Chapter 1

UnitTest Documentation

This document contains documentation for both the UnitTest library and its dependencies. The UnitTest library is dependent on the Datastructures library, and the documentation for this library is included as well.

1.1 Datastructures

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.

1.1.1 Smart Pointer

The smart_ptr class wraps the standard C++ pointer providing an optional reference count for memory management. Additionally, the smart_ptr class forces explicit casts to parent classes, forcing clarification of complex inheritance structures. Please consult figure 1.1 for the smart_ptr UML diagram.

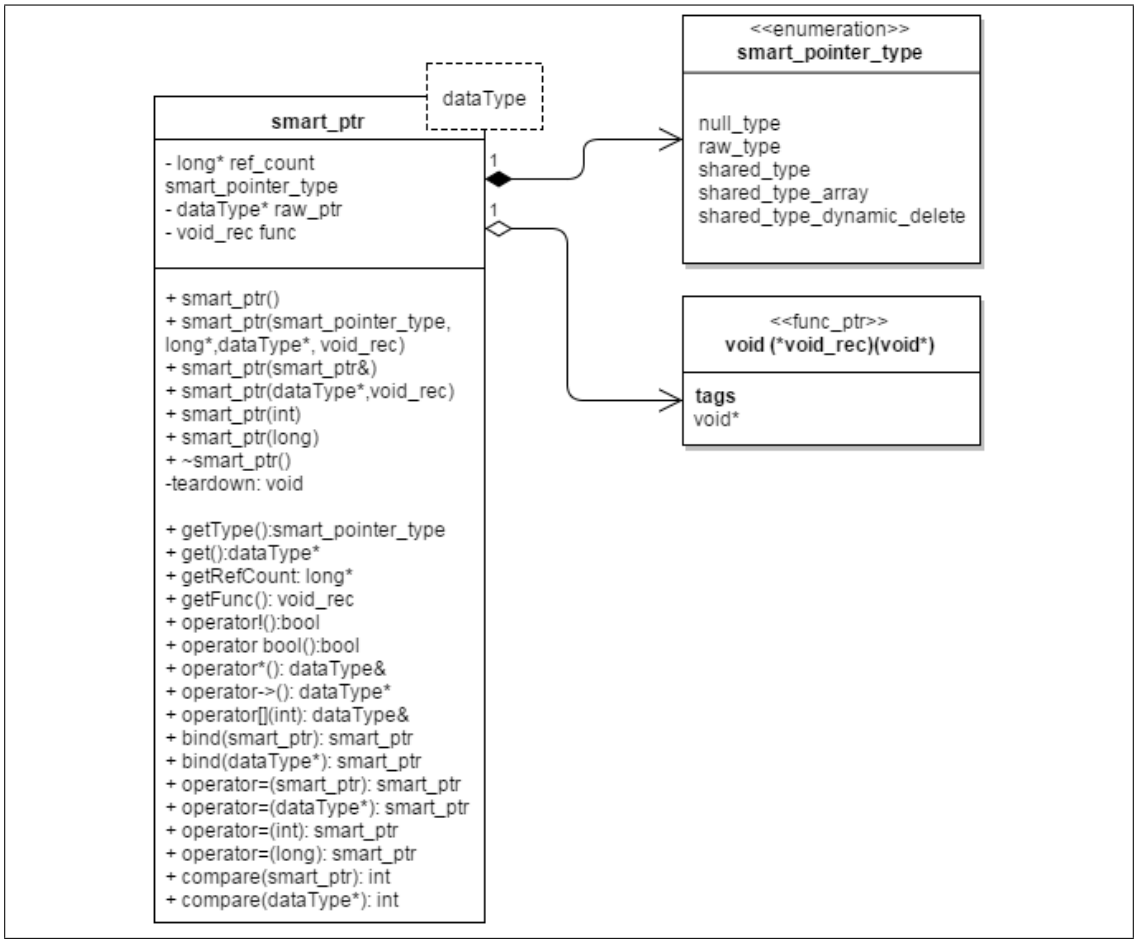

Figure 1.1: UML Diagram for smart_ptr

1.1.2 Unit Testing

The testing of the Datastructures library is preformed 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

1.2 Unit Test

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.

1.2.1 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.