kept to reduce computation time. It is dependent on the height of a node's children nodes. The **AVL_NODE**<**dataType**>**::propigateUp()** (p. 130) resets the height based on the height of the node's children.

_numChildren

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

Number of children.

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

_parent

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

Parent node one level up in the tree.

ITERABLE

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

AVL nodes are iterable.

RANDOM_ACCESS

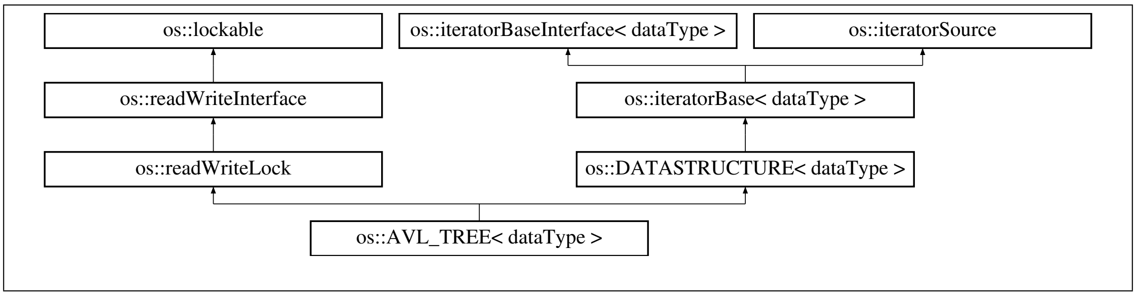
template<class dataType >
const bool **os::AVL_NODE**< dataType >::RANDOM_ACCESS = true [static]

AVL nodes allow random-access.

12.3 os::AVL_TREE< dataType > Class Template Reference

Template AVL Tree definition.

Inheritance diagram for os::AVL_TREE< dataType >:



Public Member Functions

- **AVL_TREE** ()
  Default constructor Initializes an empty AVL tree.
- **AVL_TREE** (const **AVL_TREE**< dataType > &cpy)
  Copy constructor.
- ∼**AVL_TREE** () final
  AVL tree destructor Destroys all objects held in the tree. Note that in the case the tree is holding references, the pointers will be destroyed instead of the objects.
- size_t **size** () const final
  Access size of the tree.
- bool **iterable** () const final
  Returns if the relevant node type is iterable.
- bool **randomAccess** () const final
  Returns if the relevant node type can be accessed randomly.
- bool **insert** (const dataType &x) final
  Insert item into the tree Inserts an item into the tree and procedes to re-balance the tree.
- bool **remove** (const dataType &x) final
  Remove item from the data-structure.
- bool **find** (const dataType &x) const final
  Searches for an item.
- dataType & **access** (const dataType &x) final
  Mutable item access.
- const dataType & **access** (const dataType &x) const final
  Immutable item access.
- dataType & **at** (size_t i) final throw (descriptiveException)
  Access the tree by index.
- const dataType & **at** (size_t i) const final throw (descriptiveException)
  Constant access the tree by index.

Static Public Attributes

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

Protected Member Functions

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

Private Member Functions

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

Private Attributes

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

Additional Inherited Members

12.3.1 Detailed Description

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

Template AVL Tree definition.

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

12.3.2 Constructor & Destructor Documentation

AVL_TREE() [1/2]

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

Default constructor Initializes an empty AVL tree.

AVL_TREE() [2/2]

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

Copy constructor.

This constructor builds an AVL tree from another AVL tree. Note that this copies by value, not reference.

Parameters

in	*cpy*	Target to be copied

~AVL_TREE()

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

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

12.3.3 Member Function Documentation

access() [1/2]

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

Mutable item access.

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

Returns

Mutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 179).

access() [2/2]

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

Immutable item access.

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

Returns

Immutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 181).

at() [1/2]

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

Access the tree by index.

Parameters

in	*i*	Index of the element

Returns

Reference to ith element of the tree

Reimplemented from **os::DATASTRUCTURE**< **dataType** > (p. 181).

at() [2/2]

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

Constant access the tree by index.

Parameters

in	*i*	Index of the element

Returns

Immutable reference to ith element of the tree

Reimplemented from **os::DATASTRUCTURE**< **dataType** > (p. 182).

balance()

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

Balances a single node.

Parameters

in	*x*	Node to be balanced

Returns

true if the node is already balanced, else, false

balanceDelete()

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

Removes a node and balances the tree.

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

Parameters

in	*x*	Node to be deleted

Returns

true if successful, false if failed

balanceUp()

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

Balances this node and ancestor nodes.

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

Parameters

in	*x*	Node to be balanced

Returns

void

checkBalance()

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

Checks if a sub-tree is balanced.

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

Parameters

in	*x*	Node to be checked

Returns

true if balanced, false if not

doubleRotation()

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

Double-rotate a node.

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

Parameters

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

Returns

true if successful, else, false

find()

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

Searches for an item.

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

Returns

True if found, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 182).

findBottom()

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

Find first or last node in a tree.

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

Parameters

in	*x*	Starting node
in	*dir*	Direction node to search in

Returns

First or last node in sub-tree

getFirstNode()

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

Access to first node.

Returns

First node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 230).

getFirstNodeConst()

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

Constant access to first node.

Returns

Immutable first node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 230).

getLastNode()

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

Access to last node.

Returns

Last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 231).

getLastNodeConst()

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

Constant access to last node.

Returns

Immutable last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 231).

insert()

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

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

Parameters

in	*x*	Data to be bound

Returns

True if successful, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 182).

iterable()

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

Returns if the relevant node type is iterable.

Returns

true

Reimplemented from **os::iteratorSource** (p. 239).

randomAccess()

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

Returns if the relevant node type can be accessed randomly.

Returns

true

Reimplemented from **os::iteratorSource** (p. 240).

remove()

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

Remove item from the data-structure.

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

Returns

True if removed, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 184).

searchNode()

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

Search for a node.

Parameters

in	*dt*	Pointer to search for

Returns

Muttable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 232).

searchNodeConst()

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

Const search for a node.

Parameters

in	*dt*	Pointer to search for

Returns

Immutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 232).

singleRotation()

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

Rotates a node.

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

Parameters

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

Returns

true if successful, else, false

size()

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

Access size of the tree.

Returns

Number of elements in the tree

Implements **os::DATASTRUCTURE**< **dataType** > (p. 184).

12.3.4 Member Data Documentation

_root

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

Root node of the tree.

ITERABLE

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

AVL trees are iterable.

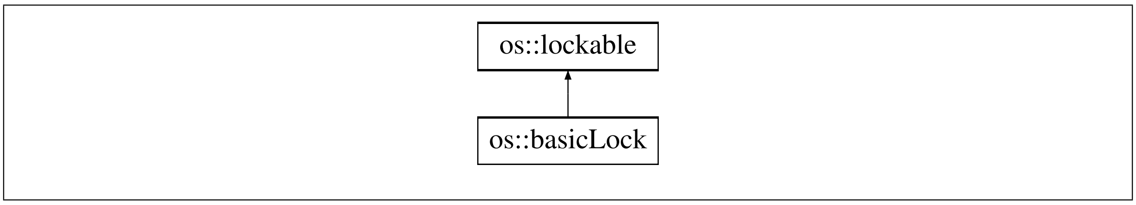
RANDOM_ACCESS

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

AVL trees allow random access.

12.4 os::basicLock Class Reference

Basic lock Wraps the std::mutex class allowing it to be accessed in a constant way.
Inheritance diagram for os::basicLock:



Public Member Functions

- **basicLock** (bool dMode=**NO_LOCK_CHECK**) throw ()

  Default constructor.
- virtual ~**basicLock** () throw (descriptiveException)

  Virtual destructor.
- void **lock** () const final throw ()

  Locks the std::mutex.
- void **unlock** () const final throw (descriptiveException)

  Unlocks the std::mutex.

- bool **locked** () const final throw ()

  Checks if the lock is locked.
- bool **try_lock** () const final throw ()

  Attempt to lock the object.

Private Member Functions

- **basicLock** (const **basicLock** &cpy)

  Undefined copy-constructor.

Private Attributes

- std::mutex **_mtx**

  Mutable mutex.
- bool **_lockedStatus**

  Locked flag.

Additional Inherited Members

12.4.1 Detailed Description

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

12.4.2 Constructor & Destructor Documentation

basicLock() [1/2]

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

Undefined copy-constructor.

basicLock() [2/2]

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

Default constructor.

~basicLock()

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

Virtual destructor.

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

12.4.3 Member Function Documentation

lock()

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

Locks the std::mutex.

Returns

void

Implements **os::lockable** (p. 261).

locked()

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

Checks if the lock is locked.

Returns

True if locked, else, false

Implements **os::lockable** (p. 261).

try_lock()

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

Attempt to lock the object.

Locks the object if possible, otherwise, returns false.

Returns

True if lock successful

Implements **os::lockable** (p. 261).

unlock()

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

Unlocks the std::mutex.

Returns

void

Implements **os::lockable** (p. 262).

12.4.4 Member Data Documentation

_lockedStatus

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

Locked flag.

_mtx

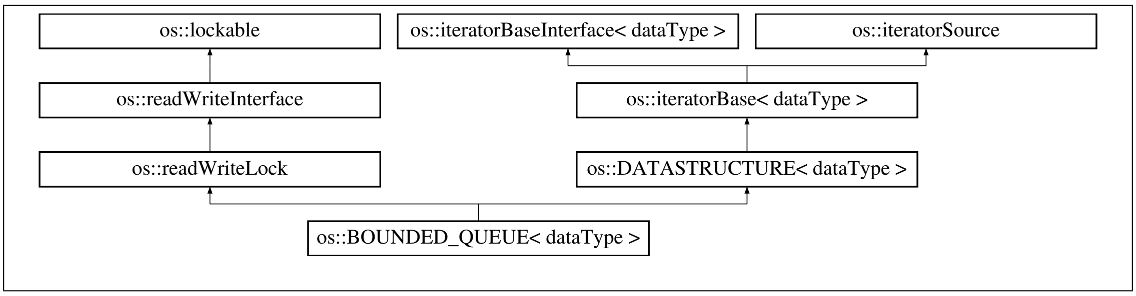
```
std::mutex os::basicLock::_mtx  [mutable], [private]
```

Mutable mutex.

12.5 os::BOUNDED_QUEUE< dataType > Class Template Reference

Template **BOUNDED_QUEUE** (p. 147) definition.

Inheritance diagram for os::BOUNDED_QUEUE< dataType >:



Public Member Functions

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

  Default constructor.
- **BOUNDED_QUEUE** (const **BOUNDED_QUEUE**< dataType > &cpy)

  Copy constructor.
- virtual ~**BOUNDED_QUEUE** () final

  ***BOUNDED_QUEUE** (p. 147) destructor Destroys all objects held in the array. Note that in the case the **BOUNDED_QUEUE** (p. 147) is holding references, the pointers will be destroyed instead of the objects.*
- size_t **boundSize** () const

  Array bound size.
- void **pop** ()

  Remove top element.
- void **setBound** (size_t sz)

  Sets the size of the bounding array.
- bool **insert** (const dataType &x) final

  *Insert item into the **BOUNDED_QUEUE** (p. 147).*
- bool **remove** (const dataType &x) final

  Remove item from the data-structure.
- bool **find** (const dataType &x) const final

  Searches for an item.
- dataType & **access** (const dataType &x) final

  Mutable item access.

- const dataType & **access** (const dataType &x) const final

  Immutable item access.
- dataType & **at** (size_t i) final throw (descriptiveException)

  *Access the **BOUNDED_QUEUE** (p. 147) by index.*
- const dataType & **at** (size_t i) const final throw (descriptiveException)

  *Access the **BOUNDED_QUEUE** (p. 147) by index.*
- bool **push** (const dataType &x)

  Add element.
- const dataType & **constTop** () const

  Const access top element.
- dataType & **top** ()

  Access top element.
- const dataType & **top** () const

  Const access top element.
- size_t **size** () const final

  *Access size of the **BOUNDED_QUEUE** (p. 147).*
- bool **iterable** () const final

  Returns if the relevant node type is iterable.
- bool **randomAccess** () const final

  Returns if the relevant node type can be accessed randomly.

Static Public Attributes

- static const bool **ITERABLE** = true

  BOUNDED_QUEUEs are iterable.
- static const bool **RANDOM_ACCESS** = true

  BOUNDED_QUEUEs allow random access.

Protected Member Functions

- **smart_ptr**< **nodeFrame**< dataType > > **getFirstNode** () final

  Access to first node.
- **smart_ptr**< **nodeFrame**< dataType > > **getLastNode** () final

  Access to last node.
- const **smart_ptr**< **nodeFrame**< dataType > > **getFirstNodeConst** () const final

  Constant access to first node.
- const **smart_ptr**< **nodeFrame**< dataType > > **getLastNodeConst** () const final

  Constant access to last node.
- **smart_ptr**< **nodeFrame**< dataType > > **searchNode** (const **smart_ptr**< dataType > dt) final

  Search for a node.
- const **smart_ptr**< **nodeFrame**< dataType > > **searchNodeConst** (const **smart_ptr**< data↩ Type > dt) const final

  Const search for a node.

Private Attributes

- dataType * **_array**

  Pointer to array of data.
- size_t **_arraySize**

  Size of the available memory.
- size_t **numElms**

  Number of elements in the ***BOUNDED_QUEUE*** *(p. 147).*
- size_t **pos**

  Starting position.

Additional Inherited Members

12.5.1 Detailed Description

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

Template **BOUNDED_QUEUE** (p. 147) definition.

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

12.5.2 Constructor & Destructor Documentation

BOUNDED_QUEUE() [1/2]

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

Default constructor.

This constructor builds the vecotr with a certain size. Note that the **BOUNDED_QUEUE** (p. 147) will always be of, at least, size 32.

Parameters

in	*sz*	Target size of **BOUNDED_QUEUE** (p. 147)

BOUNDED_QUEUE() [2/2]

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

Copy constructor.

This constructor builds a vector from another vector. Note that this copies by value, not reference.

Parameters

in	*cpy*	Target to be copied

~BOUNDED_QUEUE()

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

BOUNDED_QUEUE (p. 147) destructor Destroys all objects held in the array. Note that in the case the **BOUNDED_QUEUE** (p. 147) is holding references, the pointers will be destroyed instead of the objects.

12.5.3 Member Function Documentation

access() [1/2]

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

Mutable item access.

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

Returns

Mutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 179).

access() [2/2]

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

Immutable item access.

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

Returns

Immutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 181).

at() [1/2]

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

Access the **BOUNDED_QUEUE** (p. 147) by index.

Parameters

in	*i*	Index of the array

Returns

Reference to ith element of the **BOUNDED_QUEUE** (p. 147)

Reimplemented from **os::DATASTRUCTURE**< **dataType** > (p. 181).

at() [2/2]

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

Access the **BOUNDED_QUEUE** (p. 147) by index.

Parameters

in	*i*	Index of the array

Returns

Immutable reference to ith element of the **BOUNDED_QUEUE** (p. 147)

Reimplemented from **os::DATASTRUCTURE**< **dataType** > (p. 182).

boundSize()

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

Array bound size.

Returns

os::BOUNDED_QUEUE<**dataType**>**::_arraySize** (p. 156)

constTop()

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

Const access top element.

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

Returns

Immutable top element

find()

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

Searches for an item.

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

Returns

True if found, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 182).

getFirstNode()

template<class dataType>
smart_ptr<**nodeFrame**<dataType> > **os::BOUNDED_QUEUE**< dataType >::getFirstNode ( ) [final], [protected], [virtual]

Access to first node.

Returns

First node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 230).

getFirstNodeConst()

template<class dataType>
const **smart_ptr**<**nodeFrame**<dataType> > **os::BOUNDED_QUEUE**< dataType >::getFirstNodeConst ( ) const [final], [protected], [virtual]

Constant access to first node.

Returns

Immutable first node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 230).

getLastNode()

template<class dataType>
smart_ptr<**nodeFrame**<dataType> > **os::BOUNDED_QUEUE**< dataType >::getLastNode ( ) [final], [protected], [virtual]

Access to last node.

Returns

Last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 231).

getLastNodeConst()

template<class dataType>
const **smart_ptr**<**nodeFrame**<dataType> > **os::BOUNDED_QUEUE**< dataType >::getLastNodeConst ( ) const [final], [protected], [virtual]

Constant access to last node.

Returns

Immutable last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 231).

insert()

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

Insert item into the **BOUNDED_QUEUE** (p. 147).

Inserts an item at the end of the **BOUNDED_QUEUE** (p. 147). Note that different forms of the **BOUNDED_QUEUE** (p. 147) accept pointers instead of object references.

Returns

True

Implements **os::DATASTRUCTURE< dataType >** (p. 182).

iterable()

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

Returns if the relevant node type is iterable.

Returns

true

Reimplemented from **os::iteratorSource** (p. 239).

pop()

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

Remove top element.

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

Returns

void

push()

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

Add element.

Inserts an element at the front of the queue

Parameters

in	*x*	Element to be inserted

Returns

true

randomAccess()

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

Returns if the relevant node type can be accessed randomly.

Returns

true

Reimplemented from **os::iteratorSource** (p. 240).

remove()

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

Remove item from the data-structure.

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

Returns

True if removed, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 184).

searchNode()

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

Search for a node.

Parameters

in	*dt*	Pointer to search for

Returns

Muttable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 232).

searchNodeConst()

```
template<class dataType>
```

const **smart_ptr**<**nodeFrame**<dataType> > **os::BOUNDED_QUEUE**< dataType >::searchNodeConst (
const **smart_ptr**< dataType > *dt*) const [final], [protected], [virtual]

Const search for a node.

Parameters

in	*dt*	Pointer to search for

Returns

Immutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 232).

setBound()

template<class dataType>
void **os::BOUNDED_QUEUE**< dataType >::setBound (
size_t *sz*) [inline]

Sets the size of the bounding array.

Parameters

in	*sz*	Bound size

Returns

void

size()

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

Access size of the **BOUNDED_QUEUE** (p. 147).

Returns

Number of elements in the **BOUNDED_QUEUE** (p. 147)

Implements **os::DATASTRUCTURE**< **dataType** > (p. 184).

top() [1/2]

template<class dataType>
dataType& **os::BOUNDED_QUEUE**< dataType >::top ( ) [inline]

Access top element.

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

Returns

Mutable top element

top() [2/2]

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

Const access top element.

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

Returns

Immutable top element

12.5.4 Member Data Documentation

_array

`template<class dataType>`
`dataType* `**os::BOUNDED_QUEUE**`< dataType >::_array  [private]`

Pointer to array of data.

_arraySize

`template<class dataType>`
`size_t `**os::BOUNDED_QUEUE**`< dataType >::_arraySize  [private]`

Size of the available memory.

ITERABLE

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

BOUNDED_QUEUEs are iterable.

numElms

`template<class dataType>`
`size_t `**os::BOUNDED_QUEUE**`< dataType >::numElms  [private]`

Number of elements in the **BOUNDED_QUEUE** (p. 147).

pos

`template<class dataType>`
`size_t `**os::BOUNDED_QUEUE**`< dataType >::pos  [private]`

Starting position.

RANDOM_ACCESS

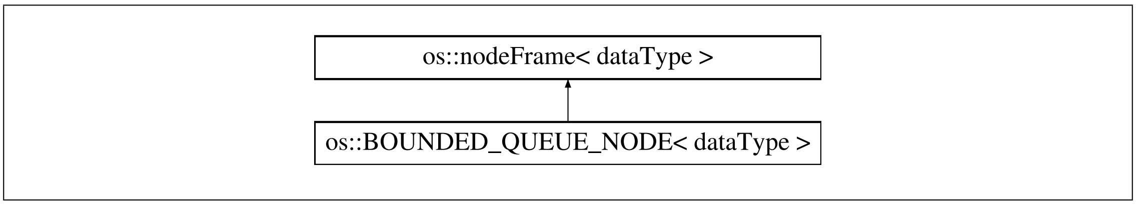
```
template<class dataType>
const bool os::BOUNDED_QUEUE< dataType >::RANDOM_ACCESS = true  [static]
```

BOUNDED_QUEUEs allow random access.

12.6 os::BOUNDED_QUEUE_NODE< dataType > Class Template Reference

BOUNDED_QUEUE (p. 147) node.

Inheritance diagram for os::BOUNDED_QUEUE_NODE< dataType >:



Public Member Functions

- ∼**BOUNDED_QUEUE_NODE** ()

  Destructor Class is not designed to be inherited from.
- **smart_ptr**< dataType > **get** () final throw ()

  Returns a pointer.
- const **smart_ptr**< dataType > **constGet** () const final throw ()

  Returns a const pointer.
- dataType & **operator*** () final throw (descriptiveException)

  De-reference.
- const dataType & **operator*** () const final throw (descriptiveException)

  De-reference.
- bool **valid** () const final

  Valid data query Checks if the provided data is valid.
- bool **iterable** () const final

  Returns if the node is iterable.
- bool **randomAccess** () const final

  Returns if the node can be accessed randomly.
- **smart_ptr**< **nodeFrame**< dataType > > **getNext** () final throw (descriptiveException)

  Returns the next ***BOUNDED_QUEUE*** *(p. 147) frame.*
- const **smart_ptr**< **nodeFrame**< dataType > > **getNextConst** () const final throw (descriptive↩ Exception)

  Returns the next ***BOUNDED_QUEUE*** *(p. 147) frame.*
- **smart_ptr**< **nodeFrame**< dataType > > **getPrev** () final throw (descriptiveException)

  Returns the previous ***BOUNDED_QUEUE*** *(p. 147) frame.*

- const **smart_ptr**< **nodeFrame**< dataType > > **getPrevConst** () const final throw (descriptive↩ Exception)

    Returns the previous ***BOUNDED_QUEUE*** *(p. 147) frame.*
- **smart_ptr**< **nodeFrame**< dataType > > **access** (long offset) final throw (descriptiveException)

    Access node by index.
- const **smart_ptr**< **nodeFrame**< dataType > > **constAccess** (long offset) const final throw (descriptiveException)

    Access node by index.
- void **remove** () final throw (descriptiveException)

    Remove this node from the ***BOUNDED_QUEUE*** *(p. 147).*

Static Public Attributes

- static const bool **ITERABLE** = true

    BOUNDED_QUEUE *(p. 147) frames are iterable.*
- static const bool **RANDOM_ACCESS** = true

    BOUNDED_QUEUE *(p. 147) frames allow random-access.*

Private Member Functions

- **BOUNDED_QUEUE_NODE** (**BOUNDED_QUEUE**< dataType > *src, size_t pos)

    Private constructor This constructor is not designed to be accessed by anything other than the ***BO↩ UNDED_QUEUE*** *(p. 147) class derivatives.*

Private Attributes

- size_t **position**

    Current position of the ***BOUNDED_QUEUE*** *(p. 147) iterator.*

12.6.1 Detailed Description

template<class dataType>
class os::BOUNDED_QUEUE_NODE< dataType >

BOUNDED_QUEUE (p. 147) node.

Used by the iterator to iterate through a **BOUNDED_QUEUE** (p. 147).

12.6.2 Constructor & Destructor Documentation

BOUNDED_QUEUE_NODE()

```
template<class dataType >
os::BOUNDED_QUEUE_NODE< dataType >::BOUNDED_QUEUE_NODE (
            BOUNDED_QUEUE< dataType > * src,
            size_t pos )  [inline], [private]
```

Private constructor This constructor is not designed to be accessed by anything other than the **BOUNDED_QUEUE** (p. 147) class derivatives.

~BOUNDED_QUEUE_NODE()

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

Destructor Class is not designed to be inherited from.

12.6.3 Member Function Documentation

access()

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

Access node by index.

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

Returns

Offset node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 276).

constAccess()

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

Access node by index.

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

Returns

Offset node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 276).

constGet()

`template<class dataType >`
`const` **smart_ptr**`<dataType>` **os::BOUNDED_QUEUE_NODE**`< dataType >::constGet ( ) const throw ) [inline], [final], [virtual]`

Returns a const pointer.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

Implements **os::nodeFrame**< **dataType** > (p. 277).

get()

template<class dataType >
smart_ptr<dataType> **os::BOUNDED_QUEUE_NODE**< dataType >::get ( ) throw) [inline], [final], [virtual]

Returns a pointer.

Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

Implements **os::nodeFrame**< **dataType** > (p. 277).

getNext()

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

Returns the next **BOUNDED_QUEUE** (p. 147) frame.

Returns

Next node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 277).

getNextConst()

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

Returns the next **BOUNDED_QUEUE** (p. 147) frame.

Returns

Next node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 278).

getPrev()

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

Returns the previous **BOUNDED_QUEUE** (p. 147) frame.

Returns

Previous node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 278).

getPrevConst()

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

Returns the previous **BOUNDED_QUEUE** (p. 147) frame.

Returns

Previous node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 279).

iterable()

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

Returns if the node is iterable.

Returns

object::ITERABLE

Reimplemented from **os::nodeFrame**< **dataType** > (p. 279).

operator*() [1/2]

`template<class dataType >`
`dataType&` **os::BOUNDED_QUEUE_NODE**`< dataType >::operator* ( ) throw` **descriptiveException**`) [inline], [final], [virtual]`

De-reference.
Returns a reference to the contained object, the reference can be modified.

Returns

Contained object

Implements **os::nodeFrame**< **dataType** > (p. 280).

operator*() [2/2]

`template<class dataType >`
`const dataType&` **os::BOUNDED_QUEUE_NODE**`< dataType >::operator* ( ) const throw` **descriptiveException**`) [inline], [final], [virtual]`

De-reference.
Returns a const reference to the contained object, the reference cannot be modified.

Returns

Contained object

Implements **os::nodeFrame**< **dataType** > (p. 280).

randomAccess()

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

Returns if the node can be accessed randomly.

Returns

object::RANDOM_ACCESS

Reimplemented from **os::nodeFrame**< **dataType** > (p. 281).

remove()

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

Remove this node from the **BOUNDED_QUEUE** (p. 147).

Returns

void

Reimplemented from **os::nodeFrame**< **dataType** > (p. 281).

valid()

`template<class dataType >`
`bool` **os::BOUNDED_QUEUE_NODE**`< dataType >::valid ( ) const [inline], [final], [virtual]`

Valid data query Checks if the provided data is valid.

Returns

true if valid, else, false

Reimplemented from **os::nodeFrame**< **dataType** > (p. 282).

12.6.4 Member Data Documentation

ITERABLE

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

BOUNDED_QUEUE (p. 147) frames are iterable.

position

`template<class dataType >`
`size_t` **os::BOUNDED_QUEUE_NODE**`< dataType >::position [private]`

Current position of the **BOUNDED_QUEUE** (p. 147) iterator.

RANDOM_ACCESS

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

BOUNDED_QUEUE (p. 147) frames allow random-access.

12.7 os::constantPrinter Class Reference

Prints constant arrays to files.

Public Member Functions

- **constantPrinter** (std::string fileName, bool has_cpp=false) throw (descriptiveException)
  Single constructor.
- virtual ~**constantPrinter** () throw (descriptiveException)
  Virtual destructor.
- void **addInclude** (std::string includeName) throw (descriptiveException)
  Add include file.
- void **addNamespace** (std::string namesp) throw (descriptiveException)
  Add a namespace.
- void **removeNamespace** () throw (descriptiveException)
  Remove namespace.
- void **addComment** (std::string comment) throw (descriptiveException)
  Insert a comment.
- bool **hasCPP** () const throw ()
  Returns if the object is writing to a .cpp file.
- bool **good** () const throw ()
  Checks file status.
- void **addArray** (std::string name, uint32_t *arr, unsigned length) throw (descriptiveException)
  Add a uin32_t array.*

Private Member Functions

- std::string **capitalize** (std::string str) const throw ()
  Capitalizes the string argument.
- std::string **tabs** () const throw ()
  Returns current tab depth.

Private Attributes

- std::ofstream **hFile**
  Output file for the .h file.
- std::ofstream **cppFile**
  Output file for the .cpp file.

- bool **_has_cpp**

    Holds if the object is generating a .cpp.
- unsigned **namespaceDepth**

    Current namespace depth.

12.7.1 Detailed Description

Prints constant arrays to files.

This class outputs configured and populated constant arrays into .h and .cpp files, depending on the configuration. This class is meant to be used as a tool for automatically generating source code files.

12.7.2 Constructor & Destructor Documentation

constantPrinter()

```
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	*fileName*	String representing the file name
in	*has_cpp*	Optional boolean defining if a .cpp will be written

~constantPrinter()

```
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

addArray()

```
void os::constantPrinter::addArray (
            std::string name,
            uint32_t * arr,
            unsigned length ) throw descriptiveException)
```

Add a uin32_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	*arr*	Array to be written to the files
in	*length*	Length of the received array

Returns

void

addComment()

```
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	*comment*	Comment string to be added as a comment

Returns

void

addInclude()

```
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	*includeName*	Name of header file to be included

Returns

void

addNamespace()

```
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. 167). Both the .h and .cpp file have this namespace added.

Parameters

in	*namesp*	Namespace added to the file

Returns

void

capitalize()

```
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

good()

```
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

hasCPP()

```
bool os::constantPrinter::hasCPP ( ) const throw )   [inline]
```

Returns if the object is writing to a .cpp file.

Returns

constantPrinter::_has_cpp (p. 167)

removeNamespace()

`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. 167).

Returns

void

tabs()

`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. 167) tabs

12.7.4 Member Data Documentation

_has_cpp

`bool os::constantPrinter::_has_cpp  [private]`

Holds if the object is generating a .cpp.

cppFile

`std::ofstream os::constantPrinter::cppFile  [private]`

Output file for the .cpp file.

hFile

`std::ofstream os::constantPrinter::hFile  [private]`

Output file for the .h file.

namespaceDepth

`unsigned os::constantPrinter::namespaceDepth  [private]`

Current namespace depth.

In order to streamline large projects, arrays of constants should be placed inside namespaces. This variable allows for the creation and management of nested namespaces.

12.8 os::constIterator< dataType > Class Template Reference

Generalized constant iterator.

Public Member Functions

- **constIterator** ()

  Default constructor.
- **constIterator** (const **constIterator**< dataType > &cpy)

  Copy constructor.
- **constIterator** (const **iterator**< dataType > &cpy)

  Copy constructor.
- virtual ~**constIterator** ()

  Virtual destructor.
- bool **operator!** () const throw ()

  Inverted boolean conversion.
- **operator bool** () const throw ()

  Boolean conversion.
- const **smart_ptr**< dataType > **constGet** () const throw (descriptiveException)

  Member access.
- const **smart_ptr**< dataType > **get** () const throw (descriptiveException)

  Member access.
- const **smart_ptr**< dataType > **operator &** () const throw (descriptiveException)

  Member access.
- const **smart_ptr**< dataType > **operator->** () const throw (descriptiveException)

  Member access.
- const dataType & **operator∗** () const throw (descriptiveException)

  De-reference.
- **constIterator**< dataType > **constAccess** (long offset) const throw (descriptiveException)

  Access iterator by index.
- **constIterator**< dataType > **access** (long offset) const throw (descriptiveException)

  Access iterator by index.
- const dataType & **operator[ ]** (size_t offset) const throw (descriptiveException)
- const dataType & **operator[ ]** (long offset) const throw (descriptiveException)
- const dataType & **operator[ ]** (int offset) const throw (descriptiveException)
- **constIterator**< dataType > **operator+** (size_t offset) const throw (descriptiveException)
- **constIterator**< dataType > **operator-** (size_t offset) const throw (descriptiveException)
- **constIterator**< dataType > **operator+** (long offset) const throw (descriptiveException)
- **constIterator**< dataType > **operator-** (long offset) const throw (descriptiveException)
- **constIterator**< dataType > **operator+** (int offset) const throw (descriptiveException)
- **constIterator**< dataType > **operator-** (int offset) const throw (descriptiveException)
- const **constIterator**< dataType > & **increment** (long offset=1) const

  Increment this iterator Increments this iterator by some value, by default, 1. Note that if incrementing by more than one, the node in question must support random access.
- const **constIterator**< dataType > & **decrement** (long offset=1) const

  Decrement this iterator Decrements this iterator by some value, by default, 1. Note that if decrementing by more than one, the node in question must support random access.
- const **constIterator**< dataType > & **operator+=** (size_t offset) const

- const **constIterator**< dataType > & **operator-=** (size_t offset) const
- const **constIterator**< dataType > & **operator+=** (long offset) const
- const **constIterator**< dataType > & **operator-=** (long offset) const
- const **constIterator**< dataType > & **operator+=** (int offset) const
- const **constIterator**< dataType > & **operator-=** (int offset) const
- **constIterator**< dataType > **operator++** (int param) const throw (descriptiveException)
- **constIterator**< dataType > **operator--** (int param) const throw (descriptiveException)
- const **constIterator**< dataType > & **operator++** () const throw (descriptiveException)
- const **constIterator**< dataType > & **operator--** () const throw (descriptiveException)
- bool **iterable** () const

  Returns if the source is iterable.
- bool **randomAccess** () const

  Returns if the source can be accessed randomly.
- int **compare** (const **constIterator**< dataType > &cmp) const

  Compares two const iterators Uses the comparison definition of the node in an iterator to compare two iterators.
- **operator size_t** () const

  Casts to size_t for hash functions Uses the size definition of the current node.
- COMPARE_OPERATORS int **compare** (const **iterator**< dataType > &cmp) const

  *Compares a **constIterator** (p. 167) and an iterator Uses the comparison definition of the node in an iterator to compare two iterators.*

Private Member Functions

- **constIterator** (const **smart_ptr**< **nodeFrame**< dataType > > node, const **iteratorBase**< dataType > &src)

  Construct with a frame and source.

Private Attributes

- const **iteratorBase**< dataType > ∗ **source_structure**

  Pointer to the iterator source.
- **smart_ptr**< **nodeFrame**< dataType > > **currentNode**

  Pointer to the current node.

Friends

- class **iteratorBase**< **dataType** >

  ***iteratorBase** (p. 228) must have access to constructor*
- class **iterator**< **dataType** >

  Two iterator types are friends.

12.8.1 Detailed Description

template<class dataType>
class os::constIterator< dataType >

Generalized constant iterator.

Since this iterator wraps the node frame, it can be generally used for any data-structure in this library. Note that this iterator is strictly immutable

12.8.2 Constructor & Destructor Documentation

constIterator() [1/4]

template<class dataType>
os::constIterator< dataType >::**constIterator** (
const **smart_ptr**< **nodeFrame**< dataType > > *node*,
const **iteratorBase**< dataType > & *src*) [inline], [private]

Construct with a frame and source.

Uses a reference to both a frame and an **iteratorBase** (p. 228) to construct an iterator which can be generally used.

constIterator() [2/4]

template<class dataType>
os::constIterator< dataType >::**constIterator** ( ) [inline]

Default constructor.

Constructs an empty iterator.

constIterator() [3/4]

template<class dataType>
os::constIterator< dataType >::**constIterator** (
const **constIterator**< dataType > & *cpy*) [inline]

Copy constructor.

Copies an **constIterator** (p. 167) into this iterator. This copy has a minimal performance penalty and retains the deletion protection of the original iterator.

Parameters

in	*cpy*	Immutable iterator

constIterator() [4/4]

template<class dataType>
os::constIterator< dataType >::**constIterator** (
const **iterator**< dataType > & *cpy*) [inline]

Copy constructor.

Copies an iterator into this **constIterator** (p. 167). This copy has a minimal performance penalty, and adds new data protection to the iterator data.

Parameters

in	*cpy*	Immutable iterator

~constIterator()

template<class dataType>
virtual **os::constIterator**< dataType >::~**constIterator** ( ) [inline], [virtual]

Virtual destructor.

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

12.8.3 Member Function Documentation

access()

template<class dataType>
constIterator<dataType> **os::constIterator**< dataType >::access (
long *offset*) const throw **descriptiveException**) [inline]

Access iterator by index.

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

Parameters

in	*offset*	Value to offset by

Returns

Offset iterator, immutable

compare() [1/2]

template<class dataType>
int **os::constIterator**< dataType >::compare (
const **constIterator**< dataType > & *cmp*) const [inline]

Compares two const iterators Uses the comparison definition of the node in an iterator to compare two iterators.

Returns

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

compare() [2/2]

template<class dataType>
COMPARE_OPERATORS int **os::constIterator**< dataType >::compare (
const **iterator**< dataType > & *cmp*) const

Compares a **constIterator** (p. 167) and an iterator Uses the comparison definition of the node in an iterator to compare two iterators.

Returns

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

constAccess()

`template<class dataType>`
constIterator`<dataType>` **os::constIterator**`< dataType >::constAccess (`
`long` *offset* `) const throw` **descriptiveException**`)   [inline]`

Access iterator by index.

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

Parameters

in	*offset*	Value to offset by

Returns

Offset iterator, immutable

constGet()

`template<class dataType>`
`const` **smart_ptr**`<dataType>` **os::constIterator**`< dataType >::constGet ( ) const throw` **descriptiveException**`)   [inline]`

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

decrement()

`template<class dataType>`
`const` **constIterator**`<dataType>&` **os::constIterator**`< dataType >::decrement (`
`long` *offset* `=` *1* `) const  [inline]`

Decrement this iterator Decrements this iterator by some value, by default, 1. Note that if decrementing by more than one, the node in question must support random access.

Parameters

in	*offset*	Value to offset by

Returns

This iterator decremented

get()

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

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

increment()

`template<class dataType>`
`const` **constIterator**`<dataType>&` **os::constIterator**`< dataType >::increment (`
`long` *offset* `= 1 ) const  [inline]`

Increment this iterator Increments this iterator by some value, by default, 1. Note that if incrementing by more than one, the node in question must support random access.

Parameters

in	*offset*	Value to offset by

Returns

This iterator incremented

iterable()

`template<class dataType>`
`bool` **os::constIterator**`< dataType >::iterable ( ) const  [inline]`

Returns if the source is iterable.

Returns

source_structure->**iterable()** (p. 173)

operator &()

`template<class dataType>`
`const` **smart_ptr**`<dataType>` **os::constIterator**`< dataType >::operator & ( ) const throw` **descriptiveException**`)`
`[inline]`

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

operator bool()

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

Boolean conversion.

Returns

currentNode

operator size_t()

```
template<class dataType>
os::constIterator< dataType >::operator size_t ( ) const  [inline]
```

Casts to size_t for hash functions Uses the size definition of the current node.

Returns

hash value

operator"!()

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

Inverted boolean conversion.

Returns

!currentNode

operator∗()

```
template<class dataType>
const dataType& os::constIterator< dataType >::operator* ( ) const throw descriptiveException)   [inline]
```

De-reference.

Returns a const reference to the contained object, the reference cannot be modified.

Returns

Contained object

operator+() [1/3]

```
template<class dataType>
constIterator<dataType> os::constIterator< dataType >::operator+ (
              size_t offset ) const throw descriptiveException)   [inline]
```

operator+() [2/3]

template<class dataType>
constIterator<dataType> **os::constIterator**< dataType >::operator+ (
long *offset*) const throw **descriptiveException**) [inline]

operator+() [3/3]

template<class dataType>
constIterator<dataType> **os::constIterator**< dataType >::operator+ (
int *offset*) const throw **descriptiveException**) [inline]

operator++() [1/2]

template<class dataType>
constIterator<dataType> **os::constIterator**< dataType >::operator++ (
int *param*) const throw **descriptiveException**) [inline]

operator++() [2/2]

template<class dataType>
const **constIterator**<dataType>& **os::constIterator**< dataType >::operator++ ( ) const throw **descriptiveException**)
[inline]

operator+=() [1/3]

template<class dataType>
const **constIterator**<dataType>& **os::constIterator**< dataType >::operator+= (
size_t *offset*) const [inline]

operator+=() [2/3]

template<class dataType>
const **constIterator**<dataType>& **os::constIterator**< dataType >::operator+= (
long *offset*) const [inline]

operator+=() [3/3]

template<class dataType>
const **constIterator**<dataType>& **os::constIterator**< dataType >::operator+= (
int *offset*) const [inline]

operator-() [1/3]

template<class dataType>
constIterator<dataType> **os::constIterator**< dataType >::operator- (
size_t *offset*) const throw **descriptiveException**) [inline]

operator-() [2/3]

```
template<class dataType>
constIterator<dataType> os::constIterator< dataType >::operator- (
            long offset ) const throw descriptiveException)   [inline]
```

operator-() [3/3]

```
template<class dataType>
constIterator<dataType> os::constIterator< dataType >::operator- (
            int offset ) const throw descriptiveException)   [inline]
```

operator--() [1/2]

```
template<class dataType>
constIterator<dataType> os::constIterator< dataType >::operator-- (
            int param ) const throw descriptiveException)   [inline]
```

operator--() [2/2]

```
template<class dataType>
const constIterator<dataType>& os::constIterator< dataType >::operator-- ( ) const throw descriptiveException)
[inline]
```

operator-=() [1/3]

```
template<class dataType>
const constIterator<dataType>& os::constIterator< dataType >::operator-= (
            size_t offset ) const  [inline]
```

operator-=() [2/3]

```
template<class dataType>
const constIterator<dataType>& os::constIterator< dataType >::operator-= (
            long offset ) const  [inline]
```

operator-=() [3/3]

```
template<class dataType>
const constIterator<dataType>& os::constIterator< dataType >::operator-= (
            int offset ) const  [inline]
```

operator->()

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

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

operator[]() [1/3]

```
template<class dataType>
const dataType& os::constIterator< dataType >::operator[] (
            size_t offset ) const throw descriptiveException)   [inline]
```

operator[]() [2/3]

```
template<class dataType>
const dataType& os::constIterator< dataType >::operator[] (
            long offset ) const throw descriptiveException)   [inline]
```

operator[]() [3/3]

```
template<class dataType>
const dataType& os::constIterator< dataType >::operator[] (
            int offset ) const throw descriptiveException)   [inline]
```

randomAccess()

```
template<class dataType>
bool os::constIterator< dataType >::randomAccess ( ) const  [inline]
```

Returns if the source can be accessed randomly.

Returns

source_structure->**randomAccess()** (p. 177)

12.8.4 Friends And Related Function Documentation

iterator< dataType >

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

Two iterator types are friends.

iteratorBase< dataType >

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

iteratorBase (p. 228) must have access to constructor

12.8.5 Member Data Documentation

currentNode

`template<class dataType>`
smart_ptr`<`**nodeFrame**`<dataType> >` **os::constIterator**`< dataType >::currentNode [mutable], [private]`

Pointer to the current node.

source_structure

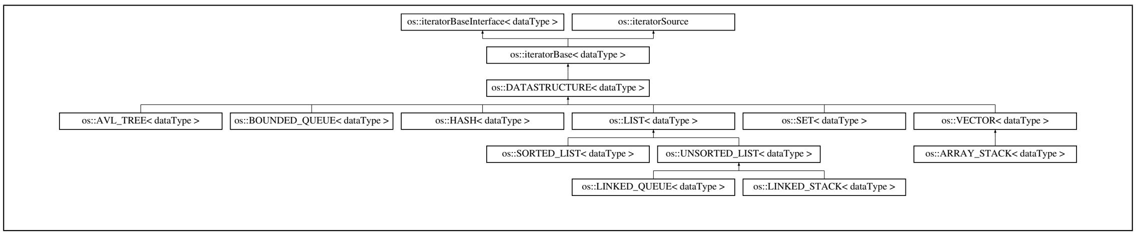
`template<class dataType>`
`const` **iteratorBase**`<dataType>*` **os::constIterator**`< dataType >::source_structure [private]`

Pointer to the iterator source.

12.9 os::DATASTRUCTURE< dataType > Class Template Reference

Object node definition.

Inheritance diagram for os::DATASTRUCTURE< dataType >:



Public Member Functions

- **DATASTRUCTURE** ()
  Default constructor.
- virtual ~**DATASTRUCTURE** ()
  Virtual destructor.
- virtual bool **insert** (const dataType &x)=0
  Insert item into the data-structure.
- bool **insertStructure** (CURRENT_CLASS &x)
  Insert data-structure into the data-structure.
- virtual bool **remove** (const dataType &x)=0
  Remove item from the data-structure.
- virtual bool **find** (const dataType &x) const =0
  Searches for an item.
- virtual dataType & **access** (const dataType &x)=0
  Mutable item access.
- virtual const dataType & **access** (const dataType &x) const =0
  Immutable item access.
- virtual dataType & **at** (size_t i) throw (descriptiveException)

Access the data-structure by index.

- virtual const dataType & **at** (size_t i) const throw (descriptiveException)

  Access the vector by data-structure.
- dataType & **operator[ ]** (size_t i) throw (descriptiveException)

  Access the data-structure by index.
- const dataType & **operator[ ]** (size_t i) const throw (descriptiveException)

  Access the data-structure by index.
- virtual size_t **size** () const =0

  Access size of the structure.

Additional Inherited Members

12.9.1 Detailed Description

template<class dataType>
class os::DATASTRUCTURE< dataType >

Object node definition.

Note that this class is defined in three forms: objectDatastructure pointerDatastructure and rawPointerDatastructure. In short, this class and it's alternate forms specify the details of the **nodeFrame** (p. 274), allowing for holding both objects and pointers to object in the various provided datastructures.

12.9.2 Constructor & Destructor Documentation

DATASTRUCTURE()

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

Default constructor.

~DATASTRUCTURE()

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

Virtual destructor.

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

12.9.3 Member Function Documentation

access() [1/2]

```
template<class dataType >
virtual dataType& os::DATASTRUCTURE< dataType >::access (
            const dataType & x )  [pure virtual]
```

Mutable item access.

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

Parameters

in	*x*	Value to be accessed

Returns

Mutable item equal to x

Implemented in **os::AVL_TREE**< **dataType** > (p. 136), **os::LIST**< **dataType** > (p. 251), **os::BOUNDED_QUEUE**< **dataType** > (p. 150), **os::VECTOR**< **dataType** > (p. 339), **os::SET**< **dataType** > (p. 295), and **os::HASH**< **dataType** > (p. 201).

access() [2/2]

```
template<class dataType >
virtual const dataType& os::DATASTRUCTURE< dataType >::access (
            const dataType & x ) const  [pure virtual]
```

Immutable item access.

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

Parameters

in	*x*	Value to be accessed

Returns

Immutable item equal to x

Implemented in **os::AVL_TREE**< **dataType** > (p. 137), **os::LIST**< **dataType** > (p. 251), **os::BOUNDED_QUEUE**< **dataType** > (p. 150), **os::VECTOR**< **dataType** > (p. 340), **os::SET**< **dataType** > (p. 296), and **os::HASH**< **dataType** > (p. 201).

at() [1/2]

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

Access the data-structure by index.

Parameters

in	*i*	Index of the data-structure

Returns

Reference to ith

Reimplemented in **os::AVL_TREE**< **dataType** > (p. 137), **os::BOUNDED_QUEUE**< **dataType** > (p. 150), **os::VECTOR**< **dataType** > (p. 340), and **os::SET**< **dataType** > (p. 296).

at() [2/2]

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

Access the vector by data-structure.

Parameters

in	*i*	Index of the data-structure

Returns

Immutable reference to ith

Reimplemented in **os::AVL_TREE**< **dataType** > (p. 137), **os::BOUNDED_QUEUE**< **dataType** > (p. 151), **os::VECTOR**< **dataType** > (p. 340), and **os::SET**< **dataType** > (p. 296).

find()

```
template<class dataType >
virtual bool os::DATASTRUCTURE< dataType >::find (
            const dataType & x ) const  [pure virtual]
```

Searches for an item.

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

Parameters

in	*x*	Value to be found

Returns

True if found, else, false

Implemented in **os::AVL_TREE**< **dataType** > (p. 139), **os::LIST**< **dataType** > (p. 251), **os::BOUNDED_QUEUE**< **dataType** > (p. 151), **os::VECTOR**< **dataType** > (p. 341), **os::SET**< **dataType** > (p. 297), and **os::HASH**< **dataType** > (p. 201).

insert()

```
template<class dataType >
virtual bool os::DATASTRUCTURE< dataType >::insert (
            const dataType & x )  [pure virtual]
```

Insert item into the data-structure.

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

Parameters

in	*x*	Value to be inserted

Returns

True if inserted, else, false

Implemented in **os::AVL_TREE**< **dataType** > (p. 141), **os::SORTED_LIST**< **dataType** > (p. 325), **os::UNSORTED_LIST**< **dataType** > (p. 335), **os::SET**< **dataType** > (p. 298), **os::BOUNDED_QUEUE**< **dataType** > (p. 152), **os::VECTOR**< **dataType** > (p. 342), and **os::HASH**< **dataType** > (p. 202).

insertStructure()

```
template<class dataType >
bool os::DATASTRUCTURE< dataType >::insertStructure (
            CURRENT_CLASS & x )  [inline]
```

Insert data-structure into the data-structure.

Note that this function relies off of iteration being defined for a data-structure.

Parameters

in	*x*	Structure tobe inserted

Returns

True if inserted, else, false

operator[]() [1/2]

```
template<class dataType >
dataType& os::DATASTRUCTURE< dataType >::operator[] (
            size_t i ) throw descriptiveException)   [inline]
```

Access the data-structure by index.

Parameters

in	*i*	Index of the data-structure

Returns

Reference to ith element

operator[]() [2/2]

```
template<class dataType >
const dataType& os::DATASTRUCTURE< dataType >::operator[] (
            size_t i ) const throw descriptiveException)   [inline]
```

Access the data-structure by index.

Parameters

in	*i*	Index of the data-structure

Returns

Immutable reference to ith element

remove()

```
template<class dataType >
virtual bool os::DATASTRUCTURE< dataType >::remove (
            const dataType & x )  [pure virtual]
```

Remove item from the data-structure.

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

Parameters

in	*x*	Value to be removed

Returns

True if removed, else, false

Implemented in **os::AVL_TREE**< **dataType** > (p. 142), **os::LIST**< **dataType** > (p. 253), **os::SET**< **dataType** > (p. 299), **os::BOUNDED_QUEUE**< **dataType** > (p. 154), **os::VECTOR**< **dataType** > (p. 342), and **os::HASH**< **dataType** > (p. 203).

size()

```
template<class dataType >
virtual size_t os::DATASTRUCTURE< dataType >::size ( ) const  [pure virtual]
```

Access size of the structure.

Returns

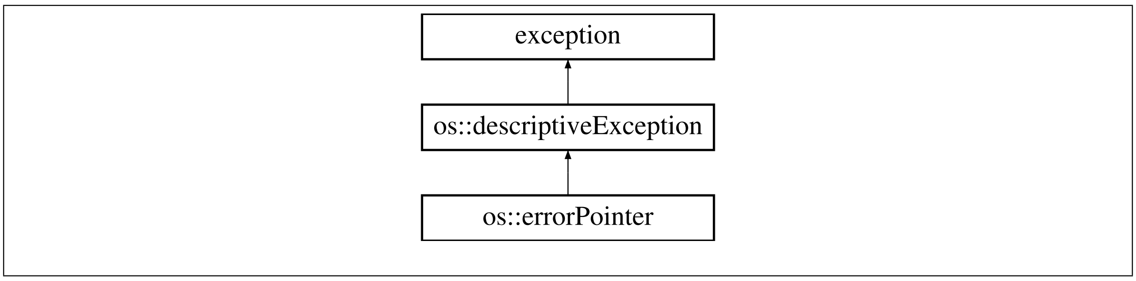
Number of elements in the structure

Implemented in **os::AVL_TREE**< **dataType** > (p. 143), **os::BOUNDED_QUEUE**< **dataType** > (p. 155), **os::VECTOR**< **dataType** > (p. 344), **os::LIST**< **dataType** > (p. 254), **os::SET**< **dataType** > (p. 301), and **os::HASH**< **dataType** > (p. 204).

12.10 os::descriptiveException Class Reference

Basic exception with description.

Inheritance diagram for os::descriptiveException:



Public Member Functions

- **descriptiveException** () throw ()

  Default constructor.
- **descriptiveException** (const **descriptiveException** &de) throw ()

  Copy constructor.
- **descriptiveException** (const char ∗str) throw ()

  Constructor with characters.
- **descriptiveException** (const std::string &str) throw ()

  Constructor with string.
- virtual ∼**descriptiveException** () throw ()

  Destructor Cannot throw exception, as this is an exception and nothing will be there to catch any exception that this throws.
- const char ∗ **what** () const throw ()

  Return exception description.

Private Attributes

- const char ∗ **_desc**

  Character description.
- std::string **_str**

  String description.

12.10.1 Detailed Description

Basic exception with description.

Allows for simple exception with either a string of constant character pointer.

12.10.2 Constructor & Destructor Documentation

descriptiveException() [1/4]

`os::descriptiveException::descriptiveException ( ) throw ) [inline]`

Default constructor.

descriptiveException() [2/4]

`os::descriptiveException::descriptiveException ( const descriptiveException & de ) throw ) [inline]`

Copy constructor.

descriptiveException() [3/4]

`os::descriptiveException::descriptiveException ( const char * str ) throw ) [inline]`

Constructor with characters.

Parameters

str	[in] Description

descriptiveException() [4/4]

`os::descriptiveException::descriptiveException ( const std::string & str ) throw ) [inline]`

Constructor with string.

Parameters

str	[in] Description

~descriptiveException()

`virtual os::descriptiveException::~descriptiveException ( ) throw ) [inline], [virtual]`

Destructor Cannot throw exception, as this is an exception and nothing will be there to catch any exception that this throws.

12.10.3 Member Function Documentation

what()

`const char* os::descriptiveException::what ( ) const throw ) [inline]`

Return exception description.

Returns

Description

12.10.4 Member Data Documentation

_desc

`const char* os::descriptiveException::_desc [private]`

Character description.

_str

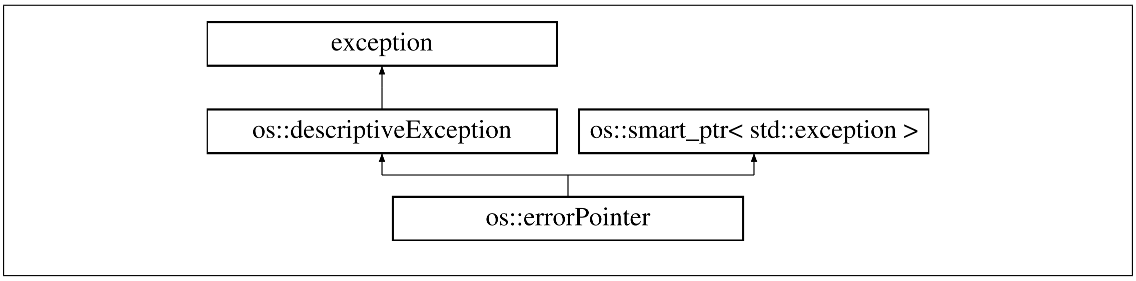
`std::string os::descriptiveException::_str [private]`

String description.

12.11 os::errorPointer Class Reference

Error pointer class.

Inheritance diagram for os::errorPointer:



Public Member Functions

- **errorPointer** () throw ()

  Default constructor.
- **errorPointer** (const **errorPointer** &sp) throw ()

  Copy constructor.
- **errorPointer** (const std::exception *rp, **smart_pointer_type** typ=**raw_type**) throw ()

  Standard constructor.
- **errorPointer** (const std::exception *rp, const **void_rec** destructor) throw ()

  Dynamic deletion constructor.
- virtual ~**errorPointer** () throw ()

  Virtual destructor.
- const char * **what** () const final throw ()

  descriptiveException (p. *185) overload*

12.11.1 Detailed Description

Error pointer class.

Uses the **descriptiveException** (p. 185) interface to allow a pointer to an exception to be handled like an **descriptiveException** (p. 185).

12.11.2 Constructor & Destructor Documentation

errorPointer() [1/4]

```
os::errorPointer::errorPointer ( ) throw )   [inline]
```

Default constructor.

Constructs an **os::smart_ptr** (p. 311) of type **os::null_type** (p. 112). All private data is set to 0 or NULL.

errorPointer() [2/4]

```
os::errorPointer::errorPointer (
            const errorPointer & sp ) throw )   [inline]
```

Copy constructor.

Constructs an **os::errorPointer** (p. 187) from an existing **os::errorPointer** (p. 187).

Parameters

in,out	*sp*	Reference to data being copied

errorPointer() [3/4]

```
os::errorPointer::errorPointer (
            const std::exception * rp,
            smart_pointer_type typ = raw_type ) throw )   [inline]
```

Standard constructor.

Constructs an **os::errorPointer** (p. 187) from a raw pointer and a type. This is the most commonly used **os::errorPointer** (p. 187) constructor, other than the copy constructor.

Parameters

in	*rp*	Raw pointer object is managing
in	*typ*	Defines reference count behaviour

errorPointer() [4/4]

```
os::errorPointer::errorPointer (
            const std::exception * rp,
            const void_rec destructor ) throw )   [inline]
```

Dynamic deletion constructor.

Constructs an **os::errorPointer** (p. 187) from a raw pointer and a destruction function.

Parameters

in	*rp*	Raw pointer object is managing
in	*destructor*	Defines the function to be executed on destroy

~errorPointer()

`virtual os::errorPointer::~errorPointer ( ) throw )   [inline], [virtual]`

Virtual destructor.

Calls **os::smart_ptr**<**dataType**>**::teardown()** (p. 323) before destroying the object.

12.11.3 Member Function Documentation

what()

`const char* os::errorPointer::what ( ) const throw )   [inline], [final]`

descriptiveException (p. 185) overload

Outputs the error or the exception pointer held in this class, if the pointer is NULL, a default warning will be returned.

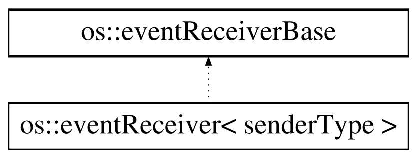
Returns

character pointer to the error description

12.12 os::eventReceiver< senderType > Class Template Reference

Class which enables event receiving.

Inheritance diagram for os::eventReceiver< senderType >:



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 **priv_receiveEvent** (**eventSenderBase** *src)

    Receive event notification.

Friends

- template<typename receiverType >
  class **eventSender**

12.12.1 Detailed Description

template<class senderType>
class os::eventReceiver< senderType >

Class which enables event receiving.

Each receiver contains a list of senders. When the receiver is destroyed, it removes itself from all senders to which it is registered.

12.12.2 Constructor & Destructor Documentation

eventReceiver()

`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.

~eventReceiver()

`template<class senderType >`
virtual **os::eventReceiver**< senderType >::~**eventReceiver** ( ) [virtual]

Virtual destructor.

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

12.12.3 Member Function Documentation

priv_receiveEvent()

`template<class senderType >`
void **os::eventReceiver**< senderType >::priv_receiveEvent (
eventSenderBase * *src*) [inline], [private]

Receive event notification.

Casts a void pointer to the type of the sender.

Parameters

src	The source of the event

Returns

void

pushSender()

```
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. 193).

Parameters

ptr	Sender to be added to the set

Returns

void

receiveEvent()

```
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

src	The source of the event

Returns

void

removeSender()

```
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

ptr	Sender to be removed to the set

Returns

void

12.12.4 Friends And Related Function Documentation

eventSender

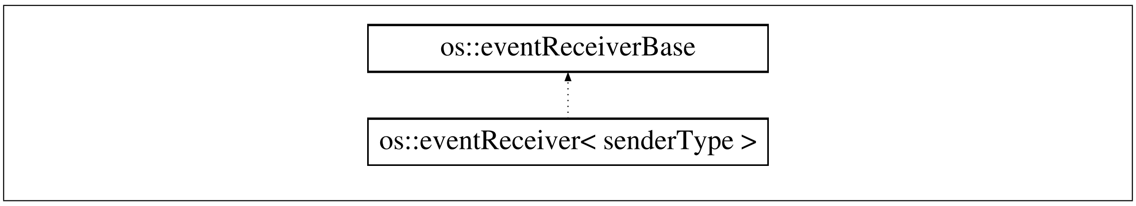
```
template<class senderType >
template<typename receiverType >
friend class eventSender [friend]
```

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.

12.13 os::eventReceiverBase Class Reference

Base class for receiving events This class is inherited by the generalized receiver class.

Inheritance diagram for os::eventReceiverBase:



Private Attributes

- **threadLock _lock**

  Trigger lock Locks an event while it is being triggered.
- rawPointerAVLTree< **eventSenderBase** > **senders**

  List of sender.

12.13.1 Detailed Description

Base class for receiving events This class is inherited by the generalized receiver class.

12.13.2 Member Data Documentation

_lock

threadLock `os::eventReceiverBase::_lock [private]`

Trigger lock Locks an event while it is being triggered.

senders

`rawPointerAVLTree`<**eventSenderBase**> `os::eventReceiverBase::senders [private]`

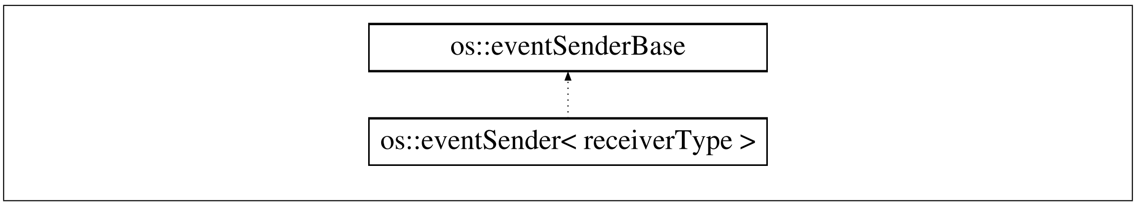
List of sender.

When the receiver is destroyed, this list is used to remove itself from all its senders.

12.14 os::eventSender< receiverType > Class Template Reference

Class which enables event sending.

Inheritance diagram for os::eventSender< receiverType >:



Public Member Functions

- **eventSender** ()

  Default constructor.
- virtual ~**eventSender** ()

  Virtual destructor.
- void **pushReceivers** (**smart_ptr**< receiverType > ptr)

  Add a receiver to the list.
- void **removeReceivers** (**smart_ptr**< receiverType > ptr)

  Remove receiver from the receiver list.

Protected Member Functions

- virtual void **sendEvent** (**smart_ptr**< receiverType > ptr)

  Receive event notification.
- virtual void **sendEvent** (**smart_ptr**< receiverType > ptr, unsigned arg, void *data)

  Receive event notification, flexible data.
- void **triggerEvent** ()

  Sends an event to all receivers.
- void **triggerEvent** (unsigned arg, void *data)

  Sends an event to all receivers with data.

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.

12.14.1 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.

12.14.2 Constructor & Destructor Documentation

eventSender()

```
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.

~eventSender()

```
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.

12.14.3 Member Function Documentation

pushReceivers()

```
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. 189).

Parameters

ptr	Receiver to be added to the set

Returns

void

removeReceivers()

```
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

ptr	Receiver to be removed to the set

Returns

void

sendEvent() [1/2]

```
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	*ptr*	The target of the event

Returns

void

sendEvent() [2/2]

```
template<class receiverType >
virtual void os::eventSender< receiverType >::sendEvent (
        smart_ptr< receiverType > ptr,
        unsigned arg,
        void * data )  [inline], [protected], [virtual]
```

Receive event notification, flexible data.

This function can be re-implemented by event senders. This function allows some function other than "receiveEvent" to be sent by the event sender to an event receiver.

Parameters

in	*ptr*	The target of the event
in	*arg*	Event flag
in	*data*	Void pointer to data

Returns

void

triggerEvent() [1/2]

```
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::event↩Sender<receiverType>::sendEvent** (p. 195) function with each receiver as an argument.

Returns

void

triggerEvent() [2/2]

```
template<class receiverType >
void os::eventSender< receiverType >::triggerEvent (
            unsigned arg,
            void * data )  [protected]
```

Sends an event to all receivers with data.

Iterates through the set of receivers and sends an event to each one. This calls the **os::event↩Sender<receiverType>::sendEvent** (p. 195) function with each receiver as an argument. This event trigger passes data arguements.

Parameters

in	*arg*	Event flag
in	*data*	Void pointer to data

Returns

void

12.14.4 Friends And Related Function Documentation

eventReceiver

```
template<class receiverType >
```

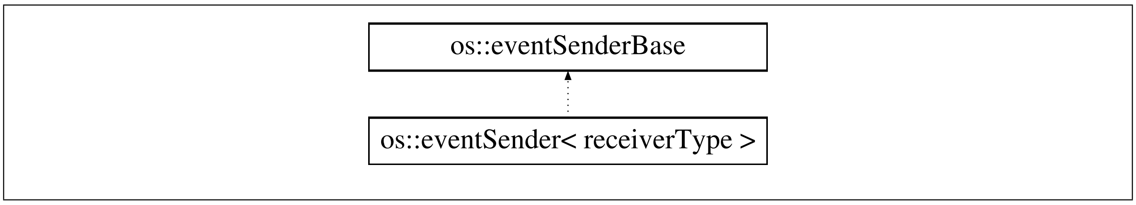
template<typename senderType >
friend class **eventReceiver** [friend]

Friendship with the senderType The receiver must be able to remove itself from the private receivers list inside the event sender.

12.15 os::eventSenderBase Class Reference

Base class for sender events This class is inherited by the generalized sender event class.

Inheritance diagram for os::eventSenderBase:



Private Attributes

- **threadLock _lock**

  Trigger lock Locks an event while it is being triggered.
- rawPointerAVLTree< **eventReceiverBase** > **receivers**

  List of receivers.

12.15.1 Detailed Description

Base class for sender events This class is inherited by the generalized sender event class.

12.15.2 Member Data Documentation

_lock

threadLock os::eventSenderBase::_lock [private]

Trigger lock Locks an event while it is being triggered.

receivers

rawPointerAVLTree<**eventReceiverBase**> os::eventSenderBase::receivers [private]

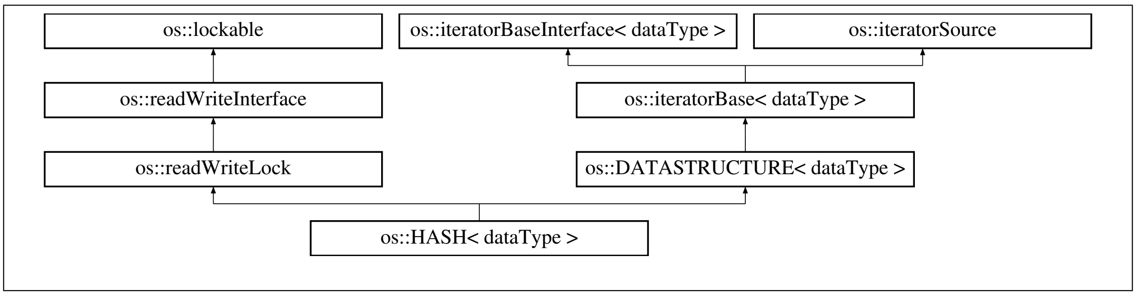
List of receivers.

This list is used to send events to all receivers. When the sender is destroyed, it must remove itself from all its receivers.

12.16 os::HASH< dataType > Class Template Reference

Iterable hash table.

Inheritance diagram for os::HASH< dataType >:



Public Member Functions

- **HASH** ()

  Constructs the hash table Uses the correct comparison function construct the hash table.
- **HASH** (const **HASH**< dataType > &cpy)

  Copy constructor.
- ~**HASH** () final

  Destroys the hash Removes every element from the hash. Depending on the hash format, this may delete everythin in the hash.
- size_t **size** () const final

  Access size of the hash.
- bool **iterable** () const final

  Returns if the relevant datastructure type is iterable.
- bool **randomAccess** () const final

  Returns if the relevant datastructure type can be accessed randomly.
- bool **insert** (const dataType &x) final

  Insert item into the hash.
- bool **remove** (const dataType &x) final

  Remove item from the hash.
- bool **find** (const dataType &x) const final

  Searches for an item in the hash.
- dataType & **access** (const dataType &x) final

  Mutable item access.
- const dataType & **access** (const dataType &x) const final

  Immutable item access.

Static Public Attributes

- static const bool **ITERABLE** = true

  Hashes are iterable.
- static const bool **RANDOM_ACCESS** = false

  Hashes do not allow random access.

Protected Member Functions

- **smart_ptr**< **nodeFrame**< dataType > > **getFirstNode** () final

  Access to first node.
- **smart_ptr**< **nodeFrame**< dataType > > **getLastNode** () final

  Access to last node.
- const **smart_ptr**< **nodeFrame**< dataType > > **getFirstNodeConst** () const final

  Constant access to first node.
- const **smart_ptr**< **nodeFrame**< dataType > > **getLastNodeConst** () const final

  Constant access to last node.
- **smart_ptr**< **nodeFrame**< dataType > > **searchNode** (const **smart_ptr**< dataType > dt) final

  Search for a node.
- const **smart_ptr**< **nodeFrame**< dataType > > **searchNodeConst** (const **smart_ptr**< dataType > dt) const final

  Const search for a node.

Private Attributes

- **simpleHash**< dataType > **_hashTable**

  Hash table This table stores the data inside this particular hash.

Additional Inherited Members

12.16.1 Detailed Description

template<class dataType>
class os::HASH< dataType >

Iterable hash table.

Uses the structure defined by the **os::simpleHash** (p. 302) to define a hash table which is iterable.

12.16.2 Constructor & Destructor Documentation

HASH() [1/2]

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

Constructs the hash table Uses the correct comparison function construct the hash table.

HASH() [2/2]

```
template<class dataType>
os::HASH< dataType >::HASH (
            const HASH< dataType > & cpy )  [inline]
```

Copy constructor.

This constructor builds a hash table from another hash table. Note that this copies by value, not reference.

Parameters

in	*cpy*	Target to be copied

~HASH()

```
template<class dataType>
os::HASH< dataType >::~HASH ( )  [inline], [final]
```

Destroys the hash Removes every element from the hash. Depending on the hash format, this may delete everythin in the hash.

12.16.3 Member Function Documentation

access() [1/2]

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

Mutable item access.

Accesses an element in the hash, O(1). Note that different forms of the hash accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Returns

Mutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 179).

access() [2/2]

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

Immutable item access.

Accesses an element in the hash, O(1). Note that different forms of the hash accept pointers instead of object references, and versions returning pointers will return NULL instead of throwing an exception upon failure.

Returns

Immutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 181).

find()

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

Searches for an item in the hash.

Finds an element in the hash, O(1). Note that different forms of the hash accept pointers instead of object references.

Returns

True if found, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 182).

getFirstNode()

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

Access to first node.

Returns

First node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 230).

getFirstNodeConst()

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

Constant access to first node.

Returns

Immutable first node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 230).

getLastNode()

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

Access to last node.

Returns

Last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 231).

getLastNodeConst()

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

Constant access to last node.

Returns

Immutable last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 231).

insert()

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

Insert item into the hash.

Inserts an item into hash, O(1). Note that different forms of the hash accept pointers instead of object references.

Returns

True

Implements **os::DATASTRUCTURE**< **dataType** > (p. 182).

iterable()

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

Returns if the relevant datastructure type is iterable.

Returns

true

Reimplemented from **os::iteratorSource** (p. 239).

randomAccess()

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

Returns if the relevant datastructure type can be accessed randomly.

Returns

true

Reimplemented from **os::iteratorSource** (p. 240).

remove()

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

Remove item from the hash.

Removes an element in the hash, O(1). Note that different forms of the hash accept pointers instead of object references.

Returns

True if removed, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 184).

searchNode()

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

Search for a node.

Parameters

in	*dt*	Pointer to search for

Returns

Muttable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 232).

searchNodeConst()

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

Const search for a node.

Parameters

in	*dt*	Pointer to search for

Returns

Immutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 232).

size()

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

Access size of the hash.

Returns

Number of elements in the hash

Implements **os::DATASTRUCTURE**< **dataType** > (p. 184).

12.16.4 Member Data Documentation

_hashTable

```
template<class dataType>
simpleHash<dataType> os::HASH< dataType >::_hashTable  [private]
```

Hash table This table stores the data inside this particular hash.

ITERABLE

template<class dataType>
const bool **os::HASH**< dataType >::ITERABLE = true [static]

Hashes are iterable.

RANDOM_ACCESS

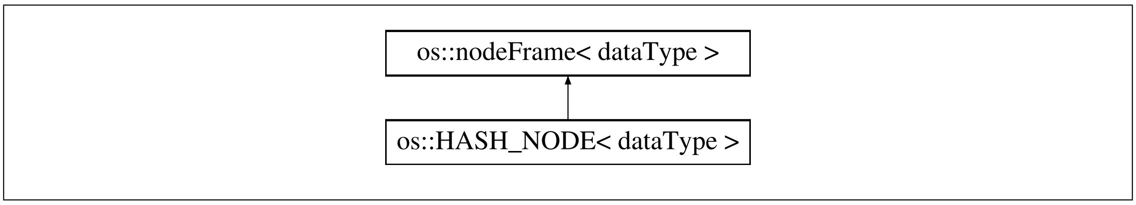
template<class dataType>
const bool **os::HASH**< dataType >::RANDOM_ACCESS = false [static]

Hashes do not allow random access.

12.17 os::HASH_NODE< dataType > Class Template Reference

Hash node.

Inheritance diagram for os::HASH_NODE< dataType >:



Public Member Functions

- ~**HASH_NODE** () final

  Destructor Class is not designed to be inherited from.
- **smart_ptr**< dataType > **get** () final throw ()

  Returns a pointer.
- const **smart_ptr**< dataType > **constGet** () const final throw ()

  Returns a const pointer.
- dataType & **operator*** () final throw (descriptiveException)

  De-reference.
- const dataType & **operator*** () const final throw (descriptiveException)

  De-reference.
- bool **valid** () const final

  Valid data query Checks if the provided data is valid.
- bool **iterable** () const final

  Returns if the node is iterable.
- bool **randomAccess** () const final

  Returns if the node can be accessed randomly.
- **smart_ptr**< **nodeFrame**< dataType > > **getNext** () final throw (descriptiveException)

  Returns the next hash frame.

- const **smart_ptr**< **nodeFrame**< dataType > > **getNextConst** () const final throw (descriptive↩ Exception)

  Returns the next hash frame.
- **smart_ptr**< **nodeFrame**< dataType > > **getPrev** () final throw (descriptiveException)

  Returns the previous hash frame.
- const **smart_ptr**< **nodeFrame**< dataType > > **getPrevConst** () const final throw (descriptive↩ Exception)

  Returns the previous hash frame.
- void **remove** () final throw (descriptiveException)

  Remove this node from the hash.

Static Public Attributes

- static const bool **ITERABLE** = true

  Hash frames are iterable.
- static const bool **RANDOM_ACCESS** = false

  Hash frames do not allow random-access.

Private Member Functions

- **HASH_NODE** (**HASH**< dataType > ∗src, size_t pos)

  *Private constructor This constructor is not designed to be accessed by anything other than the **HASH** (p. 199) class.*

Private Attributes

- size_t **position**

  Current position of the hash iterator.

12.17.1 Detailed Description

template<class dataType>
class os::HASH_NODE< dataType >

Hash node.
Used by the iterator to iterate through a hash.

12.17.2 Constructor & Destructor Documentation

HASH_NODE()

```
template<class dataType >
os::HASH_NODE< dataType >::HASH_NODE (
            HASH< dataType > * src,
            size_t pos ) [inline], [private]
```

Private constructor This constructor is not designed to be accessed by anything other than the **HASH** (p. 199) class.

~HASH_NODE()

template<class dataType >
os::HASH_NODE< dataType >::~**HASH_NODE** ( ) [inline], [final]

Destructor Class is not designed to be inherited from.

12.17.3 Member Function Documentation

constGet()

template<class dataType >
const **smart_ptr**<dataType> **os::HASH_NODE**< dataType >::constGet ( ) const throw) [inline], [final], [virtual]

Returns a const pointer.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

Implements **os::nodeFrame**< **dataType** > (p. 277).

get()

template<class dataType >
smart_ptr<dataType> **os::HASH_NODE**< dataType >::get ( ) throw) [inline], [final], [virtual]

Returns a pointer.

Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

Implements **os::nodeFrame**< **dataType** > (p. 277).

getNext()

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

Returns the next hash frame.

Returns

Next node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 277).

getNextConst()

template<class dataType >
const **smart_ptr**<**nodeFrame**<dataType> > **os::HASH_NODE**< dataType >::getNextConst ( ) const throw **descriptive↩ Exception**) [inline], [final], [virtual]

Returns the next hash frame.

Returns

Next node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 278).

getPrev()

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

Returns the previous hash frame.

Returns

Previous node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 278).

getPrevConst()

template<class dataType >
const **smart_ptr**<**nodeFrame**<dataType> > **os::HASH_NODE**< dataType >::getPrevConst ( ) const throw **descriptive←Exception**) [inline], [final], [virtual]

Returns the previous hash frame.

Returns

Previous node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 279).

iterable()

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

Returns if the node is iterable.

Returns

object::ITERABLE

Reimplemented from **os::nodeFrame**< **dataType** > (p. 279).

operator*() [1/2]

template<class dataType >
dataType& **os::HASH_NODE**< dataType >::operator* ( ) throw **descriptiveException**) [inline], [final], [virtual]

De-reference.

Returns a reference to the contained object, the reference can be modified.

Returns

Contained object

Implements **os::nodeFrame**< **dataType** > (p. 280).

operator*() [2/2]

```
template<class dataType >
const dataType& os::HASH_NODE< dataType >::operator* ( ) const throw descriptiveException)   [inline],
[final], [virtual]
```

De-reference.
Returns a const reference to the contained object, the reference cannot be modified.

Returns

Contained object

Implements **os::nodeFrame**< **dataType** > (p. 280).

randomAccess()

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

Returns if the node can be accessed randomly.

Returns

object::RANDOM_ACCESS

Reimplemented from **os::nodeFrame**< **dataType** > (p. 281).

remove()

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

Remove this node from the hash.

Returns

void

Reimplemented from **os::nodeFrame**< **dataType** > (p. 281).

valid()

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

Valid data query Checks if the provided data is valid.

Returns

true if valid, else, false

Reimplemented from **os::nodeFrame**< **dataType** > (p. 282).

12.17.4 Member Data Documentation

ITERABLE

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

Hash frames are iterable.

position

```
template<class dataType >
size_t os::HASH_NODE< dataType >::position  [private]
```

Current position of the hash iterator.

RANDOM_ACCESS

```
template<class dataType >
const bool os::HASH_NODE< dataType >::RANDOM_ACCESS = false  [static]
```

Hash frames do not allow random-access.

12.18 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.

12.18.1 Detailed Description

template<class dataType>
class os::indirectMatrix< dataType >

Indirect matrix.

This matrix class contains an array to pointers of the data type. It can interact with os::matrix<data↩ Type>.

12.18.2 Constructor & Destructor Documentation

indirectMatrix() [1/5]

```
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	*w*	Width of matrix, default 0
in	*h*	Height of matrix, default 0

indirectMatrix() [2/5]

```
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	*m*	Indirect matrix to be copied

indirectMatrix() [3/5]

```
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	*m*	Indirect matrix to be copied

indirectMatrix() [4/5]

```
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	*d*	Data array to be copied
in	*w*	Width of matrix
in	*d*	Height of matrix

indirectMatrix() [5/5]

```
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	*d*	Indirect data array to be copied
in	*w*	Width of matrix
in	*d*	Height of matrix

~indirectMatrix()

```
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.

12.18.3 Member Function Documentation

constGet()

```
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	*w*	X position
in	*h*	Y position

Returns

Constant reference to matrix element pointer

get()

```
template<class dataType>
smart_ptr<dataType>& os::indirectMatrix< dataType >::get (
            uint32_t w,
            uint32_t h ) throw )
```

Return pointer to a matrix element.

Uses a _width and _height position to index an element of the array. This function returns a reference, allowing for changes to be made to the matrix.

Parameters

in	*w*	X position
in	*h*	Y position

Returns

Modifiable reference to matrix element pointer

getArray()

```
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. 216)

getConstArray()

```
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. 216)

height()

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

Return _height of matrix.

Returns

indirectMatrix<**dataType**>**::_height** (p. 216)

operator()()

```
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

operator=() [1/2]

```
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

operator=() [2/2]

```
template<class dataType>
indirectMatrix<dataType>& os::indirectMatrix< dataType >::operator= (
            const indirectMatrix< dataType > & m ) throw )
```

Equality constructor.

Re-constructs the indirect matrix from another indirect matrix. Note that the two matrices do not share the same data array.

Parameters

in	*m*	Reference to matrix being copied

Returns

Reference to self

width()

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

Return _width of matrix.

Returns

indirectMatrix<**dataType**>**::_width** (p. 216)

12.18.4 Friends And Related Function Documentation

matrix< dataType >

`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.

12.18.5 Member Data Documentation

_height

`template<class dataType>`
`uint32_t` **os::indirectMatrix**`< dataType >::_height  [private]`

Height of the matrix.

_width

`template<class dataType>`
`uint32_t` **os::indirectMatrix**`< dataType >::_width  [private]`

Width of the matrix.

data

`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.

12.19 os::iterator< dataType > Class Template Reference

Generalized iterator.

Public Member Functions

- **iterator** ()
  Default constructor.
- **iterator** (const **iterator**< dataType > &cpy)
  Copy constructor.
- virtual ~**iterator** ()
  Virtual destructor.
- bool **operator!** () const throw ()
  Inverted boolean conversion.
- **operator bool** () const throw ()
  Boolean conversion.
- **smart_ptr**< dataType > **get** () throw (descriptiveException)
  Member access.
- const **smart_ptr**< dataType > **constGet** () const throw (descriptiveException)
  Member access.
- const **smart_ptr**< dataType > **get** () const throw (descriptiveException)
  Member access.
- **smart_ptr**< dataType > **operator &** () throw (descriptiveException)
  Member access.
- const **smart_ptr**< dataType > **operator &** () const throw (descriptiveException)
  Member access.
- **smart_ptr**< dataType > **operator->** () throw (descriptiveException)
  Member access.
- const **smart_ptr**< dataType > **operator->** () const throw (descriptiveException)
  Member access.
- dataType & **operator*** () throw (descriptiveException)
  De-reference.
- const dataType & **operator*** () const throw (descriptiveException)
  De-reference.
- void **remove** ()
  Remove node from the list.

- **iterator**< dataType > **access** (long offset) const throw (descriptiveException)

    Access iterator by index.
- dataType & **operator[ ]** (size_t offset) throw (descriptiveException)
- dataType & **operator[ ]** (long offset) throw (descriptiveException)
- dataType & **operator[ ]** (int offset) throw (descriptiveException)
- const dataType & **operator[ ]** (size_t offset) const throw (descriptiveException)
- const dataType & **operator[ ]** (long offset) const throw (descriptiveException)
- const dataType & **operator[ ]** (int offset) const throw (descriptiveException)
- **iterator**< dataType > **operator+** (size_t offset) const throw (descriptiveException)
- **iterator**< dataType > **operator-** (size_t offset) const throw (descriptiveException)
- **iterator**< dataType > **operator+** (long offset) const throw (descriptiveException)
- **iterator**< dataType > **operator-** (long offset) const throw (descriptiveException)
- **iterator**< dataType > **operator+** (int offset) const throw (descriptiveException)
- **iterator**< dataType > **operator-** (int offset) const throw (descriptiveException)
- const **iterator**< dataType > & **increment** (long offset=1) const

    Increment this iterator Increments this iterator by some value, by default, 1. Note that if incrementing by more than one, the node in question must support random access.
- const **iterator**< dataType > & **decrement** (long offset=1) const

    Decrement this iterator Decrements this iterator by some value, by default, 1. Note that if decrementing by more than one, the node in question must support random access.
- const **iterator**< dataType > & **operator+=** (size_t offset) const
- const **iterator**< dataType > & **operator-=** (size_t offset) const
- const **iterator**< dataType > & **operator+=** (long offset) const
- const **iterator**< dataType > & **operator-=** (long offset) const
- const **iterator**< dataType > & **operator+=** (int offset) const
- const **iterator**< dataType > & **operator-=** (int offset) const
- **iterator**< dataType > **operator++** (int param) throw (descriptiveException)
- **iterator**< dataType > **operator--** (int param) throw (descriptiveException)
- const **iterator**< dataType > & **operator++** () const throw (descriptiveException)
- const **iterator**< dataType > & **operator--** () const throw (descriptiveException)
- bool **iterable** () const

    Returns if the source is iterable.
- bool **randomAccess** () const

    Returns if the source can be accessed randomly.
- int **compare** (const **iterator**< dataType > &cmp) const

    Compares two iterators Uses the comparison definition of the node in an iterator to compare two iterators.
- **operator size_t** () const

    Casts to size_t for hash functions Uses the size definition of the current node.
- COMPARE_OPERATORS int **compare** (const **constIterator**< dataType > &cmp) const

    *Compares an iterator and a **constIterator** (p. 167) Uses the comparison definition of the node in an iterator to compare two iterators.*

Private Member Functions

- **iterator** (const **smart_ptr**< **nodeFrame**< dataType > > node, const **iteratorBase**< dataType > &src)

    Construct with a frame and source.

Private Attributes

- const **iteratorBase**< dataType > * **source_structure**

    Pointer to the iterator source.

- **smart_ptr**< **nodeFrame**< dataType > > **currentNode**

    Pointer to the current node.

Friends

- class **iteratorBaseInterface**< **dataType** >

    iteratorBaseInterface *(p. 233) must have access to constructor*

- class **iteratorBase**< **dataType** >

    iteratorBase *(p. 228) must have access to constructor*

- class **constIterator**< **dataType** >

    Two iterator types are friends.

12.19.1 Detailed Description

template<class dataType>
class os::iterator< dataType >

Generalized iterator.

Since this iterator wraps the node frame, it can be generally used for any data-structure in this library.

12.19.2 Constructor & Destructor Documentation

iterator() [1/3]

```
template<class dataType>
os::iterator< dataType >::iterator (
            const smart_ptr< nodeFrame< dataType > > node,
            const iteratorBase< dataType > & src ) [inline], [private]
```

Construct with a frame and source.

Uses a reference to both a frame and an **iteratorBase** (p. 228) to construct an iterator which can be generally used.

iterator() [2/3]

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

Default constructor.

Constructs an empty iterator.

iterator() [3/3]

```
template<class dataType>
os::iterator< dataType >::iterator (
              const iterator< dataType > & cpy )  [inline]
```

Copy constructor.

Copies an iterator into this iterator. This copy has a minimal performance penalty and retains the deletion protection of the original iterator.

Parameters

in	*cpy*	Immutable iterator

~iterator()

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

Virtual destructor.

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

12.19.3 Member Function Documentation

access()

```
template<class dataType>
iterator<dataType> os::iterator< dataType >::access (
              long offset ) const throw descriptiveException)   [inline]
```

Access iterator by index.

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

Returns

Offset iterator, mutable

compare() [1/2]

```
template<class dataType>
int os::iterator< dataType >::compare (
              const iterator< dataType > & cmp ) const  [inline]
```

Compares two iterators Uses the comparison definition of the node in an iterator to compare two iterators.

Returns

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

compare() [2/2]

```
template<class dataType>
COMPARE_OPERATORS int os::iterator< dataType >::compare (
            const constIterator< dataType > & cmp ) const
```

Compares an iterator and a **constIterator** (p. 167) Uses the comparison definition of the node in an iterator to compare two iterators.

Returns

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

constGet()

```
template<class dataType>
const smart_ptr<dataType> os::iterator< dataType >::constGet ( ) const throw descriptiveException)   [inline]
```

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

decrement()

```
template<class dataType>
const iterator<dataType>& os::iterator< dataType >::decrement (
            long offset = 1 ) const  [inline]
```

Decrement this iterator Decrements this iterator by some value, by default, 1. Note that if decrementing by more than one, the node in question must support random access.

Parameters

in	*offset*	Value to offset by

Returns

This iterator decremented

get() [1/2]

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

Member access.

Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

get() [2/2]

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

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

increment()

```
template<class dataType>
const iterator<dataType>& os::iterator< dataType >::increment (
              long offset = 1 ) const  [inline]
```

Increment this iterator Increments this iterator by some value, by default, 1. Note that if incrementing by more than one, the node in question must support random access.

Parameters

in	*offset*	Value to offset by

Returns

This iterator incremented

iterable()

```
template<class dataType>
bool os::iterator< dataType >::iterable ( ) const  [inline]
```

Returns if the source is iterable.

Returns

source_structure->**iterable()** (p. 222)

operator &() [1/2]

```
template<class dataType>
smart_ptr<dataType> os::iterator< dataType >::operator & ( ) throw descriptiveException)   [inline]
```

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

operator &() [2/2]

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

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

operator bool()

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

Boolean conversion.

Returns

currentNode

operator size_t()

```
template<class dataType>
os::iterator< dataType >::operator size_t ( ) const  [inline]
```

Casts to size_t for hash functions Uses the size definition of the current node.

Returns

hash value

operator"!()

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

Inverted boolean conversion.

Returns

!currentNode

operator*() [1/2]

```
template<class dataType>
dataType& os::iterator< dataType >::operator* ( ) throw descriptiveException)   [inline]
```

De-reference.

Returns a reference to the contained object, the reference can be modified.

Returns

Contained object

operator*() [2/2]

```
template<class dataType>
const dataType& os::iterator< dataType >::operator* ( ) const throw descriptiveException)   [inline]
```

De-reference.

Returns a const reference to the contained object, the reference cannot be modified.

Returns

Contained object

operator+() [1/3]

```
template<class dataType>
iterator<dataType> os::iterator< dataType >::operator+ (
            size_t offset ) const throw descriptiveException)   [inline]
```

operator+() [2/3]

```
template<class dataType>
iterator<dataType> os::iterator< dataType >::operator+ (
            long offset ) const throw descriptiveException)   [inline]
```

operator+() [3/3]

```
template<class dataType>
iterator<dataType> os::iterator< dataType >::operator+ (
            int offset ) const throw descriptiveException)   [inline]
```

operator++() [1/2]

```
template<class dataType>
iterator<dataType> os::iterator< dataType >::operator++ (
            int param ) throw descriptiveException)   [inline]
```

operator++() [2/2]

```
template<class dataType>
const iterator<dataType>& os::iterator< dataType >::operator++ ( ) const throw descriptiveException)   [inline]
```

operator+=() [1/3]

```
template<class dataType>
const iterator<dataType>& os::iterator< dataType >::operator+= (
            size_t offset ) const  [inline]
```

operator+=() [2/3]

```
template<class dataType>
const iterator<dataType>& os::iterator< dataType >::operator+= (
            long offset ) const  [inline]
```

operator+=() [3/3]

template<class dataType>
const **iterator**<dataType>& **os::iterator**< dataType >::operator+= (
int *offset*) const [inline]

operator-() [1/3]

template<class dataType>
iterator<dataType> **os::iterator**< dataType >::operator- (
size_t *offset*) const throw **descriptiveException**) [inline]

operator-() [2/3]

template<class dataType>
iterator<dataType> **os::iterator**< dataType >::operator- (
long *offset*) const throw **descriptiveException**) [inline]

operator-() [3/3]

template<class dataType>
iterator<dataType> **os::iterator**< dataType >::operator- (
int *offset*) const throw **descriptiveException**) [inline]

operator--() [1/2]

template<class dataType>
iterator<dataType> **os::iterator**< dataType >::operator-- (
int *param*) throw **descriptiveException**) [inline]

operator--() [2/2]

template<class dataType>
const **iterator**<dataType>& **os::iterator**< dataType >::operator-- ( ) const throw **descriptiveException**) [inline]

operator-=() [1/3]

template<class dataType>
const **iterator**<dataType>& **os::iterator**< dataType >::operator-= (
size_t *offset*) const [inline]

operator-=() [2/3]

template<class dataType>
const **iterator**<dataType>& **os::iterator**< dataType >::operator-= (
long *offset*) const [inline]

operator-=() [3/3]

```
template<class dataType>
const iterator<dataType>& os::iterator< dataType >::operator-= (
            int offset ) const  [inline]
```

operator->() [1/2]

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

Member access.

Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

operator->() [2/2]

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

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

operator[]() [1/6]

```
template<class dataType>
dataType& os::iterator< dataType >::operator[] (
            size_t offset ) throw descriptiveException)   [inline]
```

operator[]() [2/6]

```
template<class dataType>
dataType& os::iterator< dataType >::operator[] (
            long offset ) throw descriptiveException)   [inline]
```

operator[]() [3/6]

```
template<class dataType>
dataType& os::iterator< dataType >::operator[] (
            int offset ) throw descriptiveException)   [inline]
```

operator[]() [4/6]

```
template<class dataType>
const dataType& os::iterator< dataType >::operator[] (
            size_t offset ) const throw descriptiveException)   [inline]
```

operator[]() [5/6]

```
template<class dataType>
const dataType& os::iterator< dataType >::operator[] (
              long offset ) const throw descriptiveException)   [inline]
```

operator[]() [6/6]

```
template<class dataType>
const dataType& os::iterator< dataType >::operator[] (
              int offset ) const throw descriptiveException)   [inline]
```

randomAccess()

```
template<class dataType>
bool os::iterator< dataType >::randomAccess ( ) const  [inline]
```

Returns if the source can be accessed randomly.

Returns

source_structure->**randomAccess()** (p. 227)

remove()

```
template<class dataType>
void os::iterator< dataType >::remove ( )  [inline]
```

Remove node from the list.

Returns

void

12.19.4 Friends And Related Function Documentation

constIterator< dataType >

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

Two iterator types are friends.

iteratorBase< dataType >

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

iteratorBase (p. 228) must have access to constructor

iteratorBaseInterface< dataType >

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

iteratorBaseInterface (p. 233) must have access to constructor

12.19.5 Member Data Documentation

currentNode

template<class dataType>
smart_ptr<**nodeFrame**<dataType> > **os::iterator**< dataType >::currentNode [mutable], [private]

Pointer to the current node.

source_structure

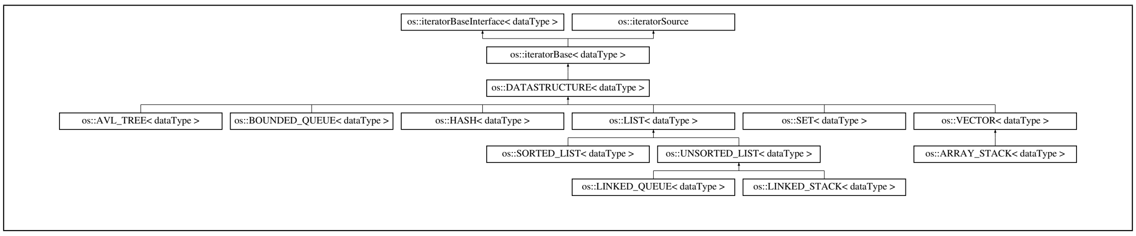
template<class dataType>
const **iteratorBase**<dataType>* **os::iterator**< dataType >::source_structure [private]

Pointer to the iterator source.

12.20 os::iteratorBase< dataType > Class Template Reference

Iterator source.

Inheritance diagram for os::iteratorBase< dataType >:



Public Member Functions

- **iterator**< dataType > **first** () final

  Get first iterator.
- **iterator**< dataType > **last** () final

  Get last iterator.
- **constIterator**< dataType > **constFirst** () const final

  Get first iterator.
- **constIterator**< dataType > **constLast** () const final

  Get last iterator.
- **iterator**< dataType > **search** (const dataType &dt) final

  Search for a node.
- **constIterator**< dataType > **constSearch** (const dataType &dt) const final

  Serach for a node.

Protected Member Functions

- virtual **smart_ptr**< **nodeFrame**< dataType > > **getFirstNode** ()=0

  Access to first node.
- virtual **smart_ptr**< **nodeFrame**< dataType > > **getLastNode** ()=0

  Access to last node.
- virtual const **smart_ptr**< **nodeFrame**< dataType > > **getFirstNodeConst** () const =0

  Constant access to first node.
- virtual const **smart_ptr**< **nodeFrame**< dataType > > **getLastNodeConst** () const =0

  Constant access to last node.
- virtual **smart_ptr**< **nodeFrame**< dataType > > **searchNode** (const **smart_ptr**< dataType > dt)=0

  Search for a node.
- virtual const **smart_ptr**< **nodeFrame**< dataType > > **searchNodeConst** (const **smart_ptr**< dataType > dt) const =0

  Const search for a node.

Friends

- class **iterator**< **dataType** >

  Iterator must have access to counter.
- class **constIterator**< **dataType** >

  constIterator (p. *167) must have access to counter*

Additional Inherited Members

12.20.1 Detailed Description

template<class dataType>
class os::iteratorBase< dataType >

Iterator source.

Defines a class template for an object which generates an iterator.

12.20.2 Member Function Documentation

constFirst()

template<class dataType>
constIterator<dataType> **os::iteratorBase**< dataType >::constFirst ( ) const [final], [virtual]

Get first iterator.

Returns

Immutable first element

Implements **os::iteratorBaseInterface**< **dataType** > (p. 234).

constLast()

`template<class dataType>`
constIterator`<dataType>` **os::iteratorBase**`< dataType >::constLast ( ) const [final], [virtual]`

Get last iterator.

Returns

Immutable last element

Implements **os::iteratorBaseInterface**< **dataType** > (p. 235).

constSearch()

`template<class dataType>`
constIterator`<dataType>` **os::iteratorBase**`< dataType >::constSearch (`
`const dataType & *dt* ) const [final], [virtual]`

Serach for a node.

Parameters

in	*dt*	Node to search for

Returns

Immutable element

Implements **os::iteratorBaseInterface**< **dataType** > (p. 235).

first()

`template<class dataType>`
iterator`<dataType>` **os::iteratorBase**`< dataType >::first ( ) [final], [virtual]`

Get first iterator.

Returns

Mutable first element

Implements **os::iteratorBaseInterface**< **dataType** > (p. 236).

getFirstNode()

`template<class dataType>`
`virtual` **smart_ptr**`<`**nodeFrame**`<dataType> >` **os::iteratorBase**`< dataType >::getFirstNode ( ) [protected], [pure virtual]`

Access to first node.

Returns

First node in the structure

Implemented in **os::AVL_TREE**< **dataType** > (p. 140), **os::LIST**< **dataType** > (p. 252), **os::SET**< **dataType** > (p. 297), **os::VECTOR**< **dataType** > (p. 341), **os::BOUNDED_QUEUE**< **dataType** > (p. 152), and **os::HASH**< **dataType** > (p. 202).

getFirstNodeConst()

```
template<class dataType>
virtual const smart_ptr<nodeFrame<dataType> > os::iteratorBase< dataType >::getFirstNodeConst ( ) const
[protected], [pure virtual]
```

Constant access to first node.

Returns

Immutable first node in the structure

Implemented in **os::AVL_TREE**< **dataType** > (p. 140), **os::LIST**< **dataType** > (p. 252), **os::SET**< **dataType** > (p. 297), **os::VECTOR**< **dataType** > (p. 341), **os::BOUNDED_QUEUE**< **dataType** > (p. 152), and **os::HASH**< **dataType** > (p. 202).

getLastNode()

```
template<class dataType>
virtual smart_ptr<nodeFrame<dataType> > os::iteratorBase< dataType >::getLastNode ( ) [protected], [pure
virtual]
```

Access to last node.

Returns

Last node in the structure

Implemented in **os::AVL_TREE**< **dataType** > (p. 141), **os::LIST**< **dataType** > (p. 252), **os::SET**< **dataType** > (p. 297), **os::VECTOR**< **dataType** > (p. 341), **os::BOUNDED_QUEUE**< **dataType** > (p. 152), and **os::HASH**< **dataType** > (p. 202).

getLastNodeConst()

```
template<class dataType>
virtual const smart_ptr<nodeFrame<dataType> > os::iteratorBase< dataType >::getLastNodeConst ( ) const
[protected], [pure virtual]
```

Constant access to last node.

Returns

Immutable last node in the structure

Implemented in **os::AVL_TREE**< **dataType** > (p. 141), **os::LIST**< **dataType** > (p. 252), **os::SET**< **dataType** > (p. 298), **os::VECTOR**< **dataType** > (p. 341), **os::BOUNDED_QUEUE**< **dataType** > (p. 152), and **os::HASH**< **dataType** > (p. 202).

last()

```
template<class dataType>
iterator<dataType> os::iteratorBase< dataType >::last ( ) [final], [virtual]
```

Get last iterator.

Returns

Mutable last element

Implements **os::iteratorBaseInterface**< **dataType** > (p. 236).

search()

```
template<class dataType>
iterator<dataType> os::iteratorBase< dataType >::search (
             const dataType & dt ) [final], [virtual]
```

Search for a node.

Returns

Mutable element

Implements **os::iteratorBaseInterface**< **dataType** > (p. 236).

searchNode()

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

Search for a node.

Parameters

Pointer	to search for

Returns

Muttable found node, if applicable

Implemented in **os::AVL_TREE**< **dataType** > (p. 142), **os::BOUNDED_QUEUE**< **dataType** > (p. 154), **os::VECTOR**< **dataType** > (p. 343), **os::LIST**< **dataType** > (p. 253), **os::SET**< **dataType** > (p. 300), and **os::HASH**< **dataType** > (p. 203).

searchNodeConst()

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

Const search for a node.

Parameters

Pointer	to search for

Returns

Immutable found node, if applicable

Implemented in **os::AVL_TREE**< **dataType** > (p. 143), **os::BOUNDED_QUEUE**< **dataType** > (p. 154), **os::VECTOR**< **dataType** > (p. 343), **os::LIST**< **dataType** > (p. 254), **os::SET**< **dataType** > (p. 300), and **os::HASH**< **dataType** > (p. 204).

12.20.3 Friends And Related Function Documentation

constIterator< dataType >

`template<class dataType>`
`friend class` **constIterator**`< dataType > [friend]`

constIterator (p. 167) must have access to counter

iterator< dataType >

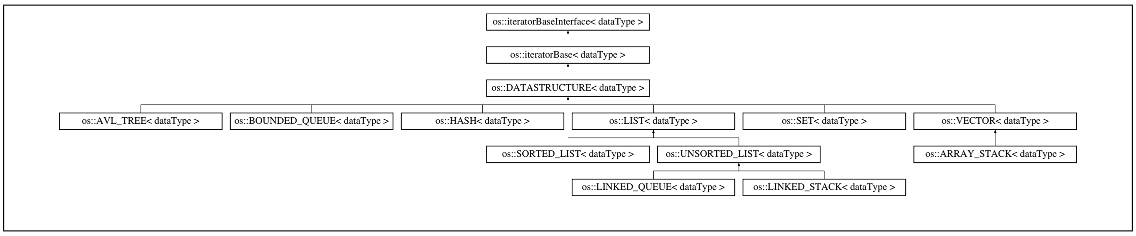
`template<class dataType>`
`friend class` **iterator**`< dataType > [friend]`

Iterator must have access to counter.

12.21 os::iteratorBaseInterface< dataType > Class Template Reference

Iterator base interface.

Inheritance diagram for os::iteratorBaseInterface< dataType >:



Public Member Functions

- virtual **iterator**< dataType > **first** ()=0

  Get first iterator.
- virtual **iterator**< dataType > **last** ()=0

  Get last iterator.
- virtual **constIterator**< dataType > **constFirst** () const =0

  Get first iterator.
- virtual **constIterator**< dataType > **constLast** () const =0

  Get last iterator.
- **constIterator**< dataType > **first** () const

  Get first iterator.
- **constIterator**< dataType > **last** () const

  Get last iterator.
- **iterator**< dataType > **begin** ()

  Get first iterator.
- **iterator**< dataType > **end** ()

Get last iterator.

- **constIterator**< dataType > **begin** () const

  Get first iterator.
- **constIterator**< dataType > **end** () const

  Get last iterator.
- virtual **iterator**< dataType > **search** (const dataType &dt)=0

  Search for a node.
- virtual **constIterator**< dataType > **constSearch** (const dataType &dt) const =0

  Serach for a node.
- **constIterator**< dataType > **search** (const dataType &dt) const

  Serach for a node.

12.21.1 Detailed Description

template<class dataType>
class os::iteratorBaseInterface< dataType >

Iterator base interface.

This interface is used to access iterators. It is designed to be used by the **os::iteratorBase** (p. 228), however, classes which provide iterator access may want to extend this interface.

12.21.2 Member Function Documentation

begin() `[1/2]`

`template<class dataType >`
iterator<dataType> **os::iteratorBaseInterface**`< dataType >::begin ( ) [inline]`

Get first iterator.

Returns

Mutable first element

begin() `[2/2]`

`template<class dataType >`
constIterator<dataType> **os::iteratorBaseInterface**`< dataType >::begin ( ) const [inline]`

Get first iterator.

Returns

Immutable first element

constFirst()

`template<class dataType >`
`virtual` **constIterator**`<dataType>` **os::iteratorBaseInterface**`< dataType >::constFirst ( ) const [pure virtual]`

Get first iterator.

Returns

Immutable first element

Implemented in **os::iteratorBase**< **dataType** > (p. 229).

constLast()

`template<class dataType >`
`virtual` **constIterator**`<dataType>` **os::iteratorBaseInterface**`< dataType >::constLast ( ) const [pure virtual]`

Get last iterator.

Returns

Immutable last element

Implemented in **os::iteratorBase**< **dataType** > (p. 229).

constSearch()

`template<class dataType >`
`virtual` **constIterator**`<dataType>` **os::iteratorBaseInterface**`< dataType >::constSearch ( const dataType & dt ) const [pure virtual]`

Serach for a node.

Parameters

in	*dt*	Node to search for

Returns

Immutable element

Implemented in **os::iteratorBase**< **dataType** > (p. 230).

end() [1/2]

`template<class dataType >`
iterator`<dataType>` **os::iteratorBaseInterface**`< dataType >::end ( ) [inline]`

Get last iterator.

Returns

Mutable last element

end() [2/2]

```
template<class dataType >
constIterator<dataType> os::iteratorBaseInterface< dataType >::end ( ) const  [inline]
```

Get last iterator.

Returns

Immutable last element

first() [1/2]

```
template<class dataType >
virtual iterator<dataType> os::iteratorBaseInterface< dataType >::first ( )  [pure virtual]
```

Get first iterator.

Returns

Mutable first element

Implemented in **os::iteratorBase**< **dataType** > (p. 230).

first() [2/2]

```
template<class dataType >
constIterator<dataType> os::iteratorBaseInterface< dataType >::first ( ) const  [inline]
```

Get first iterator.

Returns

Immutable first element

last() [1/2]

```
template<class dataType >
virtual iterator<dataType> os::iteratorBaseInterface< dataType >::last ( )  [pure virtual]
```

Get last iterator.

Returns

Mutable last element

Implemented in **os::iteratorBase**< **dataType** > (p. 231).

last() [2/2]

```
template<class dataType >
constIterator<dataType> os::iteratorBaseInterface< dataType >::last ( ) const  [inline]
```

Get last iterator.

Returns

Immutable last element

search() [1/2]

template<class dataType >
virtual **iterator**<dataType> **os::iteratorBaseInterface**< dataType >::search (
const dataType & *dt*) [pure virtual]

Search for a node.

Parameters

in	*dt*	Node to search for

Returns

Mutable element

Implemented in **os::iteratorBase**< **dataType** > (p. 231).

search() [2/2]

template<class dataType >
constIterator<dataType> **os::iteratorBaseInterface**< dataType >::search (
const dataType & *dt*) const [inline]

Serach for a node.

Parameters

in	*dt*	Node to search for

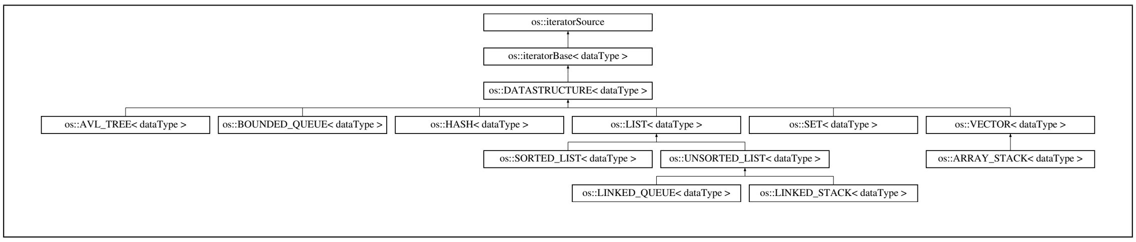
Returns

Immutable element

12.22 os::iteratorSource Class Reference

Base class for iterator source.

Inheritance diagram for os::iteratorSource:



Public Types

- enum **tearDown** { **IGNORE_OVERFLOW** =0, **THROW_ERROR**, **WAIT** }

Defines behaivor on teardown.

Public Member Functions

- **iteratorSource** ()
  Default constructor.
- **iteratorSource** (const **iteratorSource** &it)
  Copy constructor.
- virtual ~**iteratorSource** () throw (descriptiveException)
  Virtual destructor.
- unsigned **outstandingIterator** () const
  Access outstanding iterator counter.
- void **setTearDown** (**tearDown** td=**IGNORE_OVERFLOW**)
  Set tear-down behaivor.
- **tearDown getTearDown** () const
  Access tear-down behaivor.
- virtual bool **iterable** () const
  Returns if the source is iterable.
- virtual bool **randomAccess** () const
  Returns if the source can be accessed randomly.

Static Public Attributes

- static const bool **ITERABLE** = false
  By default, an iterated source is not iterable.
- static const bool **RANDOM_ACCESS** = false
  No random-access by default.

Protected Attributes

- std::atomic< unsigned > **_outstandingIterator**
  Outbound iterator count.
- **tearDown _tearDown**
  Defines the tear-down behavior.

12.22.1 Detailed Description

Base class for iterator source.

Provides an atomic iterator counter in-order to track the number of dispatched iterators.

12.22.2 Member Enumeration Documentation

tearDown

`enum` **os::iteratorSource::tearDown**

Defines behaivor on teardown.

Enumerator

IGNORE_OVERFLOW	Ignore outstanding iterators.
THROW_ERROR	Throw error for outstanding iterators.
WAIT	Wait for the destruction of outstanding iterators.

12.22.3 Constructor & Destructor Documentation

iteratorSource() [1/2]

`os::iteratorSource::iteratorSource ( ) [inline]`

Default constructor.

Sets the iterator count to 0 and initializes the standard tear-down behaviour. Constructor is protected because this class has no meaning if not extended.

iteratorSource() [2/2]

```
os::iteratorSource::iteratorSource (
            const iteratorSource & it ) [inline]
```

Copy constructor.

Iterator sources cannot be copied. Instead, they are re-created. [in] it **iteratorSource** (p. 237) to be copied

~iteratorSource()

`virtual os::iteratorSource::~iteratorSource ( ) throw descriptiveException) [inline], [virtual]`

Virtual destructor.

Destructor must be virtual, if an object of this type is deleted, the destructor of the type which inherits this class should be called. This particular destructor has behaviour defined by the value of _tearDown.

12.22.4 Member Function Documentation

getTearDown()

`tearDown os::iteratorSource::getTearDown ( ) const [inline]`

Access tear-down behaivor.

Returns

iteratorSource::_tearDown (p. 240)

iterable()

`virtual bool os::iteratorSource::iterable ( ) const [inline], [virtual]`

Returns if the source is iterable.

Returns

object::ITERABLE

Reimplemented in **os::AVL_TREE**< **dataType** > (p. 141), **os::BOUNDED_QUEUE**< **dataType** > (p. 153), **os::VECTOR**< **dataType** > (p. 342), **os::LIST**< **dataType** > (p. 253), **os::SET**< **dataType** > (p. 298), and **os::HASH**< **dataType** > (p. 203).

outstandingIterator()

`unsigned os::iteratorSource::outstandingIterator ( ) const [inline]`

Access outstanding iterator counter.

Returns

iteratorSource::_outstandingIterator (p. 240)

randomAccess()

`virtual bool os::iteratorSource::randomAccess ( ) const [inline], [virtual]`

Returns if the source can be accessed randomly.

Returns

object::RANDOM_ACCESS

Reimplemented in **os::AVL_TREE**< **dataType** > (p. 142), **os::BOUNDED_QUEUE**< **dataType** > (p. 154), **os::VECTOR**< **dataType** > (p. 342), **os::LIST**< **dataType** > (p. 253), **os::SET**< **dataType** > (p. 299), and **os::HASH**< **dataType** > (p. 203).

setTearDown()

`void os::iteratorSource::setTearDown ( **tearDown** td = **IGNORE_OVERFLOW** ) [inline]`

Set tear-down behaivor.

Parameters

in	*td*	New tear-down value

Returns

void

12.22.5 Member Data Documentation

_outstandingIterator

`std::atomic<unsigned> os::iteratorSource::_outstandingIterator [mutable], [protected]`

Outbound iterator count.

Holds the number of iterators which have been sent out into the larger world.

_tearDown

tearDown `os::iteratorSource::_tearDown [protected]`
Defines the tear-down behavior.

ITERABLE

`const bool os::iteratorSource::ITERABLE = false [static]`
By default, an iterated source is not iterable.

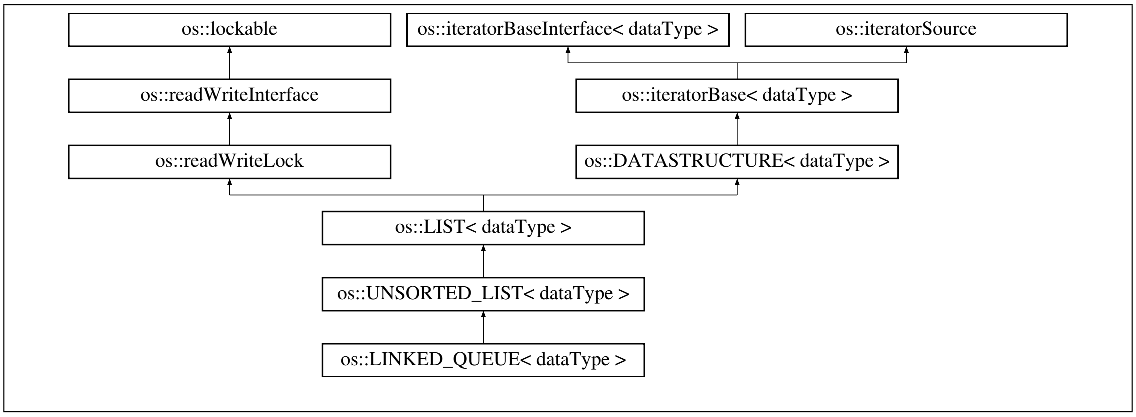
RANDOM_ACCESS

`const bool os::iteratorSource::RANDOM_ACCESS = false [static]`
No random-access by default.

12.23 os::LINKED_QUEUE< dataType > Class Template Reference

Queue built on-top of a list.

Inheritance diagram for os::LINKED_QUEUE< dataType >:



Public Member Functions

- **LINKED_QUEUE** ()

  Default constructor Simple constructor which implicitly call the constructor of it's base.
- **LINKED_QUEUE** (const **LINKED_QUEUE**< dataType > &cpy)

  Copy constructor.
- ~**LINKED_QUEUE** () final

  Destroys the stack Uses the destructor function defined by the unsorted list.
- void **pop** ()

  Remove top element.
- bool **push** (const dataType &x)

  Add element.
- const dataType & **constTop** () const

Const access top element.

- dataType & **top** ()

    Access top element.

- const dataType & **top** () const

    Const access top element.

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

Additional Inherited Members

12.23.1 Detailed Description

template<class dataType>
class os::LINKED_QUEUE< dataType >

Queue built on-top of a list.

Insertion occurs and the end, removal from the beginning. Note that this class retains all the funcitonality of it's parent, the unsorted linked list.

12.23.2 Constructor & Destructor Documentation

LINKED_QUEUE() [1/2]

template<class dataType>
os::LINKED_QUEUE< dataType >::**LINKED_QUEUE** ( ) [inline]

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

LINKED_QUEUE() [2/2]

template<class dataType>
os::LINKED_QUEUE< dataType >::**LINKED_QUEUE** (
const **LINKED_QUEUE**< dataType > & *cpy*) [inline]

Copy constructor.

This constructor builds a linked stack from another linked stack. Note that this copies by value, not reference.

Parameters

in	*cpy*	Target to be copied

~LINKED_QUEUE()

template<class dataType>
os::LINKED_QUEUE< dataType >::~**LINKED_QUEUE** ( ) [inline], [final]

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

12.23.3 Member Function Documentation

constTop() [1/2]

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

Const access top element.

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

Returns

Immutable top element

constTop() [2/2]

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

pop()

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

Remove top element.

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

Returns

void

push() [1/2]

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

Add element.

Inserts an element at the front of the queue

Parameters

in	*x*	Element to be inserted

Returns

true

push() [2/2]

```
template<class dataType>
```

bool **os::LINKED_QUEUE**< dataType >::push (
smart_ptr< dataType > *x*) [inline]

top() [1/4]

template<class dataType>
dataType& **os::LINKED_QUEUE**< dataType >::top ( ) [inline]

Access top element.

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

Returns

Mutable top element

top() [2/4]

template<class dataType>
const dataType& **os::LINKED_QUEUE**< dataType >::top ( ) const [inline]

Const access top element.

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

Returns

Immutable top element

top() [3/4]

template<class dataType>
smart_ptr<dataType> **os::LINKED_QUEUE**< dataType >::top ( ) [inline]

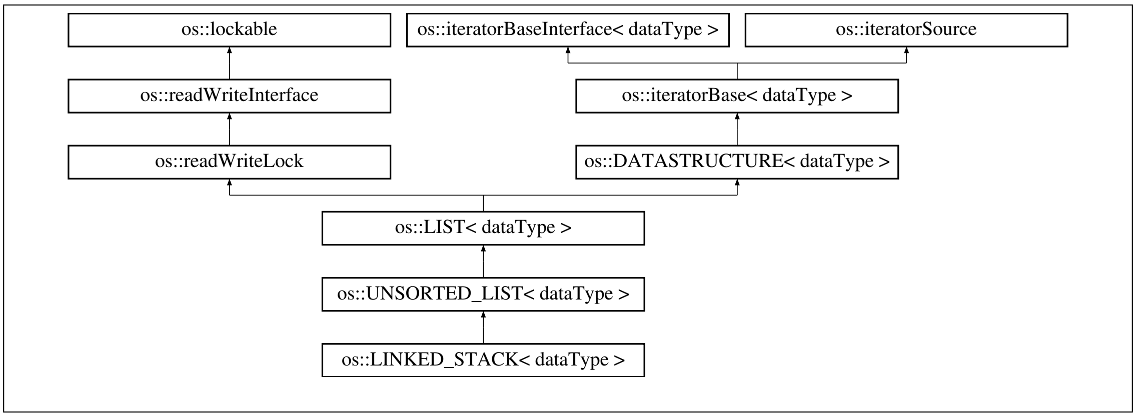
top() [4/4]

template<class dataType>
const **smart_ptr**<dataType> **os::LINKED_QUEUE**< dataType >::top ( ) const [inline]

12.24 os::LINKED_STACK< dataType > Class Template Reference

Stack built on-top of a list.

Inheritance diagram for os::LINKED_STACK< dataType >:



Public Member Functions

- **LINKED_STACK** ()

    Default constructor Simple constructor which implicitly call the constructor of it's base.
- **LINKED_STACK** (const **LINKED_STACK**< dataType > &cpy)

    Copy constructor.
- ~**LINKED_STACK** () final

    Destroys the stack Uses the destructor function defined by the unsorted list.
- void **pop** ()

    Remove top element.
- bool **push** (const dataType &x)

    Add element.
- const dataType & **constTop** () const

    Const access top element.
- dataType & **top** ()

    Access top element.
- const dataType & **top** () const

    Const access top element.
- bool **push** (**smart_ptr**< dataType > x)
- const **smart_ptr**< dataType > **constTop** () const
- **smart_ptr**< dataType > **top** ()
- const **smart_ptr**< dataType > **top** () const

Additional Inherited Members

12.24.1 Detailed Description

template<class dataType>
class os::LINKED_STACK< dataType >

Stack built on-top of a list.

Insertion occurs and the top, removal from the top. Note that this class retains all the funcitonality of it's parent, the unsorted linked list.

12.24.2 Constructor & Destructor Documentation

LINKED_STACK() [1/2]

template<class dataType>
os::LINKED_STACK< dataType >::**LINKED_STACK** ( ) [inline]

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

LINKED_STACK() [2/2]

template<class dataType>
os::LINKED_STACK< dataType >::**LINKED_STACK** (
const **LINKED_STACK**< dataType > & *cpy*) [inline]

Copy constructor.

This constructor builds a linked stack from another linked stack. Note that this copies by value, not reference.

Parameters

in	*cpy*	Target to be copied

~LINKED_STACK()

template<class dataType>
os::LINKED_STACK< dataType >::~**LINKED_STACK** ( ) [inline], [final]

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

12.24.3 Member Function Documentation

constTop() [1/2]

template<class dataType>
const dataType& **os::LINKED_STACK**< dataType >::constTop ( ) const [inline]

Const access top element.

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

Returns

Immutable top element

constTop() [2/2]

template<class dataType>
const **smart_ptr**<dataType> **os::LINKED_STACK**< dataType >::constTop ( ) const [inline]

pop()

template<class dataType>
void **os::LINKED_STACK**< dataType >::pop ( ) [inline]

Remove top element.

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

Returns

void

push() [1/2]

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

Add element.

Inserts an element at the top of the stack

Parameters

in	*x*	Element to be inserted

Returns

true

push() [2/2]

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

top() [1/4]

template<class dataType>
dataType& **os::LINKED_STACK**< dataType >::top ( ) [inline]

Access top element.

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

Returns

Mutable top element

top() [2/4]

template<class dataType>
const dataType& **os::LINKED_STACK**< dataType >::top ( ) const [inline]

Const access top element.

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

Returns

Immutable top element

top() [3/4]

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

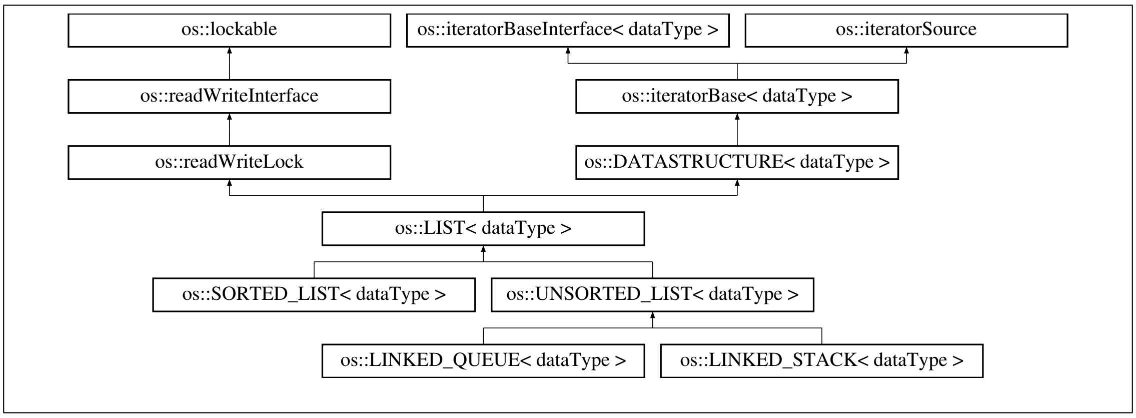
top() [4/4]

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

12.25 os::LIST< dataType > Class Template Reference

List framework Template for a linked list, insertion is fully defined in subsequent extensions of this class.

Inheritance diagram for os::LIST< dataType >:



Public Member Functions

- **LIST** ()

  Constructs the linked list Sets the head and tail to NULL, number of elements in the list to 0.
- virtual ~**LIST** ()

  Destroys the list Removes every element from the list. Depending on the list format, this may delete everythin in the list.
- size_t **size** () const final

  Access size of the list.

- bool **iterable** () const final

  Returns if the relevant node type is iterable.
- bool **randomAccess** () const final

  Returns if the relevant node type can be accessed randomly.
- bool **remove** (const dataType &x) final

  Remove item from the list.
- bool **find** (const dataType &x) const final

  Searches for an item.
- dataType & **access** (const dataType &x) final

  Mutable item access.
- const dataType & **access** (const dataType &x) const final

  Immutable item access.

Static Public Attributes

- static const bool **ITERABLE** = true

  Lists are iterable.
- static const bool **RANDOM_ACCESS** = false

  Lists do not allow random access.

Protected Member Functions

- **smart_ptr**< **nodeFrame**< dataType > > **getFirstNode** () final

  Access to first node.
- **smart_ptr**< **nodeFrame**< dataType > > **getLastNode** () final

  Access to last node.
- const **smart_ptr**< **nodeFrame**< dataType > > **getFirstNodeConst** () const final

  Constant access to first node.
- const **smart_ptr**< **nodeFrame**< dataType > > **getLastNodeConst** () const final

  Constant access to last node.
- **smart_ptr**< **nodeFrame**< dataType > > **searchNode** (const **smart_ptr**< dataType > dt) final

  Search for a node.
- const **smart_ptr**< **nodeFrame**< dataType > > **searchNodeConst** (const **smart_ptr**< data↩ Type > dt) const final

  Const search for a node.
- **LIST** (const **LIST**< dataType > ∗cpy)

  Copy constructor with pointer.
- void **assignmentHelper** (const **LIST**< dataType > ∗cpy)

Protected Attributes

- size_t **_size**

  Size of the list.
- **smart_ptr**< ITERATOR_CLASS > **_beg**

  First pointer.
- **smart_ptr**< ITERATOR_CLASS > **_end**

  Last pointer.

Additional Inherited Members

12.25.1 Detailed Description

template<class dataType>
class os::LIST< dataType >

List framework Template for a linked list, insertion is fully defined in subsequent extensions of this class.

12.25.2 Constructor & Destructor Documentation

LIST() [1/2]

```
template<class dataType>
os::LIST< dataType >::LIST (
            const LIST< dataType > * cpy )  [inline], [protected]
```

Copy constructor with pointer.

This constructor builds a linked list from another linked list. Note that this copies by value, not reference.

Parameters

in	*cpy*	Target to be copied

LIST() [2/2]

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

Constructs the linked list Sets the head and tail to NULL, number of elements in the list to 0.

~LIST()

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

Destroys the list Removes every element from the list. Depending on the list format, this may delete everythin in the list.

12.25.3 Member Function Documentation

access() [1/2]

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

Mutable item access.

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

Returns

Mutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 179).

access() [2/2]

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

Immutable item access.

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

Returns

Immutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 181).

assignmentHelper()

```
template<class dataType>
void os::LIST< dataType >::assignmentHelper (
              const LIST< dataType > * cpy )  [inline], [protected]
```

Used in the assignment operator

Parameters

in	*cpy*	Target to be copied

Returns

void

find()

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

Searches for an item.

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

Returns

True if found, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 182).

getFirstNode()

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

Access to first node.

Returns

First node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 230).

getFirstNodeConst()

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

Constant access to first node.

Returns

Immutable first node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 230).

getLastNode()

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

Access to last node.

Returns

Last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 231).

getLastNodeConst()

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

Constant access to last node.

Returns

Immutable last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 231).

iterable()

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

Returns if the relevant node type is iterable.

Returns

true

Reimplemented from **os::iteratorSource** (p. 239).

randomAccess()

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

Returns if the relevant node type can be accessed randomly.

Returns

true

Reimplemented from **os::iteratorSource** (p. 240).

remove()

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

Remove item from the list.
This function is O(n) where n is the number of elements in the list.

```
@return True if removed, else, false
```

Implements **os::DATASTRUCTURE**< **dataType** > (p. 184).

searchNode()

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

Search for a node.

Parameters

in	*dt*	Pointer to search for

Returns

Muttable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 232).

searchNodeConst()

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

Const search for a node.

Parameters

in	*dt*	Pointer to search for

Returns

Immutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 232).

size()

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

Access size of the list.

Returns

Number of elements in the list

Implements **os::DATASTRUCTURE**< **dataType** > (p. 184).

12.25.4 Member Data Documentation

_beg

```
template<class dataType>
smart_ptr<ITERATOR_CLASS > os::LIST< dataType >::_beg  [protected]
```

First pointer.

_end

```
template<class dataType>
smart_ptr<ITERATOR_CLASS > os::LIST< dataType >::_end  [protected]
```

Last pointer.

_size

```
template<class dataType>
size_t os::LIST< dataType >::_size [protected]
```

Size of the list.

ITERABLE

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

Lists are iterable.

RANDOM_ACCESS

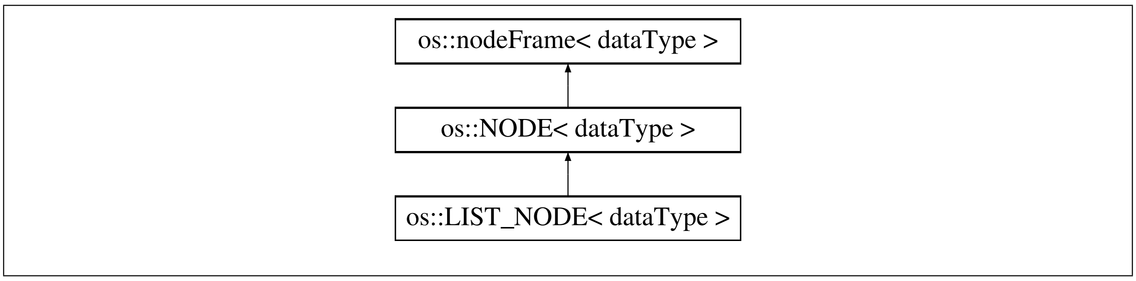
```
template<class dataType>
const bool os::LIST< dataType >::RANDOM_ACCESS = false [static]
```

Lists do not allow random access.

12.26 os::LIST_NODE< dataType > Class Template Reference

List node Node used in linked lists to store each element.

Inheritance diagram for os::LIST_NODE< dataType >:



Public Member Functions

- ~**LIST_NODE** () final

  Basic destructor Merely dereferences and deletes the contained objects. Does not disassemble the list.
- void **remove** () final throw (descriptiveException)

  Remove this node from the list.
- bool **iterable** () const final

  Returns if the node is iterable.
- bool **randomAccess** () const final

  Returns if the node can be accessed randomly.
- **smart_ptr**< **nodeFrame**< dataType > > **getNext** () final throw (descriptiveException)

  Returns the next vector frame.

- const **smart_ptr**< **nodeFrame**< dataType > > **getNextConst** () const final throw (descriptive↩ Exception)

  Returns the next vector frame.
- **smart_ptr**< **nodeFrame**< dataType > > **getPrev** () final throw (descriptiveException)

  Returns the previous vector frame.
- const **smart_ptr**< **nodeFrame**< dataType > > **getPrevConst** () const final throw (descriptive↩ Exception)

  Returns the previous vector frame.

Static Public Attributes

- static const bool **ITERABLE** = true

  List frames are iterable.
- static const bool **RANDOM_ACCESS** = false

  List frames do not allow random-access.

Private Member Functions

- **LIST_NODE** (DRIVER_CLASS ∗src, const dataType &dt)

  Private constructor.

Private Attributes

- **smart_ptr**< **LIST_NODE** > **prev**

  Previous node.
- **smart_ptr**< **LIST_NODE** > **next**

  Next node.

Additional Inherited Members

12.26.1 Detailed Description

template<class dataType>
class os::LIST_NODE< dataType >

List node Node used in linked lists to store each element.

12.26.2 Constructor & Destructor Documentation

LIST_NODE()

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

Private constructor.
Only lists can access the node constructor.

~LIST_NODE()

`template<class dataType >`
os::LIST_NODE`< dataType >::~`**LIST_NODE** `( ) [inline], [final]`

Basic destructor Merely dereferences and deletes the contained objects. Does not disassemble the list.

12.26.3 Member Function Documentation

getNext()

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

Returns the next vector frame.

Returns

Next node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 277).

getNextConst()

`template<class dataType >`
`const` **smart_ptr**<**nodeFrame**`<dataType> >` **os::LIST_NODE**`< dataType >::getNextConst ( ) const throw` **descriptive←Exception**`) [inline], [final], [virtual]`

Returns the next vector frame.

Returns

Next node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 278).

getPrev()

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

Returns the previous vector frame.

Returns

Previous node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 278).

getPrevConst()

`template<class dataType >`
`const` **smart_ptr**<**nodeFrame**`<dataType> >` **os::LIST_NODE**`< dataType >::getPrevConst ( ) const throw` **descriptive←Exception**`) [inline], [final], [virtual]`

Returns the previous vector frame.

Returns

Previous node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 279).

iterable()

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

Returns if the node is iterable.

Returns

object::ITERABLE

Reimplemented from **os::nodeFrame**< **dataType** > (p. 279).

randomAccess()

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

Returns if the node can be accessed randomly.

Returns

object::RANDOM_ACCESS

Reimplemented from **os::nodeFrame**< **dataType** > (p. 281).

remove()

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

Remove this node from the list.

Returns

void

Reimplemented from **os::nodeFrame**< **dataType** > (p. 281).

12.26.4 Member Data Documentation

ITERABLE

template<class dataType >
const bool **os::LIST_NODE**< dataType >::ITERABLE = true [static]

List frames are iterable.

next

template<class dataType >
smart_ptr<**LIST_NODE** > **os::LIST_NODE**< dataType >::next [private]

Next node.

prev

template<class dataType >
smart_ptr<**LIST_NODE** > **os::LIST_NODE**< dataType >::prev [private]

Previous node.

RANDOM_ACCESS

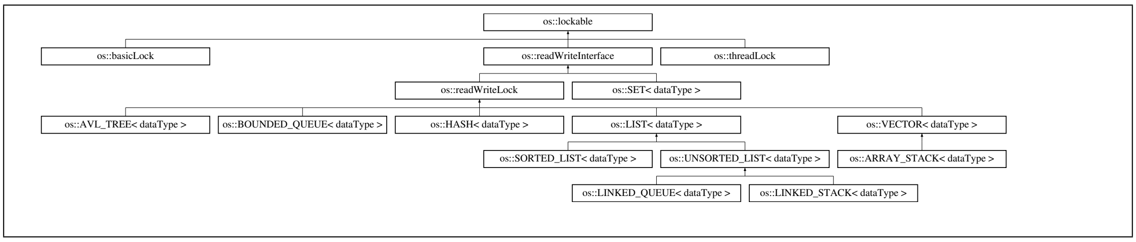
template<class dataType >
const bool **os::LIST_NODE**< dataType >::RANDOM_ACCESS = false [static]

List frames do not allow random-access.

12.27 os::lockable Class Reference

An interface defining a class which can be locked.

Inheritance diagram for os::lockable:



Public Member Functions

- **lockable** (bool dMode=**NO_LOCK_CHECK**) throw ()
  Lockable constructor.
- virtual ~**lockable** () throw (descriptiveException)
  Virtual destructor.
- void **setDeletionMode** (bool delMode) throw ()
  Set deletion mode.
- bool **deletionMode** () const throw ()
  Access deletion mode.
- virtual void **lock** () const =0 throw ()
  Locks the lock.
- virtual void **unlock** () const =0 throw (descriptiveException)
  Unlocks the lock.
- virtual bool **locked** () const =0 throw ()
  Checks if the lock is locked.
- virtual bool **try_lock** () const =0 throw ()
  Attempt to lock the object.
- void **acquire** () const throw ()
  *Calls **lockable::lock()** (p. 261)*

- void **release** () const throw (descriptiveException)

    *Calls **lockable::unlock()** (p. 262)*
- bool **attemptLock** () const throw ()

    *Calls **lockable::try_lock()** (p. 261)*

Static Public Attributes

- static const bool **NO_LOCK_CHECK** =false

    Do check lock status on deletion.
- static const bool **DELETION_LOCK_CHECK** =true

    Check lock status on deletion.

Private Attributes

- bool **_delMode**

12.27.1 Detailed Description

An interface defining a class which can be locked.

This interface is used by various types of locks to define a common method of both locking and unlocking.

12.27.2 Constructor & Destructor Documentation

lockable()

```
os::lockable::lockable (
              bool dMode = NO_LOCK_CHECK ) throw )   [inline]
```

Lockable constructor.

~lockable()

```
virtual os::lockable::~lockable ( ) throw descriptiveException)   [inline], [virtual]
```

Virtual destructor.

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

12.27.3 Member Function Documentation

acquire()

```
void os::lockable::acquire ( ) const throw )   [inline]
```

Calls **lockable::lock()** (p. 261)

attemptLock()

```
bool os::lockable::attemptLock ( ) const throw )   [inline]
```

Calls **lockable::try_lock()** (p. 261)

deletionMode()

`bool os::lockable::deletionMode ( ) const throw )   [inline]`

Access deletion mode.

Returns

lockable::_delMode (p. 262)

lock()

`virtual void os::lockable::lock ( ) const throw )   [pure virtual]`

Locks the lock.

Returns

void

Implemented in **os::SET**< **dataType** > (p. 299), **os::readWriteLock** (p. 289), **os::threadLock** (p. 332), and **os::basicLock** (p. 146).

locked()

`virtual bool os::lockable::locked ( ) const throw )   [pure virtual]`

Checks if the lock is locked.

Returns

True if locked, else, false

Implemented in **os::SET**< **dataType** > (p. 299), **os::readWriteLock** (p. 289), **os::threadLock** (p. 332), and **os::basicLock** (p. 146).

release()

`void os::lockable::release ( ) const throw descriptiveException)   [inline]`

Calls **lockable::unlock()** (p. 262)

setDeletionMode()

```
void os::lockable::setDeletionMode (
            bool delMode ) throw )   [inline]
```

Set deletion mode.

Parameters

in	*delMode*	Deletion mode to bind

Returns

void

try_lock()

`virtual bool os::lockable::try_lock ( ) const throw )   [pure virtual]`

Attempt to lock the object.

Locks the object if possible, otherwise, returns false.

Returns

True if lock successful

Implemented in **os::SET**< **dataType** > (p. 301), **os::readWriteLock** (p. 290), **os::threadLock** (p. 333), and **os::basicLock** (p. 146).

unlock()

`virtual void os::lockable::unlock ( ) const throw` **descriptiveException**`)   [pure virtual]`

Unlocks the lock.

Returns

void

Implemented in **os::SET**< **dataType** > (p. 301), **os::readWriteLock** (p. 291), **os::threadLock** (p. 333), and **os::basicLock** (p. 146).

12.27.4 Member Data Documentation

_delMode

`bool os::lockable::_delMode  [private]`

Holds deletion mode of the lockable object

DELETION_LOCK_CHECK

`const bool os::lockable::DELETION_LOCK_CHECK =true  [static]`

Check lock status on deletion.

NO_LOCK_CHECK

`const bool os::lockable::NO_LOCK_CHECK =false  [static]`

Do check lock status on deletion.

12.28 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.

12.28.1 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<data↩Type>.

12.28.2 Constructor & Destructor Documentation

matrix() [1/5]

```
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	*w*	Width of matrix, default 0
in	*h*	Height of matrix, default 0

matrix() [2/5]

```
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	*m*	Matrix to be copied

matrix() [3/5]

```
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	*m*	Indirect matrix to be copied

matrix() [4/5]

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	*d*	Data array to be copied
in	*w*	Width of matrix
in	*d*	Height of matrix

matrix() [5/5]

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	*d*	Indirect data array to be copied
in	*w*	Width of matrix
in	*d*	Height of matrix

~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.

12.28.3 Member Function Documentation

constGet()

```
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	*w*	X position
in	*h*	Y position

Returns

Constant reference to matrix element

get()

```
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	*w*	X position
in	*h*	Y position

Returns

Modifiable reference to matrix element

getArray()

```
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. 269)

getConstArray()

```
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. 269)

height()

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

Return _height of matrix.

Returns

matrix<**dataType**>**::_height** (p. 269)

operator()()

```
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	*w*	X position
in	*h*	Y position

Returns

Modifiable reference to matrix element

operator=() [1/2]

```
template<class dataType>
matrix<dataType>& os::matrix< dataType >::operator= (
            const matrix< dataType > & m ) throw )
```

Equality constructor.

Re-constructs the raw matrix from another raw matrix. Note that the two matrices do not share the same data array.

Parameters

in	*m*	Reference to matrix being copied

Returns

Reference to self

operator=() [2/2]

```
template<class dataType>
matrix<dataType>& os::matrix< dataType >::operator= (
            const indirectMatrix< dataType > & m ) throw )
```

Equality constructor.

Re-constructs the raw matrix from an indirect matrix. Note that the two matrices do not share the same data array.

Parameters

in	*m*	Reference to matrix being copied

Returns

Reference to self

width()

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

Return _width of matrix.

Returns

matrix<dataType>::_width (p. 269)

12.28.4 Friends And Related Function Documentation

indirectMatrix< dataType >

```
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.

12.28.5 Member Data Documentation

_height

```
template<class dataType>
uint32_t os::matrix< dataType >::_height  [private]
```

Height of the matrix.

_width

```
template<class dataType>
uint32_t os::matrix< dataType >::_width  [private]
```

Width of the matrix.

data

```
template<class dataType>
smart_ptr<dataType> os::matrix< dataType >::data  [private]
```

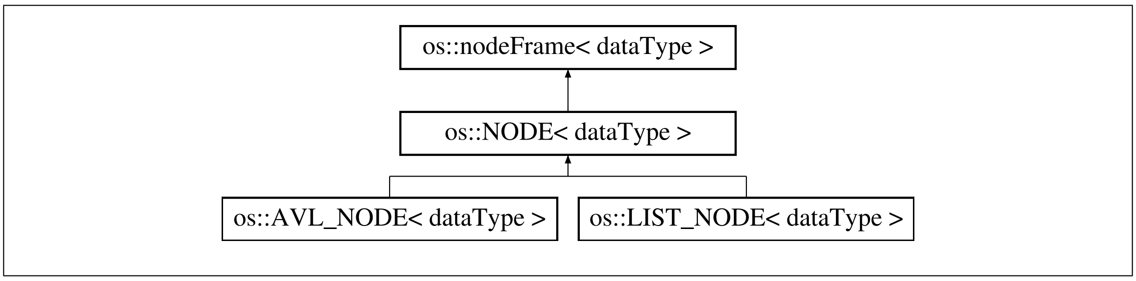
Data array.

For the raw matrix class, this array contains all of the data used by the matrix in a block of size _width∗_height.

12.29 os::NODE< dataType > Class Template Reference

Object node definition.

Inheritance diagram for os::NODE< dataType >:



Public Member Functions

- **NODE** ()

  Default constructor.
- **NODE** (const **NODE** &cph)

  Copy constructor.
- virtual ~**NODE** ()

  Virtual destructor.
- **NODE** (const dataType &dt)

  Data constructor Initializes the node with specific data.
- **smart_ptr**< dataType > **get** () final throw ()

  Access data pointer.
- const **smart_ptr**< dataType > **constGet** () const final throw ()

  Const access data pointer.
- dataType & **operator*** () final throw (descriptiveException)

  Access to data reference.
- const dataType & **operator*** () const final throw (descriptiveException)

  Const access to data reference.
- bool **valid** () const final

  Valid data query.
- int **compare** (const objectNode< dataType > &cmp) const

  Compare two nodes.
- **operator size_t** () const

  size_t cast

Protected Attributes

- dataType **_data**

  Data held by this node.

Additional Inherited Members

12.29.1 Detailed Description

template<class dataType>
class os::NODE< dataType >

Object node definition.

Note that this class is defined in three forms: objectNode, pointerNode and rawPointerNode. In short, this class and it's alternate forms specifiy the details of the **nodeFrame** (p. 274), allowing for holding both objects and pointers to object in the various provided data-structures.

12.29.2 Constructor & Destructor Documentation

NODE() [1/3]

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

Default constructor.

NODE() [2/3]

```
template<class dataType >
os::NODE< dataType >::NODE (
            const NODE< dataType > & cph )  [inline]
```

Copy constructor.

Parameters

in	*cph*	Node to be copied

~NODE()

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

Virtual destructor.

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

NODE() [3/3]

```
template<class dataType >
os::NODE< dataType >::NODE (
            const dataType & dt )  [inline]
```

Data constructor Initializes the node with specific data.

Parameters

in	*dt*	Reference to data

12.29.3 Member Function Documentation

compare()

```
template<class dataType >
int os::NODE< dataType >::compare (
            const objectNode< dataType > & cmp ) const  [inline]
```

Compare two nodes.

Uses the == and > operators of the given data type to compare two nodes. Note that other forms of this class use different means of comparison.

Parameters

in	*cmp*	Node to compare against

Returns

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

constGet()

template<class dataType >
const **smart_ptr**<dataType> **os::NODE**< dataType >::constGet ( ) const throw) [inline], [final], [virtual]

Const access data pointer.

Returns a pointer to the data held in this node. Note that this function will return "data" if the node is in pointer form.

Returns

Immutable pointer to data

Implements **os::nodeFrame**< **dataType** > (p. 277).

get()

template<class dataType >
smart_ptr<dataType> **os::NODE**< dataType >::get ( ) throw) [inline], [final], [virtual]

Access data pointer.

Returns a pointer to the data held in this node. Note that this function will return "data" if the node is in pointer form.

Returns

Mutable pointer to _data

Implements **os::nodeFrame**< **dataType** > (p. 277).

operator size_t()

template<class dataType >
os::NODE< dataType >::operator size_t ( ) const [inline]

size_t cast

Used in hash functions to place this object in a hash table. Note that other forms of this class use different methods for converting to size_t.

Returns

data cast to size_t

operator*() [1/2]

```
template<class dataType >
dataType& os::NODE< dataType >::operator* ( ) throw descriptiveException)   [inline], [final], [virtual]
```

Access to data reference.

Returns an object reference to the data enclosed in the node. Note that if the node is in pointer form, a reference to the object pointed to by the pointer will be returned.

Returns

Mutable object reference

Implements **os::nodeFrame**< **dataType** > (p. 280).

operator*() [2/2]

```
template<class dataType >
const dataType& os::NODE< dataType >::operator* ( ) const throw descriptiveException)   [inline], [final], [virtual]
```

Const access to data reference.

Returns an object reference to the data enclosed in the node. Note that if the node is in pointer form, a reference to the object pointed to by the pointer will be returned.

Returns

Immutable object reference

Implements **os::nodeFrame**< **dataType** > (p. 280).

valid()

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

Valid data query.

Checks if the provided data is valid. Note that pointer forms of this class return false if the provided pointer is NULL

Returns

true

Reimplemented from **os::nodeFrame**< **dataType** > (p. 282).

12.29.4 Member Data Documentation

_data

```
template<class dataType >
dataType os::NODE< dataType >::_data  [protected]
```

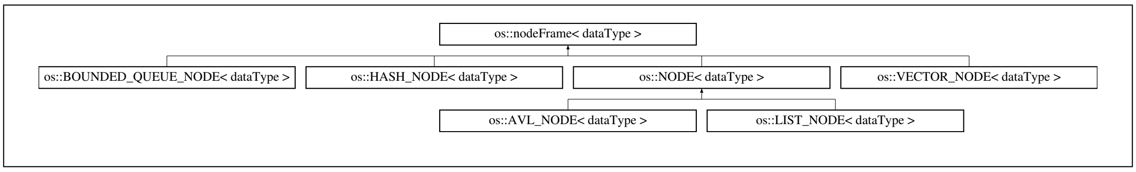
Data held by this node.

Note that other forms of this class hold pointers instead of objects

12.30 os::nodeFrame< dataType > Class Template Reference

nodeFrame (p. 274) interface

Inheritance diagram for os::nodeFrame< dataType >:



Public Member Functions

- virtual ~**nodeFrame** ()

  Virtual destructor.
- virtual **smart_ptr**< dataType > **get** ()=0 throw ()

  Returns a pointer.
- virtual const **smart_ptr**< dataType > **constGet** () const =0 throw ()

  Returns a const pointer.
- const **smart_ptr**< dataType > **get** () const throw ()

  Returns a const pointer.
- **smart_ptr**< dataType > **operator->** () throw ()

  Member access.
- const **smart_ptr**< dataType > **operator->** () const throw ()

  Member access.
- virtual dataType & **operator*** ()=0 throw (descriptiveException)

  De-reference.
- virtual const dataType & **operator*** () const =0 throw (descriptiveException)

  De-reference.
- virtual bool **valid** () const

  Valid data query Checks if the provided data is valid.
- **operator bool** () const throw ()

  Boolean conversion.
- bool **operator!** () const throw ()

  Inverted boolean conversion.
- virtual bool **iterable** () const

  Returns if the node is iterable.
- virtual bool **randomAccess** () const

  Returns if the node can be accessed randomly.
- virtual bool **isObject** () const =0

  True if holding an object.
- virtual bool **isPointer** () const =0

  True is holding a pointer.
- virtual bool **usingComparison** () const =0

Defines which comparison method is used.

- virtual void **remove** () throw (descriptiveException)

  Remove node from data structure.
- virtual **smart_ptr**< **nodeFrame**< dataType > > **getNext** () throw (descriptiveException)

  *Returns the next **nodeFrame** (p. 274).*
- virtual const **smart_ptr**< **nodeFrame**< dataType > > **getNextConst** () const throw (descriptiveException)

  *Returns the next **nodeFrame** (p. 274).*
- const **smart_ptr**< **nodeFrame**< dataType > > **getNext** () const throw (descriptiveException)

  *Returns the next **nodeFrame** (p. 274).*
- virtual **smart_ptr**< **nodeFrame**< dataType > > **getPrev** () throw (descriptiveException)

  *Returns the previous **nodeFrame** (p. 274).*
- virtual const **smart_ptr**< **nodeFrame**< dataType > > **getPrevConst** () const throw (descriptiveException)

  *Returns the previous **nodeFrame** (p. 274).*
- const **smart_ptr**< **nodeFrame**< dataType > > **getPrev** () const throw (descriptiveException)

  *Returns the previous **nodeFrame** (p. 274).*
- virtual **smart_ptr**< **nodeFrame**< dataType > > **access** (long offset) throw (descriptiveException)

  Access node by index.
- virtual const **smart_ptr**< **nodeFrame**< dataType > > **constAccess** (long offset) const throw (descriptiveException)

  Access node by index.
- const **smart_ptr**< **nodeFrame**< dataType > > **access** (long offset) const throw (descriptiveException)

  Access node by index.

Static Public Attributes

- static const bool **ITERABLE** = false

  By default, nodeFrames are not iterable.
- static const bool **RANDOM_ACCESS** = false

  No random-access by default.

12.30.1 Detailed Description

template<class dataType>
class os::nodeFrame< dataType >

nodeFrame (p. 274) interface

Declares a number of functions to be used by various nodes inside data-structures.

12.30.2 Constructor & Destructor Documentation

~nodeFrame()

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

Virtual destructor.

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

12.30.3 Member Function Documentation

access() [1/2]

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

Access node by index.

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

Returns

Offset node, mutable

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 159), **os::VECTOR_NODE**< **dataType** > (p. 374), and **os::AVL_NODE**< **dataType** > (p. 126).

access() [2/2]

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

Access node by index.

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

Returns

Offset node, immutable

constAccess()

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

Access node by index.

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

Returns

Offset node, immutable

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 159), **os::VECTOR_NO**↩ **DE**< **dataType** > (p. 374), and **os::AVL_NODE**< **dataType** > (p. 127).

constGet()

`template<class dataType >`
`virtual const` **smart_ptr**`<dataType>` **os::nodeFrame**`< dataType >::constGet ( ) const throw )   [pure virtual]`

Returns a const pointer.
Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

Implemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 159), **os::VECTOR_NOD**↩ **E**< **dataType** > (p. 374), **os::HASH_NODE**< **dataType** > (p. 207), and **os::NODE**< **dataType** > (p. 272).

get() [1/2]

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

Returns a pointer.
Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

Implemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 159), **os::VECTOR_NOD**↩ **E**< **dataType** > (p. 375), **os::HASH_NODE**< **dataType** > (p. 207), and **os::NODE**< **dataType** > (p. 272).

get() [2/2]

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

Returns a const pointer.
Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

getNext() [1/2]

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

Returns the next **nodeFrame** (p. 274).

Attempts to return the next node frame, assuming that the node in question is iterable. If a node is not iterable, an exception will be thrown.

Returns

Next node, mutable

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 160), **os::VECTOR_NODE**< **dataType** > (p. 375), **os::HASH_NODE**< **dataType** > (p. 207), **os::AVL_NODE**< **dataType** > (p. 127), and **os::LIST_NODE**< **dataType** > (p. 257).

getNext() [2/2]

template<class dataType >
const **smart_ptr**<**nodeFrame**<dataType> > **os::nodeFrame**< dataType >::getNext ( ) const throw **descriptiveException**) [inline]

Returns the next **nodeFrame** (p. 274).

Attempts to return the next node frame, assuming that the node in question is iterable. If a node is not iterable, an exception will be thrown.

Returns

Next node, immutable

getNextConst()

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

Returns the next **nodeFrame** (p. 274).

Attempts to return the next node frame, assuming that the node in question is iterable. If a node is not iterable, an exception will be thrown.

Returns

Next node, immutable

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 160), **os::VECTOR_NODE**< **dataType** > (p. 375), **os::HASH_NODE**< **dataType** > (p. 207), **os::AVL_NODE**< **dataType** > (p. 128), and **os::LIST_NODE**< **dataType** > (p. 257).

getPrev() [1/2]

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

Returns the previous **nodeFrame** (p. 274).

Attempts to return the previous node frame, assuming that the node in question is iterable. If a node is not iterable, an exception will be thrown.

Returns

Previous node, mutable

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 160), **os::VECTOR_NODE**< **dataType** > (p. 375), **os::HASH_NODE**< **dataType** > (p. 208), **os::AVL_NODE**< **dataType** > (p. 128), and **os::LIST_NODE**< **dataType** > (p. 257).

getPrev() [2/2]

`template<class dataType >`
`const` **smart_ptr**<**nodeFrame**<dataType> > **os::nodeFrame**< dataType >::getPrev ( ) const throw **descriptiveException**) [inline]

Returns the previous **nodeFrame** (p. 274).

Attempts to return the previous node frame, assuming that the node in question is iterable. If a node is not iterable, an exception will be thrown.

Returns

Previous node, immutable

getPrevConst()

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

Returns the previous **nodeFrame** (p. 274).

Attempts to return the previous node frame, assuming that the node in question is iterable. If a node is not iterable, an exception will be thrown.

Returns

Previous node, immutable

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 160), **os::VECTOR_NODE**< **dataType** > (p. 376), **os::HASH_NODE**< **dataType** > (p. 208), **os::AVL_NODE**< **dataType** > (p. 128), and **os::LIST_NODE**< **dataType** > (p. 257).

isObject()

`template<class dataType >`
`virtual bool` **os::nodeFrame**< dataType >::isObject ( ) const [pure virtual]

True if holding an object.

isPointer()

`template<class dataType >`
`virtual bool` **os::nodeFrame**< dataType >::isPointer ( ) const [pure virtual]

True is holding a pointer.

iterable()

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

Returns if the node is iterable.

Returns

object::ITERABLE

Reimplemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 161), **os::VECTOR_NODE**< **dataType** > (p. 376), **os::HASH_NODE**< **dataType** > (p. 208), **os::AVL_NODE**< **dataType** > (p. 129), and **os::LIST_NODE**< **dataType** > (p. 258).

operator bool()

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

Boolean conversion.

Returns

valid() (p. 282)

operator"!()

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

Inverted boolean conversion.

Returns

!valid()

operator*() [1/2]

`template<class dataType >`
`virtual dataType&` **os::nodeFrame**`< dataType >::operator* ( ) throw` **descriptiveException**`) [pure virtual]`

De-reference.

Returns a reference to the contained object, the reference can be modified.

Returns

Contained object

Implemented in **os::BOUNDED_QUEUE_NODE**< **dataType** > (p. 161), **os::VECTOR_NODE**< **dataType** > (p. 376), **os::HASH_NODE**< **dataType** > (p. 208), and **os::NODE**< **dataType** > (p. 272).

operator*() [2/2]

```
template<class dataType >
virtual const dataType& os::nodeFrame< dataType >::operator* ( ) const throw descriptiveException) [pure virtual]
```

De-reference.

Returns a const reference to the contained object, the reference cannot be modified.

Returns

Contained object

Implemented in **os::BOUNDED_QUEUE_NODE< dataType >** (p. 161), **os::VECTOR_NODE< dataType >** (p. 376), **os::HASH_NODE< dataType >** (p. 208), and **os::NODE< dataType >** (p. 273).

operator->() [1/2]

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

Member access.

Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

operator->() [2/2]

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

Member access.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

randomAccess()

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

Returns if the node can be accessed randomly.

Returns

object::RANDOM_ACCESS

Reimplemented in **os::BOUNDED_QUEUE_NODE< dataType >** (p. 161), **os::VECTOR_NODE< dataType >** (p. 377), **os::HASH_NODE< dataType >** (p. 209), **os::AVL_NODE< dataType >** (p. 130), and **os::LIST_NODE< dataType >** (p. 258).

remove()

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

Remove node from data structure.

Returns

void

Reimplemented in **os::BOUNDED_QUEUE_NODE< dataType >** (p. 162), **os::VECTOR_NODE< dataType >** (p. 377), **os::HASH_NODE< dataType >** (p. 209), **os::AVL_NODE< dataType >** (p. 130), and **os::LIST_NODE< dataType >** (p. 258).

usingComparison()

```
template<class dataType >
virtual bool os::nodeFrame< dataType >::usingComparison ( ) const  [pure virtual]
```

Defines which comparison method is used.

valid()

```
template<class dataType >
virtual bool os::nodeFrame< dataType >::valid ( ) const  [inline], [virtual]
```

Valid data query Checks if the provided data is valid.

Returns

true if valid, else, false

Reimplemented in **os::BOUNDED_QUEUE_NODE< dataType >** (p. 162), **os::VECTOR_NODE< dataType >** (p. 377), **os::HASH_NODE< dataType >** (p. 209), and **os::NODE< dataType >** (p. 273).

12.30.4 Member Data Documentation

ITERABLE

```
template<class dataType >
const bool os::nodeFrame< dataType >::ITERABLE = false  [static]
```

By default, nodeFrames are not iterable.

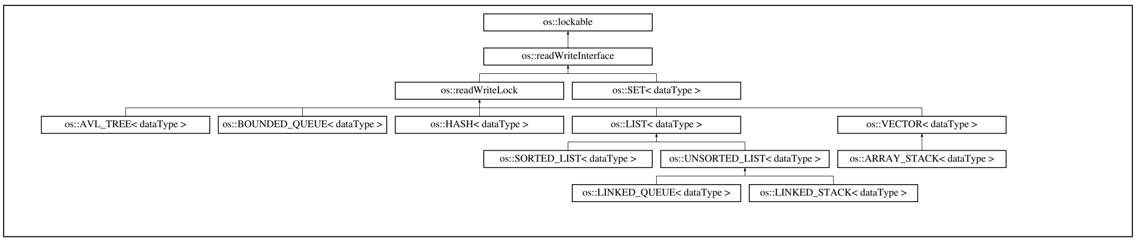
RANDOM_ACCESS

```
template<class dataType >
const bool os::nodeFrame< dataType >::RANDOM_ACCESS = false  [static]
```

No random-access by default.

12.31 os::readWriteInterface Class Reference

Read/write interface.
Inheritance diagram for os::readWriteInterface:



Public Member Functions

- virtual ~**readWriteInterface** () throw (descriptiveException)
  Virtual destructor.
- virtual void **increment** () const =0 throw ()
  Acquires the structure for reading.
- virtual bool **try_increment** () const =0 throw ()
  Attempt to acquire the structure for reading.
- virtual void **decrement** () const =0 throw (descriptiveException)
  Releases the structure after reading.
- void **operator++** () const throw ()
  Calls increment.
- void **operator--** () const throw (descriptiveException)
  Calls decrement.
- void **operator++** (int param) const throw ()
  Calls increment.
- void **operator--** (int param) const throw (descriptiveException)
  Calls decrement.
- void **readLock** () const throw ()
  Calls increment.
- void **attemptReadLock** () const throw ()
  Calls try_increment.
- void **readUnlock** () const throw (descriptiveException)
  Calls decrement.

Additional Inherited Members

12.31.1 Detailed Description

Read/write interface.
Defines an interface to be used by class which need one writer but allow multiple readers.

12.31.2 Constructor & Destructor Documentation

~readWriteInterface()

`virtual os::readWriteInterface::~readWriteInterface ( ) throw` **descriptiveException**`)   [inline], [virtual]`

Virtual destructor.

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

12.31.3 Member Function Documentation

attemptReadLock()

`void os::readWriteInterface::attemptReadLock ( ) const throw )   [inline]`

Calls try_increment.

Returns

void

decrement()

`virtual void os::readWriteInterface::decrement ( ) const throw` **descriptiveException**`)   [pure virtual]`

Releases the structure after reading.

Returns

void

Implemented in **os::SET**< **dataType** > (p. 297), and **os::readWriteLock** (p. 288).

increment()

`virtual void os::readWriteInterface::increment ( ) const throw )   [pure virtual]`

Acquires the structure for reading.

Returns

void

Implemented in **os::SET**< **dataType** > (p. 298), and **os::readWriteLock** (p. 289).

operator++() [1/2]

`void os::readWriteInterface::operator++ ( ) const throw )   [inline]`

Calls increment.

Returns

void

operator++() [2/2]

```
void os::readWriteInterface::operator++ (
              int param ) const throw )   [inline]
```

Calls increment.

Returns

void

operator--() [1/2]

```
void os::readWriteInterface::operator-- ( ) const throw descriptiveException)   [inline]
```

Calls decrement.

Returns

void

operator--() [2/2]

```
void os::readWriteInterface::operator-- (
              int param ) const throw descriptiveException)   [inline]
```

Calls decrement.

Returns

void

readLock()

```
void os::readWriteInterface::readLock ( ) const throw )   [inline]
```

Calls increment.

Returns

void

readUnlock()

```
void os::readWriteInterface::readUnlock ( ) const throw descriptiveException)   [inline]
```

Calls decrement.

Returns

void

try_increment()

`virtual bool os::readWriteInterface::try_increment ( ) const throw )   [pure virtual]`

Attempt to acquire the structure for reading.

Returns

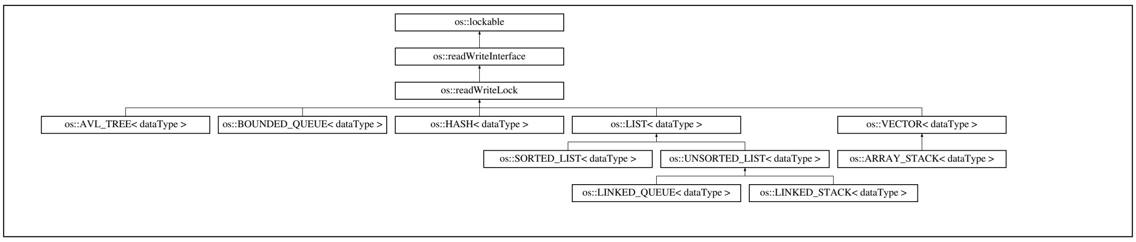
True if acquired, else, false

Implemented in **os::SET**< **dataType** > (p. 301), and **os::readWriteLock** (p. 290).

12.32 os::readWriteLock Class Reference

Read/write lock.

Inheritance diagram for os::readWriteLock:



Public Types

- enum **lockType** { **STANDARD** =0, **RECURSIVE** }

  Enum for read/write lock types.

Public Member Functions

- **readWriteLock** (**readWriteLock::lockType** typ=**readWriteLock::STANDARD**) throw ()

  Read/write lock constructor.
- virtual ~**readWriteLock** () throw (descriptiveException)

  Virtual destructor.
- void **lock** () const final throw ()

  Lock to prevent reads.
- bool **try_lock** () const final throw ()

  Attempt to lock the object.
- void **unlock** () const final throw (descriptiveException)

  Unlock to allow reads.
- void **setMax** (size_t mx) throw (descriptiveException)

  Set maximum number of readers.
- void **increment** () const final throw ()

  Acquires the structure for reading.
- bool **try_increment** () const final throw ()

Attempt to acquire the structure for reading.

- void **decrement** () const final throw (descriptiveException)

  Releases the structure after reading.
- bool **locked** () const throw ()

  Checks if the lock is locked.
- **lockType type** () const throw ()

  Return the type of lock used.
- std::thread::id **lockedThreadID** () const throw ()

  Return the thread which has locked the object.
- size_t **getMax** () const

  Return the maximum number of reader threads.
- size_t **numReaders** () const

  Returns the number of readers.
- size_t **counter** () const

  Returns the number of readers.

Private Member Functions

- **readWriteLock** (const **readWriteLock** &cpy)

  Undefined copy-constructor.

Private Attributes

- **threadCounter lockedThread**

  Holds the thread which last locked.
- **lockType _type**

  The type of lock.
- **basicLock lckb**

  Basic lock used to secure intrinsics.
- **simpleHash**< **threadCounter** > ∗ **_hash**

  Pointer to reader hash bank.
- bool **_attemptingLock**

  Holds writer lock action state.
- bool **_locked**

  Holds writer lock status.
- size_t **_counter**

  Number of threads which have locked.
- size_t **_max**

  Maximum number of readers.

Additional Inherited Members

12.32.1 Detailed Description

Read/write lock.

Allows for multiple threads accessing a thread to read, but only one accessing the thread to write.

12.32.2 Member Enumeration Documentation

lockType

`enum` **os::readWriteLock::lockType**

Enum for read/write lock types.

Enumerator

STANDARD	Use standard mutex rules.
RECURSIVE	Use recursive mutex rules.

12.32.3 Constructor & Destructor Documentation

readWriteLock() [1/2]

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

Undefined copy-constructor.

readWriteLock() [2/2]

`os::readWriteLock::readWriteLock (`
readWriteLock::lockType *typ* `=` **readWriteLock::STANDARD** `) throw )`

Read/write lock constructor.

Constructs a read/write lock, accepting a lock type definition.

Parameters

in	*typ*	Standard lock by default

~readWriteLock()

`virtual os::readWriteLock::~readWriteLock ( ) throw` **descriptiveException**`)   [virtual]`

Virtual destructor.

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

12.32.4 Member Function Documentation

counter()

`size_t os::readWriteLock::counter ( ) const  [inline]`

Returns the number of readers.

Returns

readWriteLock::_counter (p. 291)

decrement()

`void os::readWriteLock::decrement ( ) const throw` **descriptiveException**`)   [final], [virtual]`

Releases the structure after reading.

Returns

void

Implements **os::readWriteInterface** (p. 284).

getMax()

`size_t os::readWriteLock::getMax ( ) const  [inline]`

Return the maximum number of reader threads.

Returns

readWriteLock::_max (p. 291)

increment()

`void os::readWriteLock::increment ( ) const throw )   [final], [virtual]`

Acquires the structure for reading.

Returns

void

Implements **os::readWriteInterface** (p. 284).

lock()

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

Lock to prevent reads.

Returns

void

Implements **os::lockable** (p. 261).

locked()

`bool os::readWriteLock::locked ( ) const throw )   [inline], [virtual]`

Checks if the lock is locked.

Returns

True if locked, else, false

Implements **os::lockable** (p. 261).

lockedThreadID()

`std::thread::id os::readWriteLock::lockedThreadID ( ) const throw ) [inline]`

Return the thread which has locked the object.

Returns

readWriteLock::lockedThread.id()

numReaders()

`size_t os::readWriteLock::numReaders ( ) const`

Returns the number of readers.

Returns

readWriteLock::_counter (p. 291) or **readWriteLock::_hash** (p. 291)->numElements

setMax()

`void os::readWriteLock::setMax (`
`size_t mx ) throw` **descriptiveException**`)`

Set maximum number of readers.

Please note that if the **readWriteLock** (p. 286) is declared to be recursive, this will also set the size of the hash table.

Returns

void

try_increment()

`bool os::readWriteLock::try_increment ( ) const throw ) [final], [virtual]`

Attempt to acquire the structure for reading.

Returns

True if acquired, else, false

Implements **os::readWriteInterface** (p. 285).

try_lock()

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

Attempt to lock the object.

Locks the object if possible, otherwise, returns false.

Returns

True if lock successful

Implements **os::lockable** (p. 261).

type()

lockType `os::readWriteLock::type ( ) const throw )   [inline]`

Return the type of lock used.

Returns

readWriteLock::_type (p. 291)

unlock()

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

Unlock to allow reads.

Returns

void

Implements **os::lockable** (p. 262).

12.32.5 Member Data Documentation

_attemptingLock

`bool os::readWriteLock::_attemptingLock  [mutable], [private]`

Holds writer lock action state.

_counter

`size_t os::readWriteLock::_counter  [mutable], [private]`

Number of threads which have locked.

_hash

simpleHash<**threadCounter**>* `os::readWriteLock::_hash  [mutable], [private]`

Pointer to reader hash bank.

_locked

`bool os::readWriteLock::_locked  [mutable], [private]`

Holds writer lock status.

_max

`size_t os::readWriteLock::_max  [private]`

Maximum number of readers.

_type

lockType `os::readWriteLock::_type  [private]`

The type of lock.

lckb

basicLock `os::readWriteLock::lckb [private]`

Basic lock used to secure intrinsics.

lockedThread

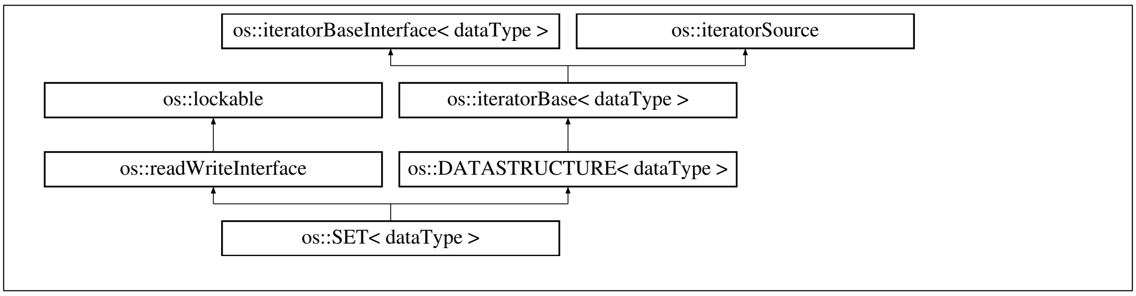
threadCounter `os::readWriteLock::lockedThread [mutable], [private]`

Holds the thread which last locked.

12.33 os::SET< dataType > Class Template Reference

Template AVL Tree definition.

Inheritance diagram for os::SET< dataType >:



Public Types

- enum **type** {
  set =0, **vector**, **sortedList**, **unsortedList**,
  AVLTree, **hash** }

  Type of set Sets can dynamically change their method of storing data.

Public Member Functions

- void **setType** (**SET::type** typ)

  Set type of set.
- **SET** (**SET::type** typ=type::set)

  Construct the set Initializes an set.
- **SET** (const **SET**< dataType > &cpy)

  Copy constructor.
- **SET**< dataType > & **operator=** (const **SET**< dataType > &cpy)

  Assignment operator.
- ~**SET** () final
- void **lock** () const final throw ()

  Locks datastructure.
- void **unlock** () const final throw (descriptiveException)

Unlocks the datastructure.

- bool **locked** () const final throw ()

  Checks if the datastructure is locked.
- bool **try_lock** () const final throw ()

  Attempt to lock the datastructure.
- void **increment** () const final throw ()

  Acquires the datastructure for reading.
- bool **try_increment** () const final throw ()

  Attempt to acquire the datastructure for reading.
- void **decrement** () const final throw (descriptiveException)

  Releases the datastructure after reading.
- size_t **size** () const final

  Access size of the set.
- bool **iterable** () const final

  Returns if the relevant node type is iterable.
- bool **randomAccess** () const final

  Returns if the relevant node type can be accessed randomly.
- bool **insert** (const dataType &x) final

  Insert item into the tree Inserts an item into the tree and procedes to re-balance the tree.
- bool **remove** (const dataType &x) final

  Remove item from the data-structure.
- bool **find** (const dataType &x) const final

  Searches for an item.
- dataType & **access** (const dataType &x) final

  Mutable item access.
- const dataType & **access** (const dataType &x) const final

  Immutable item access.
- dataType & **at** (size_t i) final throw (descriptiveException)

  Access the tree by index.
- const dataType & **at** (size_t i) const final throw (descriptiveException)

  Constant access the tree by index.

Static Public Attributes

- static const bool **ITERABLE** = true

  AVL trees are iterable.
- static const bool **RANDOM_ACCESS** = false

  AVL trees allow random access.

Protected Member Functions

- **smart_ptr**< **nodeFrame**< dataType > > **getFirstNode** () final

  Access to first node.
- **smart_ptr**< **nodeFrame**< dataType > > **getLastNode** () final

  Access to last node.
- const **smart_ptr**< **nodeFrame**< dataType > > **getFirstNodeConst** () const final

  Constant access to first node.
- const **smart_ptr**< **nodeFrame**< dataType > > **getLastNodeConst** () const final

  Constant access to last node.
- **smart_ptr**< **nodeFrame**< dataType > > **searchNode** (const **smart_ptr**< dataType > dt) final

  Search for a node.
- const **smart_ptr**< **nodeFrame**< dataType > > **searchNodeConst** (const **smart_ptr**< data↩Type > dt) const final

  Const search for a node.

Private Attributes

- **smart_ptr**< **DATASTRUCTURE**< dataType > > **_data**

  Datastructure contained in the set Sets can dynamically set what kind of datastructure they are defined by.
- **type _type**

  Type of datastructure Defines the type of datastructure used in this set.

Additional Inherited Members

12.33.1 Detailed Description

template<class dataType>
class os::SET< dataType >

Template AVL Tree definition.

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

12.33.2 Member Enumeration Documentation

type

```
template<class dataType>
enum os::SET::type
```

Type of set Sets can dynamically change their method of storing data.

Enumerator

set	Default type value Defaults to an unsorted list.
vector	Vector type.

Enumerator

sortedList	Sorted list type.
unsortedList	Unsorted list type.
AVLTree	AVL tree type.
hash	Hash table type.

12.33.3 Constructor & Destructor Documentation

SET() [1/2]

```
template<class dataType>
os::SET< dataType >::SET (
              SET< dataType >::type typ = type::set )  [inline]
```

Construct the set Initializes an set.

Parameters

in	*typ*	Type of datastructure defining set

SET() [2/2]

```
template<class dataType>
os::SET< dataType >::SET (
              const SET< dataType > & cpy )  [inline]
```

Copy constructor.

Copies over the entire set. The copied set will used the data-structure of the previous set.

~SET()

```
template<class dataType>
os::SET< dataType >::~SET ( )  [inline], [final]
```

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

12.33.4 Member Function Documentation

access() [1/2]

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

Mutable item access.

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

Returns

Mutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 179).

access() [2/2]

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

Immutable item access.

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

Returns

Immutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 181).

at() [1/2]

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

Access the tree by index.

Parameters

in	*i*	Index of the element

Returns

Reference to ith element of the tree

Reimplemented from **os::DATASTRUCTURE**< **dataType** > (p. 181).

at() [2/2]

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

Constant access the tree by index.

Parameters

in	*i*	Index of the element

Returns

Immutable reference to ith element of the tree

Reimplemented from **os::DATASTRUCTURE**< **dataType** > (p. 182).

decrement()

`template<class dataType>`
`void` **os::SET**`< dataType >::decrement ( ) const throw` **descriptiveException**`) [inline], [final], [virtual]`

Releases the datastructure after reading.

Returns

void

Implements **os::readWriteInterface** (p. 284).

find()

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

Searches for an item.

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

Returns

True if found, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 182).

getFirstNode()

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

Access to first node.

Returns

First node in the set

Implements **os::iteratorBase**< **dataType** > (p. 230).

getFirstNodeConst()

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

Constant access to first node.

Returns

Immutable first node in the set

Implements **os::iteratorBase**< **dataType** > (p. 230).

getLastNode()

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

Access to last node.

Returns

Last node in the set

Implements **os::iteratorBase**< **dataType** > (p. 231).

getLastNodeConst()

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

Constant access to last node.

Returns

Immutable last node in the set

Implements **os::iteratorBase**< **dataType** > (p. 231).

increment()

`template<class dataType>`
void **os::SET**< dataType >::increment ( ) const throw) [inline], [final], [virtual]

Acquires the datastructure for reading.

Returns

void

Implements **os::readWriteInterface** (p. 284).

insert()

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

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

Parameters

in	*x*	Data to be bound

Returns

True if successful, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 182).

iterable()

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

Returns if the relevant node type is iterable.

Returns

true

Reimplemented from **os::iteratorSource** (p. 239).

lock()

```
template<class dataType>
void os::SET< dataType >::lock ( ) const throw )   [inline], [final], [virtual]
```

Locks datastructure.

Returns

void

Implements **os::lockable** (p. 261).

locked()

```
template<class dataType>
bool os::SET< dataType >::locked ( ) const throw )   [inline], [final], [virtual]
```

Checks if the datastructure is locked.

Returns

True if locked, else, false

Implements **os::lockable** (p. 261).

operator=()

```
template<class dataType>
SET<dataType>& os::SET< dataType >::operator= (
              const SET< dataType > & cpy )  [inline]
```

Assignment operator.

Copies over the entire set. The copied set will used the data-structure of the previous set.

randomAccess()

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

Returns if the relevant node type can be accessed randomly.

Returns

true

Reimplemented from **os::iteratorSource** (p. 240).

remove()

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

Remove item from the data-structure.

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

Returns

True if removed, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 184).

searchNode()

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

Search for a node.

Parameters

in	*dt*	Pointer to search for

Returns

Muttable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 232).

searchNodeConst()

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

Const search for a node.

Parameters

in	*dt*	Pointer to search for

Returns

Immutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 232).

setType()

```
template<class dataType>
void os::SET< dataType >::setType (
            SET< dataType >::type typ )  [inline]
```

Set type of set.

Parameters

in	*typ*	Type of datastructure defining set

Returns

void

size()

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

Access size of the set.

Returns

Number of elements in the set

Implements **os::DATASTRUCTURE< dataType >** (p. 184).

try_increment()

```
template<class dataType>
bool os::SET< dataType >::try_increment ( ) const throw )   [inline], [final], [virtual]
```

Attempt to acquire the datastructure for reading.

Returns

True if acquired, else, false

Implements **os::readWriteInterface** (p. 285).

try_lock()

```
template<class dataType>
bool os::SET< dataType >::try_lock ( ) const throw )   [inline], [final], [virtual]
```

Attempt to lock the datastructure.
Locks the datastructure if possible, otherwise, returns false.

Returns

True if lock successful

Implements **os::lockable** (p. 261).

unlock()

template<class dataType>
void **os::SET**< dataType >::unlock ( ) const throw **descriptiveException**) [inline], [final], [virtual]

Unlocks the datastructure.

Returns

void

Implements **os::lockable** (p. 262).

12.33.5 Member Data Documentation

_data

template<class dataType>
smart_ptr<**DATASTRUCTURE**<dataType> > **os::SET**< dataType >::_data [private]

Datastructure contained in the set Sets can dynamically set what kind of datastructure they are defined by.

_type

template<class dataType>
type **os::SET**< dataType >::_type [private]

Type of datastructure Defines the type of datastructure used in this set.

ITERABLE

template<class dataType>
const bool **os::SET**< dataType >::ITERABLE = true [static]

AVL trees are iterable.

RANDOM_ACCESS

template<class dataType>
const bool **os::SET**< dataType >::RANDOM_ACCESS = false [static]

AVL trees allow random access.

12.34 os::simpleHash< dataType > Class Template Reference

Basic hash function.

Public Member Functions

- **simpleHash** (size_t(&hashFunc)(const dataType &) throw()=**simpleHash**< dataType >::**defaultHashFunction**, bool(&compFunc)(const dataType &, const dataType &) throw()=**simpleHash**< dataType >::**defaultCompareFunction**, size_t sz=**DEFAULT_SIZE**) throw (descriptiveException)

    Simple hash constructor Instantiates the simple hash constructor with a hash function and size.
- virtual ~**simpleHash** ()

Virtual destructor.

- size_t **hashFunction** (const dataType &dtp) throw ()

  Call the hash function.
- void **setSize** (size_t sz)

  Attempts to set the size of the hash table.
- size_t **size** () const throw ()

  Return size of hash table.
- size_t **numElements** () const throw ()

  Return the number of elements in the structure.
- dataType & **insert** (const dataType &dtp) throw (descriptiveException)

  Insert data into the table.
- bool **remove** (const dataType &dtp)

  Remove data from the table.
- bool **exists** (const dataType &dtp) const throw ()

  Check if data exists in the table.
- size_t **find** (const dataType &dtp) const throw ()

  Find data in the table.
- void **empty** ()

  Clear the hash table.
- bool **atPosition** (size_t pos) const throw ()

  Check position in the table.
- bool **check** (size_t pos) const throw ()

  Check position in the table.
- dataType & **at** (size_t pos) throw (descriptiveException)

  Access element by index.
- const dataType & **at** (size_t pos) const throw (descriptiveException)

  Const access element by index.
- dataType & **operator[ ]** (size_t pos) throw (descriptiveException)

  Access element by index.
- const dataType & **operator[ ]** (size_t pos) const throw (descriptiveException)

  Const access element by index.

Static Public Member Functions

- static size_t **defaultHashFunction** (const dataType &dtp) throw ()

  Default hash function.
- static bool **defaultCompareFunction** (const dataType &dt1, const dataType &dt2) throw ()

  Default compare function.

Static Public Attributes

- static const size_t **DEFAULT_SIZE** =2048

  Default size of the hash table.

Private Attributes

- bool ∗ **_isTaken**

  Pointer to taken array Exists parallel to the table and informs whether the object in that position in the table is valid.
- dataType ∗ **_table**

  Point to the table Contains the actual data for the hash table.
- size_t **_size**

  Size of the hash table.
- size_t **_numElements**

  Holds the number of elements.
- size_t(&) **_hashFunction** (const dataType &) throw ()

  Reference to the hash function This function is used to convert an object to a position in the hash table.
- bool(&) **_compareFunction** (const dataType &, const dataType &) throw ()

  Reference to the comparison function This function is used to compare two objects in the table.

12.34.1 Detailed Description

template<class dataType>
class os::simpleHash< dataType >

Basic hash function.

This class defines a very simple hash structure to be used by classes which may require use of a hash table.

12.34.2 Constructor & Destructor Documentation

simpleHash()

```
template<class dataType>
os::simpleHash< dataType >::simpleHash (
            size_t(&)(const dataType &) throw hashFunc()  = simpleHash<dataType>::defaultHashFunction,
            bool(&)(const dataType &, const dataType &) throw compFunc()  = simpleHash<dataType>↩
::defaultCompareFunction,
            size_t sz = DEFAULT_SIZE ) throw descriptiveException)   [inline]
```

Simple hash constructor Instantiates the simple hash constructor with a hash function and size.

Parameters

in	*hashFunc*	**os::simpleHash<dataType>::defaultHashFunction** (p. 306) by default
in	*compFunc*	os::simpleCompare<dataType>::defaultCompareFunction by default
in	*sz*	DEFAULT_SIZE by default

~simpleHash()

```
template<class dataType>
```

virtual **os::simpleHash**< dataType >::~**simpleHash** ( ) [inline], [virtual]

Virtual destructor.

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

12.34.3 Member Function Documentation

at() [1/2]

template<class dataType>
dataType& **os::simpleHash**< dataType >::at (
size_t *pos*) throw **descriptiveException**) [inline]

Access element by index.

Parameters

in	*pos*	Position to access

Returns

Reference to object at position

at() [2/2]

template<class dataType>
const dataType& **os::simpleHash**< dataType >::at (
size_t *pos*) const throw **descriptiveException**) [inline]

Const access element by index.

Parameters

in	*pos*	Position to access

Returns

Constant reference to object at position

atPosition()

template<class dataType>
bool **os::simpleHash**< dataType >::atPosition (
size_t *pos*) const throw) [inline]

Check position in the table.

Parameters

in	*pos*	Position to check

Returns

true if occupied, else, false

check()

```
template<class dataType>
bool os::simpleHash< dataType >::check (
             size_t pos ) const throw )   [inline]
```

Check position in the table.

Parameters

in	*pos*	Position to check

Returns

true if occupied, else, false

defaultCompareFunction()

```
template<class dataType>
static bool os::simpleHash< dataType >::defaultCompareFunction (
             const dataType & dt1,
             const dataType & dt2 ) throw )   [inline], [static]
```

Default compare function.

Compares two objects using the defined equality operator. Note that for some data types, this may need to be re-defined.

Parameters

in	*dt1*	Data to be compared
in	*dt2*	Data to be compared

Returns

True if equal, else, false

defaultHashFunction()

```
template<class dataType>
static size_t os::simpleHash< dataType >::defaultHashFunction (
             const dataType & dtp ) throw )   [inline], [static]
```

Default hash function.

Casts an object to size_t. This allows the hash table to use the size representation of an element to place it in the hash table.

Parameters

in	*dt*	Data type to be cast

Returns

Data type cast to size

empty()

```
template<class dataType>
void os::simpleHash< dataType >::empty ( )  [inline]
```

Clear the hash table.

Returns

void

exists()

```
template<class dataType>
bool os::simpleHash< dataType >::exists (
            const dataType & dtp ) const throw )   [inline]
```

Check if data exists in the table.

This function should have O(1), although, if the table is sufficiently full, this function approaches O(n) where n is the size of the table.

Parameters

in	*dtp*	Data to be checked

Returns

True if found, else, false

find()

```
template<class dataType>
size_t os::simpleHash< dataType >::find (
            const dataType & dtp ) const throw )   [inline]
```

Find data in the table.

This function should have O(1), although, if the table is sufficiently full, this function approaches O(n) where n is the size of the table.

Parameters

in	*dtp*	Data to be checked

Returns

Position in the hash

hashFunction()

```
template<class dataType>
size_t os::simpleHash< dataType >::hashFunction (
              const dataType & dtp ) throw )   [inline]
```

Call the hash function.

Parameters

in	*dtp*	Data type to be hashed

Returns

simpleHash::_hashFunction(dtp)

insert()

```
template<class dataType>
dataType& os::simpleHash< dataType >::insert (
              const dataType & dtp ) throw descriptiveException)   [inline]
```

Insert data into the table.

This function should have O(1), although, if the table is sufficiently full, this function approaches O(n) where n is the size of the table.

Parameters

in	*dtp*	Data to be inserted

Returns

void

numElements()

```
template<class dataType>
size_t os::simpleHash< dataType >::numElements ( ) const throw )   [inline]
```

Return the number of elements in the structure.

Returns

simpleHash::_numElements (p. 310)

operator[]() [1/2]

```
template<class dataType>
dataType& os::simpleHash< dataType >::operator[] (
              size_t pos ) throw descriptiveException)   [inline]
```

Access element by index.

Parameters

in	*pos*	Position to access

Returns

Reference to object at position

operator[]() [2/2]

```
template<class dataType>
const dataType& os::simpleHash< dataType >::operator[] (
              size_t pos ) const throw descriptiveException)   [inline]
```

Const access element by index.

Parameters

in	*pos*	Position to access

Returns

Constant reference to object at position

remove()

```
template<class dataType>
bool os::simpleHash< dataType >::remove (
              const dataType & dtp )   [inline]
```

Remove data from the table.

This function should have O(1), although, if the table is sufficiently full, this function approaches O(n) where n is the size of the table.

Parameters

in	*dtp*	Data to be removed

Returns

True if deleted, else, false

setSize()

```
template<class dataType>
void os::simpleHash< dataType >::setSize (
            size_t sz )  [inline]
```

Attempts to set the size of the hash table.

Sets the size of the hash table and re-inserts all data. Note that this function will fail if the size is out of bounds or if the new size cannot fit all of the old data.

Parameters

in	*sz*	Target size

Returns

void

size()

```
template<class dataType>
size_t os::simpleHash< dataType >::size ( ) const throw )  [inline]
```

Return size of hash table.

Returns

simpleHash::_size (p. 311)

12.34.4 Member Data Documentation

_compareFunction

```
template<class dataType>
bool(&) os::simpleHash< dataType >::_compareFunction(const dataType &, const dataType &) throw )  [private]
```

Reference to the comparison function This function is used to compare two objects in the table.

_hashFunction

```
template<class dataType>
size_t(&) os::simpleHash< dataType >::_hashFunction(const dataType &) throw )  [private]
```

Reference to the hash function This function is used to convert an object to a position in the hash table.

_isTaken

```
template<class dataType>
bool* os::simpleHash< dataType >::_isTaken  [private]
```

Pointer to taken array Exists parallel to the table and informs whether the object in that position in the table is valid.

_numElements

template<class dataType>
size_t **os::simpleHash**< dataType >::_numElements [private]

Holds the number of elements.

_size

template<class dataType>
size_t **os::simpleHash**< dataType >::_size [private]

Size of the hash table.

_table

template<class dataType>
dataType* **os::simpleHash**< dataType >::_table [private]

Point to the table Contains the actual data for the hash table.

DEFAULT_SIZE

template<class dataType>
const size_t **os::simpleHash**< dataType >::DEFAULT_SIZE =2048 [static]

Default size of the hash table.

12.35 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< size_t > *rc, const dataType *rp, const **void_rec** f) throw ()

  Forced constructor.
- **smart_ptr** (const **smart_ptr**< dataType > &sp) throw ()

  Copy constructor.
- **smart_ptr** (const dataType *rp, **smart_pointer_type** typ=**raw_type**) throw ()

  Standard constructor.
- **smart_ptr** (const dataType *rp, const **void_rec** destructor) throw ()

  Dynamic deletion constructor.
- virtual ~**smart_ptr** ()

  Virtual destructor.
- **smart_ptr** (const int rp) throw ()

  Integer constructor.
- **smart_ptr** (const long rp) throw ()

Long constructor.

- **smart_ptr** (const unsigned long rp) throw ()

  Unsigned long constructor.
- **smart_pointer_type getType** () const throw ()

  Return type.
- dataType ∗ **get** () throw ()

  Return data.
- const dataType ∗ **get** () const throw ()

  Return constant data.
- const dataType ∗ **constGet** () const throw ()

  Return constant data.
- const std::atomic< size_t > ∗ **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[ ]** (size_t i) throw ()

  Array de-reference.
- const dataType & **operator[ ]** (size_t 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. 311).*
- 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< size_t > ∗ **ref_count**
  Reference count.
- dataType ∗ **raw_ptr**
  Pointer to data.
- **void_rec func**
  Non-standard deletion.

12.35.1 Detailed Description

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

Reference counted pointer.

The **os::smart_ptr** (p. 311) template class allows for automatic memory management. **os::smart_ptr** (p. 311)'s have a type defined by **os::smart_pointer_type** (p. 112) which defines the copy and deletion behaviour of the object.

12.35.2 Constructor & Destructor Documentation

smart_ptr() [1/8]

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

Default constructor.

Constructs an **os::smart_ptr** (p. 311) of type **os::null_type** (p. 112). All private data is set to 0 or NULL.

smart_ptr() [2/8]

```
template<class dataType>
os::smart_ptr< dataType >::smart_ptr (
        const smart_pointer_type t,
        const std::atomic< size_t > * rc,
        const dataType * rp,
        const void_rec f ) throw )   [inline]
```

Forced constructor.

Constructs an **os::smart_ptr** (p. 311) explicitly from each of the parameters provided. This constructor is primarily used for testing purposes.

Parameters

in	*t*	Type definition for the object
in,out	*rp*	Pointer to the reference count
in	*rp*	Raw pointer object is managing
in	*f*	Dynamic deletion function

smart_ptr() [3/8]

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

Copy constructor.

Constructs an **os::smart_ptr** (p. 311) from an existing **os::smart_ptr** (p. 311). Will increment the reference count as defined by the received **os::smart_pointer_type** (p. 112).

Parameters

in,out	*sp*	Reference to data being copied

smart_ptr() [4/8]

```
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. 311) from a raw pointer and a type. This is the most commonly used **os::smart_ptr** (p. 311) constructor, other than the copy constructor. Note that **os::shared_type_dynamic_delete** (p. 113) cannot be constructed through this method.

Parameters

in	*rp*	Raw pointer object is managing

Parameters

in	*typ*	Defines reference count behaviour

smart_ptr() [5/8]

```
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. 311) from a raw pointer and a destruction function. This constructor generates an **os::smart_ptr** (p. 311) of type **os::shared_type_dynamic_delete** (p. 113).

Parameters

in	*rp*	Raw pointer object is managing
in	*destructor*	Defines the function to be executed on destroy

∼smart_ptr()

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

Virtual destructor.

Calls **os::smart_ptr<dataType>::teardown()** (p. 323) before destroying the object.

smart_ptr() [6/8]

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

Integer constructor.

Constructs an **os::smart_ptr** (p. 311) 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	*rp*	Integer cast to raw pointer

smart_ptr() [7/8]

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

Long constructor.

Constructs an **os::smart_ptr** (p. 311) 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	*rp*	Long cast to raw pointer

smart_ptr() [8/8]

```
template<class dataType>
os::smart_ptr< dataType >::smart_ptr (
            const unsigned long rp ) throw )   [inline]
```

Unsigned long constructor.

Constructs an **os::smart_ptr** (p. 311) 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	*rp*	Unsigned long cast to raw pointer

12.35.3 Member Function Documentation

bind() [1/2]

```
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. 311) from an existing **os::smart_ptr** (p. 311). Will increment the reference count as defined by the received **os::smart_pointer_type** (p. 112).

Parameters

in	*sp*	Reference to data being copied

Returns

Reference to self

bind() [2/2]

```
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. 311) from a dataType pointer. This new **os::smart_ptr** (p. 311) will be of type **os::raw_type** (p. 112) unless the dataType pointer is NULL, then it will be of type **os::null_type** (p. 112).

Parameters

in	*rp*	Reference to dataType pointer

Returns

Reference to self

compare() [1/3]

`template<class dataType>`
`int os::smart_ptr< dataType >::compare ( const smart_ptr< dataType > & c ) const throw ) [inline]`

Compare **os::smart_ptr** (p. 311).

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

Parameters

in	*c*	os::smart_ptr<dataType>

Returns

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

compare() [2/3]

`template<class dataType>`
`int os::smart_ptr< dataType >::compare ( const dataType * c ) const throw ) [inline]`

Compare raw pointers.

Compares a os::smart_ptr<dataType> and a raw pointer of type dataType and returns the result in the form of a 1,0 or -1.

Parameters

in	*c*	Raw dataType pointer

Returns

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

compare() [3/3]

```
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)

constGet()

```
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. 311).

Returns

dataType∗ in constant form, **os::smart_ptr**<**dataType**>**::raw_ptr** (p. 323)

get() [1/2]

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

Return data.

Returns the dataType pointer of the **os::smart_ptr** (p. 311).

Returns

dataType∗ in modifiable form, **os::smart_ptr**<**dataType**>**::raw_ptr** (p. 323)

get() [2/2]

```
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. 311).

Returns

dataType* in constant form, **os::smart_ptr<dataType>::raw_ptr** (p. 323)

getFunc()

`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. 113) mode)

Returns

os::void_rec (p. 112) **os::smart_ptr<dataType>::func** (p. 323)

getRefCount()

`template<class dataType>`
`const std::atomic<size_t>*` **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. 324)

getType()

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

Return type.

Returns the **os::smart_pointer_type** (p. 112) of the **os::smart_ptr** (p. 311).

Returns

os::smart_pointer_type (p. 112) **os::smart_ptr<dataType>::type** (p. 324)

operator bool()

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

Boolean conversion.

Returns

os::smart_ptr<dataType>::raw_ptr (p. 323) cast to boolean

operator long()

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

long conversion

Returns

os::smart_ptr<dataType>::raw_ptr (p. 323) cast to long

operator size_t()

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

size_t conversion

Returns

os::smart_ptr<dataType>::raw_ptr (p. 323) cast to size_t

operator"!()

```
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. 323) cast to boolean

operator*() [1/2]

```
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. 323) de-referenced

operator*() [2/2]

```
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. 323) de-referenced

operator->() [1/2]

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

Pointer pass.

Returns

os::smart_ptr<dataType>::raw_ptr (p. 323)

operator->() [2/2]

```
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. 323)

operator=() [1/5]

```
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. 316).

Parameters

in	*source*	Reference to data being copied

Returns

Reference to self

operator=() [2/5]

```
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. 316).

Parameters

in	*source*	Reference to dataType pointer

Returns

Reference to self

operator=() [3/5]

```
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. 316) with the integer cast to a dataType pointer.

Parameters

in	*source*	Integer cast to raw pointer

Returns

Reference to self

operator=() [4/5]

```
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. 316) with the long cast to a dataType pointer.

Parameters

in	*source*	Long cast to raw pointer

Returns

Reference to self

operator=() [5/5]

```
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. 316) with the unsigned long cast to a dataType pointer.

Parameters

in	*source*	Unsigned long cast to raw pointer

Returns

Reference to self

operator[]() [1/2]

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

Array de-reference.

Returns

dataType reference of **os::smart_ptr<dataType>::raw_ptr** (p. 323) incremented i de-referenced

operator[]() [2/2]

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

Constant array de-reference.

Returns

Constant dataType reference of **os::smart_ptr<dataType>::raw_ptr** (p. 323) incremented i de-referenced

teardown()

```
template<class dataType>
void os::smart_ptr< dataType >::teardown ( ) throw descriptiveException)   [inline], [private]
```

Delete data.

Tears down the **os::smart_ptr** (p. 311). Decrements the reference counter, if not of **os::raw_type** (p. 112) or **os::null_type** (p. 112), and delete **os::smart_ptr<dataType>::raw_ptr** (p. 323) if needed. Note that if **os::smart_ptr<dataType>::raw_ptr** (p. 323) is deleted, so is **os::smart_ptr<dataType>::ref_count** (p. 324).

Returns

void

12.35.4 Member Data Documentation

func

```
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. 311) is of type **os::shared_type_dynamic_delete** (p. 113).

raw_ptr

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

Pointer to data.

The **os::smart_ptr<dataType>::raw_ptr** (p. 323) holds the pointer to the block of memory to be managed by the **os::smart_ptr** (p. 311). If this pointer is NULL, the **os::smart_ptr** (p. 311) is of type **os::null_type** (p. 112).

ref_count

`template<class dataType>`
`std::atomic<size_t>* `**os::smart_ptr**`< dataType >::ref_count  [private]`

Reference count.

This pointer stores the current reference count of the **os::smart_ptr** (p. 311). Note that all **os::smart_ptr** (p. 311)'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. 311) is either of type **os::null_type** (p. 112) or **os::raw_type** (p. 112).

type

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

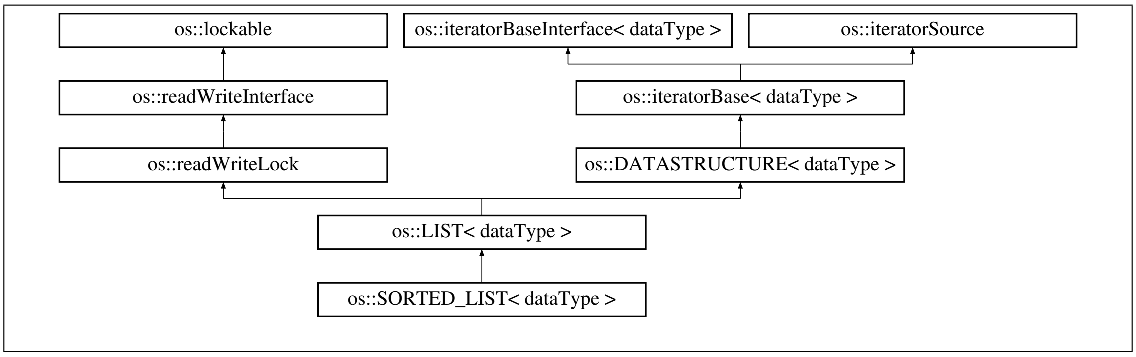
Stores the type.

Defines the type of the **os::smart_ptr** (p. 311). See **os::smart_pointer_type** (p. 112) for details on the available types.

12.36 os::SORTED_LIST< dataType > Class Template Reference

Sorted list A basic list which remains unsorted unless the sort function is called on it.

Inheritance diagram for os::SORTED_LIST< dataType >:



Public Member Functions

- **SORTED_LIST** ()

  Default constructor Constructs an empty sorted list.

- **SORTED_LIST** (const **SORTED_LIST**< dataType > &cpy)

  Copy constructor.
- **SORTED_LIST**< dataType > & **operator=** (const **SORTED_LIST**< dataType > &cpy)

  Assignment operator.
- bool **insert** (const dataType &x) final

  Insert element into the list.

Static Public Attributes

- static const bool **ITERABLE** = true

  Lists are iterable.
- static const bool **RANDOM_ACCESS** = false

  Lists do not allow random access.

Additional Inherited Members

12.36.1 Detailed Description

template<class dataType>
class os::SORTED_LIST< dataType >

Sorted list A basic list which remains unsorted unless the sort function is called on it.

12.36.2 Constructor & Destructor Documentation

SORTED_LIST() [1/2]

template<class dataType>
os::SORTED_LIST< dataType >::**SORTED_LIST** ( ) [inline]

Default constructor Constructs an empty sorted list.

SORTED_LIST() [2/2]

template<class dataType>
os::SORTED_LIST< dataType >::**SORTED_LIST** (
const **SORTED_LIST**< dataType > & *cpy*) [inline]

Copy constructor.

This constructor builds a sorted list from another sorted list. Note that this copies by value, not reference.

Parameters

in	*cpy*	Target to be copied

12.36.3 Member Function Documentation

insert()

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

Insert element into the list.

Parameters

in	*x*	Data type to be inserted

Returns

true if inserted, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 182).

operator=()

```
template<class dataType>
SORTED_LIST<dataType>& os::SORTED_LIST< dataType >::operator= (
            const SORTED_LIST< dataType > & cpy ) [inline]
```

Assignment operator.

This function builds assigns a sorted list from another sorted list. Note that this copies by value, not reference.

Parameters

in	*cpy*	Target to be copied

12.36.4 Member Data Documentation

ITERABLE

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

Lists are iterable.

RANDOM_ACCESS

```
template<class dataType>
const bool os::SORTED_LIST< dataType >::RANDOM_ACCESS = false [static]
```

Lists do not allow random access.

12.37 os::threadCounter Class Reference

Thread counter.

Public Member Functions

- **threadCounter** (std::thread::id thr=std::this_thread::get_id()) throw ()
  Thread ID constructor.
- **threadCounter** (const **threadCounter** &tcnt) throw ()
- std::thread::id **id** () const throw ()
  Access thread ID.
- size_t **count** () const throw ()
  Access thread counter.
- **operator size_t** () const
  Cast object to size_t Used to cast the ***threadCounter*** *(*p. *326)* *to a size_t for use in a simple hash table.*
- **threadCounter** & **operator++** () throw (descriptiveException)
  Increment operator.
- **threadCounter operator++** (int dummy) throw (descriptiveException)
  Increment operator.
- **threadCounter** & **operator--** () throw (descriptiveException)
  Decrement operator.
- **threadCounter operator--** (int dummy) throw (descriptiveException)
  Decrement operator.
- int **compare** (const **threadCounter** &tcnt) const
  Compare function.

Static Public Attributes

- static std::hash< std::thread::id > **hashFunction**
  Thread ID hash function.

Private Attributes

- std::thread::id **_thread**
  Thread ID.
- size_t **_count**
  Thread counter.

12.37.1 Detailed Description

Thread counter.

The thread counter is used by various mutexes to keep track of the thread ID's of threads which have acquired a lock.

12.37.2 Constructor & Destructor Documentation

threadCounter() [1/2]

```
os::threadCounter::threadCounter (
        std::thread::id thr = std::this_thread::get_id() ) throw )
```

Thread ID constructor.

Parameters

in	*thr*	Thread ID to initialize with

threadCounter() [2/2]

```
os::threadCounter::threadCounter (
            const threadCounter & tcnt ) throw )
```

Parameters

in	*tcnt*	Object to copy

12.37.3 Member Function Documentation

compare()

```
int os::threadCounter::compare (
            const threadCounter & tcnt ) const  [inline]
```

Compare function.

This comparison function is used in the comparison operators defined in the COMPARE_OPE←RATORS macro

Parameters

in	*tcnt*	Thread counter to compare against

Returns

Integer representing comparison

count()

```
size_t os::threadCounter::count ( ) const throw )   [inline]
```

Access thread counter.

Returns

threadCounter::_count (p. 330)

id()

```
std::thread::id os::threadCounter::id ( ) const throw )   [inline]
```

Access thread ID.

Returns

threadCounter::_thread (p. 330)

operator size_t()

`os::threadCounter::operator size_t ( ) const [inline]`

Cast object to size_t Used to cast the **threadCounter** (p. 326) to a size_t for use in a simple hash table.

operator++() [1/2]

threadCounter& `os::threadCounter::operator++ ( ) throw` **descriptiveException**)

Increment operator.

Returns

++this

operator++() [2/2]

threadCounter `os::threadCounter::operator++ (`
`int` *dummy* `) throw` **descriptiveException**)

Increment operator.

Returns

this++

operator--() [1/2]

threadCounter& `os::threadCounter::operator-- ( ) throw` **descriptiveException**)

Decrement operator.

Returns

–this

operator--() [2/2]

threadCounter `os::threadCounter::operator-- (`
`int` *dummy* `) throw` **descriptiveException**)

Decrement operator.

Returns

this–

12.37.4 Member Data Documentation

_count

`size_t os::threadCounter::_count [private]`

Thread counter.

_thread

```
std::thread::id os::threadCounter::_thread  [private]
```

Thread ID.

hashFunction

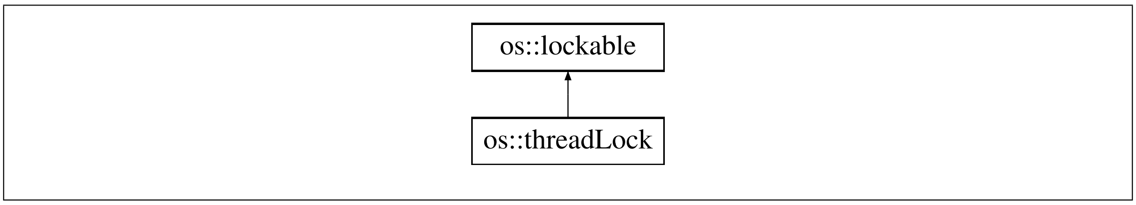
```
std::hash<std::thread::id> os::threadCounter::hashFunction  [static]
```

Thread ID hash function.

12.38 os::threadLock Class Reference

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

Inheritance diagram for os::threadLock:



Public Member Functions

- **threadLock** (bool dMode=**NO_LOCK_CHECK**) throw ()

  Default constructor.
- virtual ~**threadLock** () throw (descriptiveException)

  Virtual destructor.
- void **lock** () const final throw ()

  Locks the std::recursive_mutex.
- void **unlock** () const final throw (descriptiveException)

  Unlocks the std::recursive_mutex.
- bool **locked** () const final throw ()

  Checks if the lock is locked.
- bool **try_lock** () const final throw ()

  Attempt to lock the object.

Private Member Functions

- **threadLock** (const **threadLock** &cpy)

  Undefined copy-constructor.

Private Attributes

- std::recursive_mutex **_mtx**

  Mutable recursive mutex.
- size_t **_lockedStatus**

  Locked flag.

Additional Inherited Members

12.38.1 Detailed Description

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

12.38.2 Constructor & Destructor Documentation

threadLock() [1/2]

```
os::threadLock::threadLock (
            const threadLock & cpy )  [inline], [private]
```

Undefined copy-constructor.

threadLock() [2/2]

```
os::threadLock::threadLock (
            bool dMode = NO_LOCK_CHECK ) throw )
```

Default constructor.

~threadLock()

```
virtual os::threadLock::~threadLock ( ) throw descriptiveException)   [virtual]
```

Virtual destructor.

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

12.38.3 Member Function Documentation

lock()

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

Locks the std::recursive_mutex.

Returns

void

Implements **os::lockable** (p. 261).

locked()

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

Checks if the lock is locked.

Returns

True if locked, else, false

Implements **os::lockable** (p. 261).

try_lock()

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

Attempt to lock the object.
Locks the object if possible, otherwise, returns false.

Returns

True if lock successful

Implements **os::lockable** (p. 261).

unlock()

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

Unlocks the std::recursive_mutex.

Returns

void

Implements **os::lockable** (p. 262).

12.38.4 Member Data Documentation

_lockedStatus

`size_t os::threadLock::_lockedStatus  [mutable], [private]`

Locked flag.

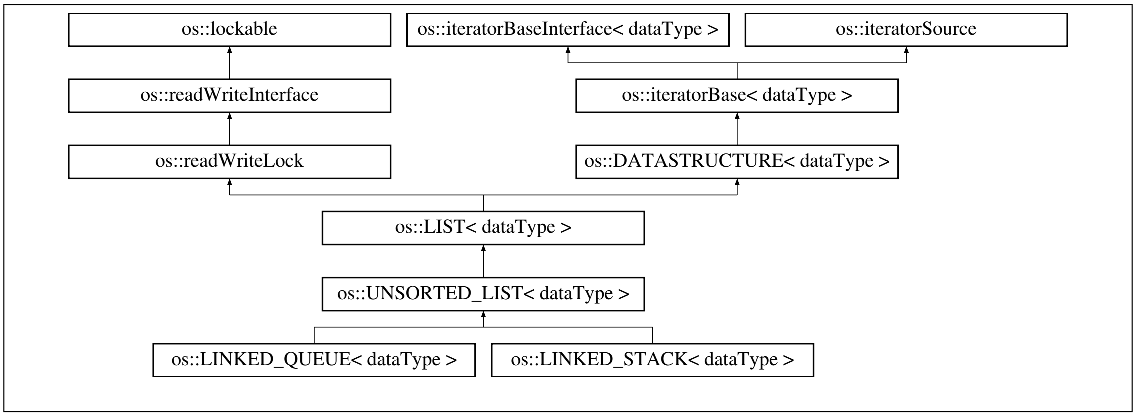
_mtx

`std::recursive_mutex os::threadLock::_mtx  [mutable], [private]`

Mutable recursive mutex.

12.39 os::UNSORTED_LIST< dataType > Class Template Reference

Unsorted list A basic list which remains unsorted unless the sort function is called on it.
Inheritance diagram for os::UNSORTED_LIST< dataType >:



Public Member Functions

- **UNSORTED_LIST** ()

  Default constructor Constructs an empty sorted list.
- **UNSORTED_LIST** (const **UNSORTED_LIST**< dataType > &cpy)

  Copy constructor.
- **UNSORTED_LIST**< dataType > & **operator=** (const **UNSORTED_LIST**< dataType > &cpy)

  Assignment operator.
- bool **insert** (const dataType &x) final

  Insert element into the list.

Static Public Attributes

- static const bool **ITERABLE** = true

  Lists are iterable.
- static const bool **RANDOM_ACCESS** = false

  Lists do not allow random access.

Protected Member Functions

- **UNSORTED_LIST** (const **UNSORTED_LIST**< dataType > ∗cpy)

  Protected copy constructor.

Additional Inherited Members

12.39.1 Detailed Description

template<class dataType>
class os::UNSORTED_LIST< dataType >

Unsorted list A basic list which remains unsorted unless the sort function is called on it.

12.39.2 Constructor & Destructor Documentation

UNSORTED_LIST() [1/3]

```
template<class dataType>
os::UNSORTED_LIST< dataType >::UNSORTED_LIST (
            const UNSORTED_LIST< dataType > * cpy )  [inline], [protected]
```

Protected copy constructor.

This constructor builds a sorted list from another sorted list. Note that this copies by value, not reference.

Parameters

in	*cpy*	Target to be copied

UNSORTED_LIST() [2/3]

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

Default constructor Constructs an empty sorted list.

UNSORTED_LIST() [3/3]

```
template<class dataType>
os::UNSORTED_LIST< dataType >::UNSORTED_LIST (
            const UNSORTED_LIST< dataType > & cpy )  [inline]
```

Copy constructor.

This constructor builds a sorted list from another sorted list. Note that this copies by value, not reference.

Parameters

in	*cpy*	Target to be copied

12.39.3 Member Function Documentation

insert()

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

Insert element into the list.

Parameters

in	*x*	Data type to be inserted

Returns

true if inserted, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 182).

operator=()

```
template<class dataType>
UNSORTED_LIST<dataType>& os::UNSORTED_LIST< dataType >::operator= (
            const UNSORTED_LIST< dataType > & cpy ) [inline]
```

Assignment operator.

This function builds assigns a sorted list from another sorted list. Note that this copies by value, not reference.

Parameters

in	*cpy*	Target to be copied

12.39.4 Member Data Documentation

ITERABLE

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

Lists are iterable.

RANDOM_ACCESS

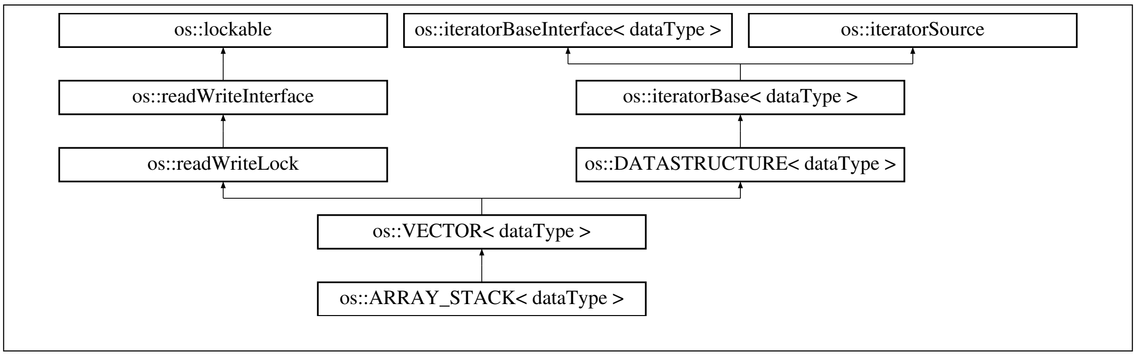
```
template<class dataType>
const bool os::UNSORTED_LIST< dataType >::RANDOM_ACCESS = false [static]
```

Lists do not allow random access.

12.40 os::VECTOR< dataType > Class Template Reference

Template vector definition.

Inheritance diagram for os::VECTOR< dataType >:



Public Member Functions

- **VECTOR** (size_t sz=0)

    Default constructor.
- **VECTOR** (const **VECTOR**< dataType > &cpy)

    Copy constructor.
- ~**VECTOR** ()

    Vector destructor Destroys all objects held in the array. Note that in the case the vector is holding references, the pointers will be destroyed instead of the objects.
- void **resize** (size_t sz)

    Resizes the entire vector.
- bool **insert** (const dataType &x) final

    Insert item into the vector.
- bool **remove** (const dataType &x) final

    Remove item from the data-structure.
- bool **find** (const dataType &x) const final

    Searches for an item.
- dataType & **access** (const dataType &x) final

    Mutable item access.
- const dataType & **access** (const dataType &x) const final

    Immutable item access.
- dataType & **at** (size_t i) final throw (descriptiveException)

    Access the vector by index.
- const dataType & **at** (size_t i) const final throw (descriptiveException)

    Access the vector by index.
- void **sort** (int(∗sort_comparison)(const dataType &, const dataType &)=&**defaultCompare**< dataType >)

    Sorts the vector.
- size_t **size** () const final

    Access size of the vector.
- bool **iterable** () const final

    Returns if the relevant node type is iterable.
- bool **randomAccess** () const final

    Returns if the relevant node type can be accessed randomly.

Static Public Attributes

- static const bool **ITERABLE** = true

  Vectors are iterable.
- static const bool **RANDOM_ACCESS** = true

  Vectors allow random access.

Protected Member Functions

- **smart_ptr**< **nodeFrame**< dataType > > **getFirstNode** () final

  Access to first node.
- **smart_ptr**< **nodeFrame**< dataType > > **getLastNode** () final

  Access to last node.
- const **smart_ptr**< **nodeFrame**< dataType > > **getFirstNodeConst** () const final

  Constant access to first node.
- const **smart_ptr**< **nodeFrame**< dataType > > **getLastNodeConst** () const final

  Constant access to last node.
- **smart_ptr**< **nodeFrame**< dataType > > **searchNode** (const **smart_ptr**< dataType > dt) final

  Search for a node.
- const **smart_ptr**< **nodeFrame**< dataType > > **searchNodeConst** (const **smart_ptr**< data↩ Type > dt) const final

  Const search for a node.

Private Attributes

- dataType ∗ **_array**

  Pointer to array of data.
- size_t **_arraySize**

  Size of the available memory.
- size_t **vecSize**

  Number of elements in the vector.

Additional Inherited Members

12.40.1 Detailed Description

template<class dataType>
class os::VECTOR< dataType >

Template vector definition.

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

12.40.2 Constructor & Destructor Documentation

VECTOR() [1/2]

```
template<class dataType>
os::VECTOR< dataType >::VECTOR (
            size_t sz = 0 )  [inline]
```

Default constructor.

This constructor builds the vecotr with a certain size. Note that the vector will always be of, at least, size 16.

Parameters

in	*sz*	Target size of vector

VECTOR() [2/2]

```
template<class dataType>
os::VECTOR< dataType >::VECTOR (
            const VECTOR< dataType > & cpy )  [inline]
```

Copy constructor.

This constructor builds a vector from another vector. Note that this copies by value, not reference.

Parameters

in	*cpy*	Target to be copied

~VECTOR()

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

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

12.40.3 Member Function Documentation

access() [1/2]

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

Mutable item access.

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

Returns

Mutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 179).

access() [2/2]

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

Immutable item access.

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

Returns

Immutable item equal to x

Implements **os::DATASTRUCTURE**< **dataType** > (p. 181).

at() [1/2]

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

Access the vector by index.

Parameters

in	*i*	Index of the array

Returns

Reference to ith element of the vector

Reimplemented from **os::DATASTRUCTURE**< **dataType** > (p. 181).

at() [2/2]

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

Access the vector by index.

Parameters

in	*i*	Index of the array

Returns

Immutable reference to ith element of the vector

Reimplemented from **os::DATASTRUCTURE**< **dataType** > (p. 182).

find()

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

Searches for an item.

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

Returns

True if found, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 182).

getFirstNode()

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

Access to first node.

Returns

First node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 230).

getFirstNodeConst()

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

Constant access to first node.

Returns

Immutable first node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 230).

getLastNode()

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

Access to last node.

Returns

Last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 231).

getLastNodeConst()

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

Constant access to last node.

Returns

Immutable last node in the structure

Implements **os::iteratorBase**< **dataType** > (p. 231).

insert()

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

Insert item into the vector.

Inserts an item at the end of the vector. Note that different forms of the vector accept pointers instead of object references.

Returns

True

Implements **os::DATASTRUCTURE**< **dataType** > (p. 182).

iterable()

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

Returns if the relevant node type is iterable.

Returns

true

Reimplemented from **os::iteratorSource** (p. 239).

randomAccess()

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

Returns if the relevant node type can be accessed randomly.

Returns

true

Reimplemented from **os::iteratorSource** (p. 240).

remove()

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

Remove item from the data-structure.

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

Returns

True if removed, else, false

Implements **os::DATASTRUCTURE**< **dataType** > (p. 184).

resize()

```
template<class dataType>
void os::VECTOR< dataType >::resize (
            size_t sz ) [inline]
```

Resizes the entire vector.

This operation sets the vector size. Note that this operation will remove any extra nodes.

Parameters

in	*sz*	Target size of vector

Returns

void

searchNode()

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

Search for a node.

Parameters

in	*dt*	Pointer to search for

Returns

Muttable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 232).

searchNodeConst()

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

Const search for a node.

Parameters

in	*dt*	Pointer to search for

Returns

Immutable found node, if applicable

Implements **os::iteratorBase**< **dataType** > (p. 232).

size()

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

Access size of the vector.

Returns

Number of elements in the vector

Implements **os::DATASTRUCTURE**< **dataType** > (p. 184).

sort()

```
template<class dataType>
void os::VECTOR< dataType >::sort (
            int(*)(const dataType &, const dataType &) sort_comparison = &defaultCompare<dataType>
)  [inline]
```

Sorts the vector.

Parameters

in	*sort_comparison*	Comparison function

Returns

void

12.40.4 Member Data Documentation

_array

```
template<class dataType>
dataType* os::VECTOR< dataType >::_array  [private]
```

Pointer to array of data.

_arraySize

```
template<class dataType>
size_t os::VECTOR< dataType >::_arraySize [private]
```

Size of the available memory.

ITERABLE

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

Vectors are iterable.

RANDOM_ACCESS

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

Vectors allow random access.

vecSize

```
template<class dataType>
size_t os::VECTOR< dataType >::vecSize [private]
```

Number of elements in the vector.

12.41 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.
- **operator size_t** () const

  Used for hash tables.
- int **compare** (const **vector2d**< dataType > &vec) const

  Compares two vectors.
- 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.

12.41.1 Detailed Description

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

2-dimensional vector

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

12.41.2 Constructor & Destructor Documentation

vector2d() [1/3]

```
template<class dataType>
os::vector2d< dataType >::vector2d ( ) [inline]
```

Default constructor.

Constructs a 2 dimensional vector with x and y as 0.

vector2d() [2/3]

```
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

vector2d() [3/3]

```
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

~vector2d()

```
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.

12.41.3 Member Function Documentation

add()

```
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

addSelf()

```
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

compare()

```
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 that 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

dotProduct()

```
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

length()

```
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

operator size_t()

```
template<class dataType>
os::vector2d< dataType >::operator size_t ( ) const  [inline]
```

Used for hash tables.

Returns

Vector converted to size_t

operator"!=()

```
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

operator()()

```
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

operator+()

```
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)

operator++() [1/2]

```
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

operator++() [2/2]

```
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	*dummy*	Parameter required to define operator

Returns

Original copy

operator+=()

```
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)

operator-() [1/2]

```
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

operator-() [2/2]

```
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)

operator--() [1/2]

```
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

operator--() [2/2]

```
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

operator-=()

```
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)

operator<()

```
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

operator<=()

```
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

operator=()

```
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

operator==()

```
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

operator>()

```
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

operator>=()

```
template<class dataType>
bool os::vector2d< dataType >::operator>= (
              const vector2d< dataType > & vec ) const  [inline]
```

rotate()

```
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

rotateSelf()

```
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

scale()

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

scaleSelf()

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

subtract()

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

subtractSelf()

```
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

12.41.4 Member Data Documentation

x

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

X axis vector component.

y

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

Y axis vector component.

12.42 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.
- **operator size_t** () const

  Used for hash tables.
- 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.

12.42.1 Detailed Description

template<class dataType>
class os::vector3d< dataType >

3-dimensional vector

This template class contains the functions and operators needed to preform arithmetic on a 3 dimensional vector

12.42.2 Constructor & Destructor Documentation

vector3d() [1/4]

```
template<class dataType>
os::vector3d< dataType >::vector3d ( )  [inline]
```

Default constructor.

Constructs a 3 dimensional vector with x, y and z as 0.

vector3d() [2/4]

```
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

vector3d() [3/4]

```
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

vector3d() [4/4]

```
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

~vector3d()

```
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.

12.42.3 Member Function Documentation

add()

```
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

addSelf()

`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

compare()

`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 that 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

crossProduct()

`template<class dataType>`
vector3d`<dataType> `**os::vector3d**`< dataType >::crossProduct (`
`const `**vector3d**`< dataType > & `*vec*` ) const [inline]`

Cross-product.

Preform 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

crossSelf()

```
template<class dataType>
vector3d<dataType>& os::vector3d< dataType >::crossSelf (
            const vector3d< dataType > & vec )  [inline]
```

Cross-product to self.

Preform 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

dotProduct()

```
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

length()

```
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

operator size_t()

```
template<class dataType>
os::vector3d< dataType >::operator size_t ( ) const  [inline]
```

Used for hash tables.

Returns

Vector converted to size_t

operator"!=()

```
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

operator()()

```
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
in	*Y*	Value of y dimension
in	*Z*	Value of z dimension

Returns

Reference to this vector

operator∗()

```
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)

operator*=()

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)

operator+()

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)

operator++() [1/2]

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

operator++() [2/2]

```
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	*dummy*	Parameter required to define operator

Returns

Original copy

operator+=()

```
template<class dataType>
vector3d<dataType>& os::vector3d< dataType >::operator+= (
            const vector3d< dataType > & vec ) [inline]
```

Add vector to self.

Parameters

in	*vec*	Reference to vector to be added

Returns

vector3d<dataType>::addSelf(vec)

operator-() [1/2]

```
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

operator-() [2/2]

```
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)

operator--() [1/2]

```
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

operator--() [2/2]

```
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	*dummy*	Parameter required to define operator

Returns

Original copy

operator-=()

```
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)

operator<()

```
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

operator<=()

```
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

operator=()

```
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

operator==()

```
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

operator>()

```
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

operator>=()

```
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

scale()

```
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	*target*	Vector length to be scaled to

Returns

The scaled vector

scaleSelf()

```
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	*target*	Vector length to be scaled to

Returns

Reference to this

subtract()

```
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	*vec*	Reference to vector to be subtracted

Returns

Result of the vector subtraction

subtractSelf()

```
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	*vec*	Reference to vector to be subtracted

Returns

Reference to self

12.42.4 Member Data Documentation

x

```
template<class dataType>
dataType os::vector3d< dataType >::x
```

X axis vector component.

y

template<class dataType>
dataType **os::vector3d**< dataType >::y

Y axis vector component.

z

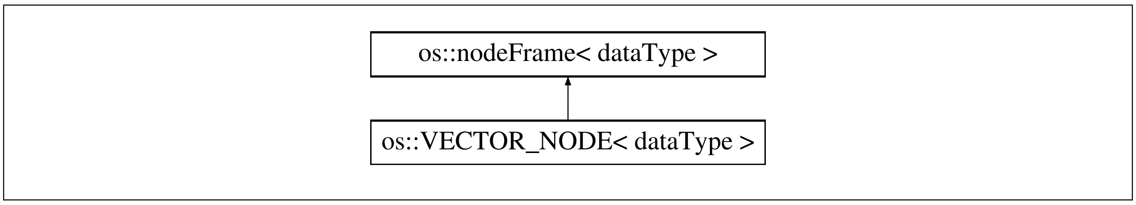
template<class dataType>
dataType **os::vector3d**< dataType >::z

Z axis vector component.

12.43 os::VECTOR_NODE< dataType > Class Template Reference

Vector node.

Inheritance diagram for os::VECTOR_NODE< dataType >:



Public Member Functions

- ~**VECTOR_NODE** () final

  Destructor Class is not designed to be inherited from.
- **smart_ptr**< dataType > **get** () final throw ()

  Returns a pointer.
- const **smart_ptr**< dataType > **constGet** () const final throw ()

  Returns a const pointer.
- dataType & **operator*** () final throw (descriptiveException)

  De-reference.
- const dataType & **operator*** () const final throw (descriptiveException)

  De-reference.
- bool **valid** () const final

  Valid data query Checks if the provided data is valid.
- bool **iterable** () const final

  Returns if the node is iterable.
- bool **randomAccess** () const final

  Returns if the node can be accessed randomly.
- **smart_ptr**< **nodeFrame**< dataType > > **getNext** () final throw (descriptiveException)

  Returns the next vector frame.

- const **smart_ptr**< **nodeFrame**< dataType > > **getNextConst** () const final throw (descriptiveException)

  Returns the next vector frame.
- **smart_ptr**< **nodeFrame**< dataType > > **getPrev** () final throw (descriptiveException)

  Returns the previous vector frame.
- const **smart_ptr**< **nodeFrame**< dataType > > **getPrevConst** () const final throw (descriptiveException)

  Returns the previous vector frame.
- **smart_ptr**< **nodeFrame**< dataType > > **access** (long offset) final throw (descriptiveException)

  Access node by index.
- const **smart_ptr**< **nodeFrame**< dataType > > **constAccess** (long offset) const final throw (descriptiveException)

  Access node by index.
- void **remove** () final throw (descriptiveException)

  Remove this node from the vector.

Static Public Attributes

- static const bool **ITERABLE** = true

  Vector frames are iterable.
- static const bool **RANDOM_ACCESS** = true

  Vector frames allow random-access.

Private Member Functions

- **VECTOR_NODE** (DRIVER_CLASS ∗src, size_t pos)

  *Private constructor This constructor is not designed to be accessed by anything other than the **VECTOR** (p. 336) class derivatives.*

Private Attributes

- size_t **position**

  Current position of the vector iterator.

12.43.1 Detailed Description

template<class dataType>
class os::VECTOR_NODE< dataType >

Vector node.

Used by the iterator to iterate through a vector.

12.43.2 Constructor & Destructor Documentation

VECTOR_NODE()

template<class dataType >
os::VECTOR_NODE< dataType >::**VECTOR_NODE** (
DRIVER_CLASS * *src*,
size_t *pos*) [inline], [private]

Private constructor This constructor is not designed to be accessed by anything other than the **VECTOR** (p. 336) class derivatives.

~VECTOR_NODE()

template<class dataType >
os::VECTOR_NODE< dataType >::~**VECTOR_NODE** ( ) [inline], [final]

Destructor Class is not designed to be inherited from.

12.43.3 Member Function Documentation

access()

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

Access node by index.

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

Returns

Offset node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 276).

constAccess()

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

Access node by index.

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

Returns

Offset node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 276).

constGet()

```
template<class dataType >
const smart_ptr<dataType> os::VECTOR_NODE< dataType >::constGet ( ) const throw )   [inline], [final],
[virtual]
```

Returns a const pointer.

Returns a const pointer to the contained object, this pointer cannot be modified.

Returns

Const pointer to contained object

Implements **os::nodeFrame**< **dataType** > (p. 277).

get()

```
template<class dataType >
smart_ptr<dataType> os::VECTOR_NODE< dataType >::get ( ) throw )   [inline], [final], [virtual]
```

Returns a pointer.

Returns a pointer to the contained object, this pointer can be modified.

Returns

Pointer to contained object

Implements **os::nodeFrame**< **dataType** > (p. 277).

getNext()

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

Returns the next vector frame.

Returns

Next node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 277).

getNextConst()

```
template<class dataType >
const smart_ptr<nodeFrame<dataType> > os::VECTOR_NODE< dataType >::getNextConst ( ) const throw descriptive←
Exception)   [inline], [final], [virtual]
```

Returns the next vector frame.

Returns

Next node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 278).

getPrev()

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

Returns the previous vector frame.

Returns

Previous node, mutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 278).

getPrevConst()

`template<class dataType >`
const **smart_ptr**<**nodeFrame**<dataType> > **os::VECTOR_NODE**< dataType >::getPrevConst ( ) const throw **descriptive↩ Exception**) [inline], [final], [virtual]

Returns the previous vector frame.

Returns

Previous node, immutable

Reimplemented from **os::nodeFrame**< **dataType** > (p. 279).

iterable()

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

Returns if the node is iterable.

Returns

object::ITERABLE

Reimplemented from **os::nodeFrame**< **dataType** > (p. 279).

operator*() [1/2]

`template<class dataType >`
dataType& **os::VECTOR_NODE**< dataType >::operator* ( ) throw **descriptiveException**) [inline], [final], [virtual]

De-reference.
Returns a reference to the contained object, the reference can be modified.

Returns

Contained object

Implements **os::nodeFrame**< **dataType** > (p. 280).

operator*() [2/2]

```
template<class dataType >
const dataType& os::VECTOR_NODE< dataType >::operator* ( ) const throw descriptiveException)   [inline],
[final], [virtual]
```

De-reference.

Returns a const reference to the contained object, the reference cannot be modified.

Returns

Contained object

Implements **os::nodeFrame< dataType >** (p. 280).

randomAccess()

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

Returns if the node can be accessed randomly.

Returns

object::RANDOM_ACCESS

Reimplemented from **os::nodeFrame< dataType >** (p. 281).

remove()

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

Remove this node from the vector.

Returns

void

Reimplemented from **os::nodeFrame< dataType >** (p. 281).

valid()

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

Valid data query Checks if the provided data is valid.

Returns

true if valid, else, false

Reimplemented from **os::nodeFrame< dataType >** (p. 282).

12.43.4 Member Data Documentation

ITERABLE

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

Vector frames are iterable.

position

template<class dataType >
size_t **os::VECTOR_NODE**< dataType >::position [private]

Current position of the vector iterator.

RANDOM_ACCESS

template<class dataType >
const bool **os::VECTOR_NODE**< dataType >::RANDOM_ACCESS = true [static]

Vector frames allow random-access.