C++ LRU Cache Template

1.3

Generated by Doxygen 1.7.1

Sun May 15 2011 20:17:27

Contents

Chapter 1

LRU Cache

1.1 Introduction

Fast, thread safe C++ template with Least Recently Used (LRU) removal semantics. Complete with a comprehensive unit test suite. Threading features require the BOOST scientific library to be installed.

1.2 Usage

An LRU cache is a fixed size cache that discards the oldest (least recently accessed) elements after it fills up. It's ideally suited to be used in situations where you need to speed up access to slower data sources (databases, synthetic structures, etc.). Below is a simple example of using it to cache strings using integer keys.

1.3 See Also

See: `LRU Cache`

Chapter 2

Unit Testing Framework

See implementation in **unit_test.h** (p. 37)

2.1 Writing Unit Tests

Ideally, unit tests are written with the code they test. The easiest way to do this is to include all the unit tests at the bottom of each source that they relate too. It's also possible to create source files of nothing but tests to expand coverage across multiple translation modules.

Let's take a simple example: we have a new function that adds two numbers.

```
int addTwoNumbers( int a, int b ) {
  return a + b;
}
```

To write a unit test that checks that $addTwoNumbers(x_1, x_2) = x_1 + x_2$ we would write the unit test like so:

```
#ifdef UNITTEST
#include "unit_test.h"

UNIT_TEST_DEFINES

DEFINE_TEST( check_two_plus_two ) {
  unit_assert( "2+2=4", addTwoNumbers(2,2)==4 );
}
```

```
UNIT_TEST_RUN( "addTwoNumbers Tests" )
  ADD_TEST( check_two_plus_two )
UNIT_TEST_END

#endif // UNITTEST
```

Now we have a test suite defined that will only be compiled when we define UNITTEST. UNIT_TEST_RUN actually creates a main() function that runs the tests so if we put this code into a file (say add.cpp) then we can compile and run it like so:

```
# gcc -DUNITTEST -o unit_test_add add.cpp
# ./unit_test_add
---[ addTwoNumbers Tests ]---
  2+2=4: PASSED
```

So far so good, let's add a new test that we think will fail.

```
#ifdef UNITTEST
#include "unit_test.h"

UNIT_TEST_DEFINES

DEFINE_TEST( check_two_plus_two ) {
  unit_assert( "2+2=4", addTwoNumbers(2,2)==4 );
  unit_pass();
}

DEFINE_TEST( check_bogus ) {
  unit_assert( "1+5=9", addTwoNumbers(1,5)==9 );
  unit_pass();
}

UNIT_TEST_RUN( "addTwoNumbers Tests" )
  ADD_TEST( check_negatives )
UNIT_TEST_END

#endif // UNITTEST
```

Running the unit_test now we get:

```
# gcc -DUNITTEST -o unit_test_add add.cpp
# ./unit_test_add
---[ addTwoNumbers Tests ]---
  2+2=4: PASSED
  1+5=9: FAILED
```

2.2 Integrating with Automake

Automake has the ability to define testing targets that get run when issue "make check" command. Adding these tests are pretty straight forward. For the above we would add this to our Makefile.am:

```
TESTS = unit_test_add
noinst_PROGRAMS = unit_test_add
CLEANFILES = add_unit.cpp
unit_test_add_SOURCES = add_unit.cpp

%_unit.cpp: %.cpp
$(CXX) -E -o $*_unit.cpp $*.C @CFLAGS@ -DUNITTEST=1
```

To add addtional unit tests you just modify the first four lines. For example: to add a new unit test suite in the file sub.C we might do this.

```
TESTS = unit_test_add unit_test_sub
noinst_PROGRAMS = unit_test_add unit_test_sub
CLEANFILES = add_unit.cpp sub_unit.cpp
unit_test_add_SOURCES = add_unit.cpp
unit_test_sub_SOURCES = sub_unit.cpp
```

2.3 Implementation Notes

Chapter 3

Test List

Member DEFINE_TEST (p. **24**)**(lru_cache_1cycle)** Basic creation and desctruction test

Member DEFINE_TEST (p. **28**)**(lru_cache_threads)** Check for badness with multithreaded access, this is more of a stress test than an empirical test.

Member DEFINE_TEST (p. **27**)**(lru_cache_scope_check)** Check that objects inserted in a different scope are still there.

Member DEFINE_TEST (p. **26**)**(lru_cache_stress)** Insert lots of objects and benchmark the rate.

Chapter 4

Class Index

4.1 Class List

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

Chapter 5

File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 6

Class Documentation

6.1 LRUCache< Key, Data, Sizefn > Class Template Reference

Template cache with an LRU removal policy.

`#include <lru_cache.h>`

Public Types

- typedef std::list< std::pair< Key, Data > > **List**
  Main cache storage typedef.
- typedef List::iterator **List_Iter**
  Main cache iterator.
- typedef List::const_iterator **List_cIter**
  Main cache iterator (const).
- typedef std::vector< Key > **Key_List**
  List of keys.
- typedef Key_List::iterator **Key_List_Iter**
  Main cache iterator.
- typedef Key_List::const_iterator **Key_List_cIter**
  Main cache iterator (const).

- typedef std::map< Key, **List_Iter** > **Map**

  Index typedef.

- typedef std::pair< Key, **List_Iter** > **Pair**

  Pair of Map elements.

- typedef Map::iterator **Map_Iter**

  Index iterator.

- typedef Map::const_iterator **Map_cIter**

  Index iterator (const).

Public Member Functions

- **LRUCache** (const unsigned long Size)

  Creates a cache that holds at most Size worth of elements.

- ~**LRUCache** ()

  Destructor - cleans up both index and storage.

- const unsigned long **size** (void) const

  Gets the current abstract size of the cache.

- const unsigned long **max_size** (void) const

  Gets the maximum sbstract size of the cache.

- void **clear** (void)

  Clears all storage and indices.

- bool **exists** (const Key &key)

  Checks for the existance of a key in the cache.

- void **remove** (const Key &key)

  Removes a key-data pair from the cache.

- void **touch** (const Key &key)

  Touches a key in the Cache and makes it the most recently used.

- Data * **fetch_ptr** (const Key &key, bool touch=true)

  Fetches a pointer to cache data.

- Data **fetch** (const Key &key, bool touch_data=true)

  Fetches a copy of cached data.

- bool **fetch** (const Key &key, Data &data, bool touch_data=true)

  Fetches a pointer to cache data.

- void **insert** (const Key &key, const Data &data)

  Inserts a key-data pair into the cache and removes entries if neccessary.

- const **Key_List get_all_keys** (void)

  Get a list of keys.

6.1.1 Detailed Description

template<class Key, class Data, class Sizefn = Countfn< Data >> class LRUCache< Key, Data, Sizefn >

Template cache with an LRU removal policy.

This template creats a simple collection of key-value pairs that grows until the size specified at construction is reached and then begins discard the Least Recently Used element on each insertion.

Examples:

lru_example.cpp.

Definition at line **73** of file **lru_cache.h**.

6.1.2 Constructor & Destructor Documentation

6.1.2.1 template<class Key , class Data , class Sizefn = Countfn< Data >> LRUCache< Key, Data, Sizefn >::LRUCache (const unsigned long *Size*) [inline]

Creates a cache that holds at most Size worth of elements.

Parameters

Size maximum size of cache

Definition at line **101** of file **lru_cache.h**.

```
                                        :
_max_size( Size ),
_curr_size( 0 )
{}
```

6.1.3 Member Function Documentation

6.1.3.1 template<class Key , class Data , class Sizefn = Countfn< Data >> const unsigned long LRUCache< Key, Data, Sizefn >::size (void) const [inline]

Gets the current abstract size of the cache.

Returns

current size

Definition at line **112** of file **lru_cache.h**.

Referenced by **DEFINE_TEST**().

```
{ return _curr_size; }
```

6.1.3.2 template<class Key , class Data , class Sizefn = Countfn< Data >> const unsigned long LRUCache< Key, Data, Sizefn >::max_size (void) const [inline]

Gets the maximum sbstract size of the cache.

Returns

maximum size

Definition at line **117** of file **lru_cache.h**.

Referenced by **DEFINE_TEST**().

```
{ return _max_size; }
```

6.1.3.3 template<class Key , class Data , class Sizefn = Countfn< Data >> bool LRUCache< Key, Data, Sizefn >::exists (const Key & *key*) [inline]

Checks for the existance of a key in the cache.

Parameters

key to check for

Returns

bool indicating whether or not the key was found.

Definition at line **131** of file **lru_cache.h**.

Referenced by **DEFINE_TEST**().

```
                                                    {
                SCOPED_MUTEX;
#else
        inline bool exists( const Key &key ) const {
#endif
                return _index.find( key ) != _index.end();
        }
```

6.1.3.4 template<class Key , class Data , class Sizefn = Countfn< Data >> void LRUCache< Key, Data, Sizefn >::remove (const Key & *key*) [inline]

Removes a key-data pair from the cache.

Parameters

key to be removed

Definition at line **142** of file **lru_cache.h**.

Referenced by **DEFINE_TEST**().

```
                                                    {
#ifdef _REENTRANT
                SCOPED_MUTEX;
#endif
                Map_Iter miter = _index.find( key );
                if( miter == _index.end() ) return;
                _remove( miter );
        }
```

6.1.3.5 template<class Key , class Data , class Sizefn = Countfn< Data >> void LRUCache< Key, Data, Sizefn >::touch (const Key & *key*) [inline]

Touches a key in the Cache and makes it the most recently used.

Parameters

key to be touched

Definition at line **154** of file **lru_cache.h**.

Referenced by **DEFINE_TEST()**.

```
                                                {
        SCOPED_MUTEX;
        _touch( key );
}
```

6.1.3.6 template<class Key , class Data , class Sizefn = Countfn< Data >> Data* LRUCache< Key, Data, Sizefn >::fetch_ptr (const Key & *key*, bool *touch* = `true`) [inline]

Fetches a pointer to cache data.

Parameters

key to fetch data for

touch whether or not to touch the data

Returns

pointer to data or NULL on error

Definition at line **164** of file **lru_cache.h**.

Referenced by **DEFINE_TEST()**.

```
                                                                    {
        SCOPED_MUTEX;
        Map_Iter miter = _index.find( key );
        if( miter == _index.end() ) return NULL;
        _touch( key );
        return &(miter->second->second);
}
```

6.1.3.7 template<class Key , class Data , class Sizefn = Countfn< Data >> Data LRUCache< Key, Data, Sizefn >::fetch (const Key & *key*, bool *touch_data* = `true`) [inline]

Fetches a copy of cached data.

Parameters

key to fetch data for

touch_data whether or not to touch the data

Returns

copy of the data or an empty Data object if not found

Definition at line **177** of file **lru_cache.h**.

Referenced by **DEFINE_TEST()**, and **dump()**.

```
                                                                    {
        SCOPED_MUTEX;
        Map_Iter miter = _index.find( key );
        if( miter == _index.end() )
                return Data();
        Data tmp = miter->second->second;
        if( touch_data )
                _touch( key );
        return tmp;
}
```

6.1.3.8 template<class Key , class Data , class Sizefn = Countfn< Data >> bool LRUCache< Key, Data, Sizefn >::fetch (const Key & *key,* Data & *data,* bool *touch_data* = `true`) [inline]

Fetches a pointer to cache data.

Parameters

key to fetch data for

data to fetch data into

touch_data whether or not to touch the data

Returns

whether or not data was filled in

Definition at line **194** of file **lru_cache.h**.

```
{
        SCOPED_MUTEX;
        Map_Iter miter = _index.find( key );
        if( miter == _index.end() ) return false;
        if( touch_data )
          _touch( key );
        data = miter->second->second;
        return true;
}
```

6.1.3.9 template<class Key , class Data , class Sizefn = Countfn< Data >> void LRUCache< Key, Data, Sizefn >::insert (const Key & *key*, const Data & *data*) [inline]

Inserts a key-data pair into the cache and removes entries if neccessary.

Parameters

key object key for insertion

data object data for insertion

Note

This function checks key existance and touches the key if it already exists.

Definition at line **209** of file **lru_cache.h**.

Referenced by **DEFINE_TEST()**.

```
                                                                {
                        SCOPED_MUTEX;
                        // Touch the key, if it exists, then replace the content.

                        Map_Iter miter = _touch( key );
                        if( miter != _index.end() )
                                _remove( miter );

                        // Ok, do the actual insert at the head of the list
                        _list.push_front( std::make_pair( key, data ) );
                        List_Iter liter = _list.begin();

                        // Store the index
                        _index.insert( std::make_pair( key, liter ) );
                        _curr_size += Sizefn()( data );

                        // Check to see if we need to remove an element due to ex
    ceeding max_size
                        while( _curr_size > _max_size ) {
                                // Remove the last element.
                                liter = _list.end();
                                --liter;
                                _remove( liter->first );
                        }
                }
```

6.1.3.10 template<class Key , class Data , class Sizefn = Countfn< Data >> const Key_List LRUCache< Key, Data, Sizefn >::get_all_keys (void) [inline]

Get a list of keys.

Returns

list of the current keys.

Definition at line **236** of file **lru_cache.h**.

Referenced by **dump**().

```
                                                    {
                    SCOPED_MUTEX;
                    Key_List ret;
                    for( List_cIter liter = _list.begin(); liter != _list.end
    (); liter++ )
                            ret.push_back( liter->first );
                    return ret;
            }
```

The documentation for this class was generated from the following files:

- **lru_cache.h**
- lru_cache_unit.cpp

Chapter 7

File Documentation

7.1 lru_cache.cpp File Reference

Typedefs

- typedef **LRUCache**< std::string, std::string > **unit_lru_type**
  LRUCache *(*p. *13) type for use in the unit tests.*
- typedef **LRUCache**< int, int > **unit_lru_type2**
  LRUCache *(*p. *13) POD type for use in the unit tests.*
- typedef **LRUCache**< int, test_big_data > **unit_lru_type3**
  LRUCache *(*p. *13) with large data for use in the unit tests.*

Functions

- std::string **dump** (**unit_lru_type** *L)
  Dumps the cache for debugging.
- UNIT_TEST_DEFINES **DEFINE_TEST** (lru_cache_1cycle)
- **DEFINE_TEST** (lru_cache_stress)
- **DEFINE_TEST** (lru_cache_scope_check)
- **DEFINE_TEST** (lru_cache_threads)

Variables

- **unit_lru_type3** * **L3**

Scoping test object.

7.1.1 Detailed Description

Template cache with an LRU removal policy (unit tests)

Author

Patrick Audley

Definition in file **lru_cache.cpp**.

7.1.2 Function Documentation

7.1.2.1 UNIT_TEST_DEFINES DEFINE_TEST (lru_cache_1cycle)

Test

Basic creation and desctruction test

Definition at line **50** of file **lru_cache.cpp**.

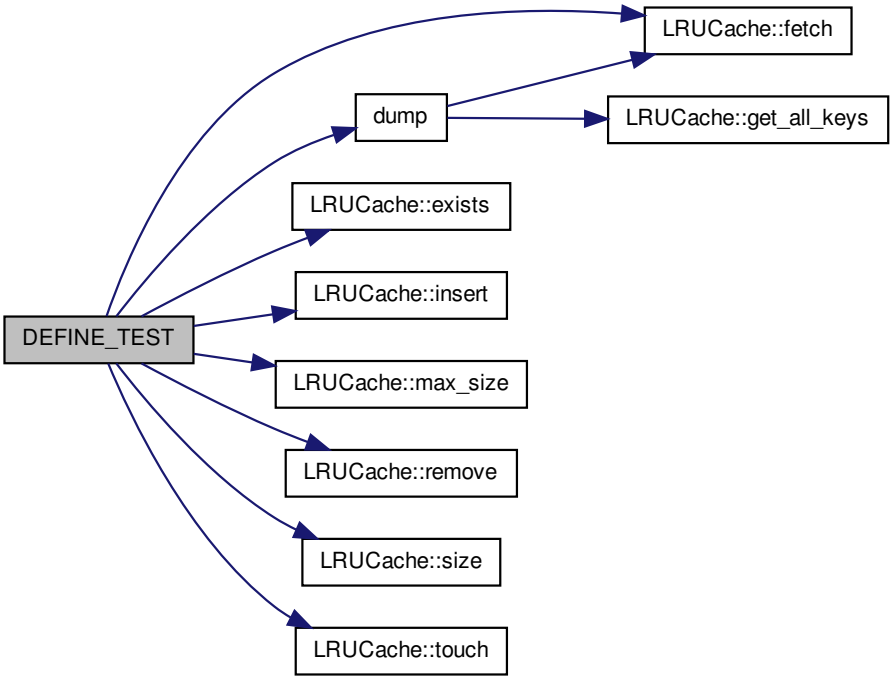
References **dump**(), **LRUCache< Key, Data, Sizefn >::exists**(), **LRUCache< Key, Data, Sizefn >::fetch**(), **LRUCache< Key, Data, Sizefn >::insert**(), **LRUCache< Key, Data, Sizefn >::max_size**(), **LRUCache< Key, Data, Sizefn >::remove**(), **LRUCache< Key, Data, Sizefn >::size**(), **LRUCache< Key, Data, Sizefn >::touch**(), **unit_assert**, and **unit_pass**.

```
                          {
  const std::string unit_data_1cycle_a("foo:4\n");
  const std::string unit_data_1cycle_b("bar:flower\nfoo:4\n");
  const std::string unit_data_1cycle_c("foo:4\nbar:flower\n");
  const std::string unit_data_1cycle_d("foo:moose\nbaz:Stalin\nbar:flower\n
");
  const std::string unit_data_1cycle_e("foo:moose\nbar:flower\n");
  const std::string unit_data_1cycle_f("quz:xyzzy\nbaz:monkey\nfoo:moose\n"
);
  const std::string unit_data_1cycle_g("coat:mouse\npants:cat\nsocks:bear\n
");

  unit_lru_type *L = new unit_lru_type(3);
  unit_assert( "size==0", (L->size() == 0) );
  unit_assert( "maxsize==3", (L->max_size() == 3) );

  // Checking a bogus key shouldn't alter the cache.
  L->exists( "foo" );
  unit_assert( "exists() doesn't increase size", (L->size() == 0) );
```

```
  // Check insert() and exists()
  L->insert( "foo", "4" );
  unit_assert( "size==1 after insert(foo,4)", (L->size() == 1) );
  unit_assert( "check exists(foo)", L->exists( "foo" ) );
  unit_assert( "contents check a)", unit_data_1cycle_a.compare( dump( L ) )
 == 0 );

  // Check second insert and ordering
  L->insert( "bar", "flower" );
  unit_assert( "size==2 after insert(bar,flower)", (L->size() == 2) );
  unit_assert( "contents check b)", unit_data_1cycle_b.compare( dump( L ) )
 == 0 );

  // Check touching
  L->touch( "foo" );
  unit_assert( "contents check c)", unit_data_1cycle_c.compare( dump( L ) )
 == 0 );

  // Insert of an existing element should result in only a touch
  L->insert( "bar", "flower" );
  unit_assert( "verify insert touches", unit_data_1cycle_b.compare( dump( L
 ) ) == 0 );

  // Verify that fetch works
  unit_assert( "verify fetch(bar)", ( std::string("flower").compare( L->
fetch("bar") ) == 0 ) );

  // Insert of an existing element with new data should replace and touch
  L->insert( "baz", "Stalin" );
  L->insert( "foo", "moose" );
  unit_assert( "verify insert replaces", unit_data_1cycle_d.compare( dump(
L ) ) == 0 );

  // Test removal of an existing member.
  L->remove( "baz" );
  unit_assert( "verify remove works", unit_data_1cycle_e.compare( dump( L )
 ) == 0 );

  // Test LRU removal as we add more members than max_size()
  L->insert( "baz", "monkey" );
  L->insert( "quz", "xyzzy" );
  unit_assert( "verify LRU semantics", unit_data_1cycle_f.compare( dump( L
) ) == 0 );

  // Stress test the implementation a little..
  const char *names[10] = { "moose", "dog", "bear", "cat", "mouse", "hat",
"mittens", "socks", "pants", "coat" };
  for( int i = 0; i < 50; i++ ) {
          L->insert( names[ i % 10 ], names[ i % 9 ] );
  }
  unit_assert( "stress test a little", unit_data_1cycle_g.compare( dump( L
) ) == 0 );

  // Setup a little for the third test which verifies that scoped reference
s inserted into the cache don't disappear.
  L3 = new unit_lru_type3(2);
  for( int i = 0; i < 10; i++ ) {
```

```
                test_big_data B;
                snprintf( B.buffer, 1000, "%d\n", i );
                L3->insert( i, B );
        }

        unit_pass();
}
```

Here is the call graph for this function:



7.1.2.2 DEFINE_TEST (lru_cache_stress)

Test

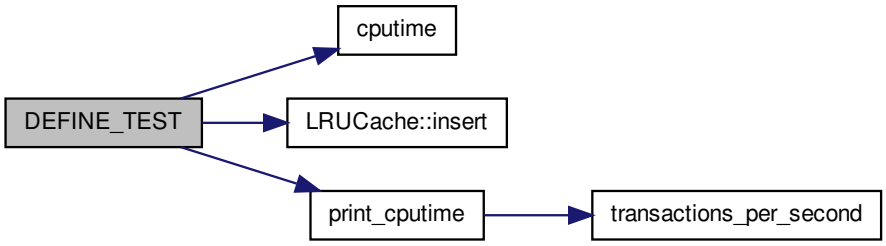
Insert lots of objects and benchmark the rate.

Definition at line **123** of file **lru_cache.cpp**.

References **cputime()**, **LRUCache< Key, Data, Sizefn >::insert()**, **print_cputime()**, and **unit_pass**.

```
                                {
        // Stress test the implementation a little more using no objects
        unit_lru_type2 *L2 = new unit_lru_type2(5);
        double t0 = cputime();
        for( int i = 0; i < TRANSACTIONS; i++ ) {
                L2->insert( i, i-1 );
        }
        double t1 = cputime();
        delete L2;
        print_cputime( "(int,int) inserts", t1-t0, TRANSACTIONS );
        unit_pass();
}
```

Here is the call graph for this function:



7.1.2.3 DEFINE_TEST (lru_cache_scope_check)

Test

Check that objects inserted in a different scope are still there.

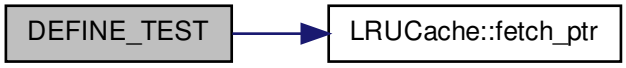
Definition at line **137** of file **lru_cache.cpp**.

References **LRUCache< Key, Data, Sizefn >::fetch_ptr()**, **unit_assert**, and **unit_pass**.

```
                                {
        test_big_data* B = L3->fetch_ptr( 9 );
        unit_assert( "scope check element L3[1]", ( strncmp( B->buffer, "9\n", 10
```

```
      00 ) == 0 ) );
        B = L3->fetch_ptr( 8 );
        unit_assert( "scope check element L3[2]", ( strncmp( B->buffer, "8\n", 10
      00 ) == 0 ) );
        delete L3;
        unit_pass();
}
```

Here is the call graph for this function:



7.1.2.4 DEFINE_TEST (lru_cache_threads)

Test

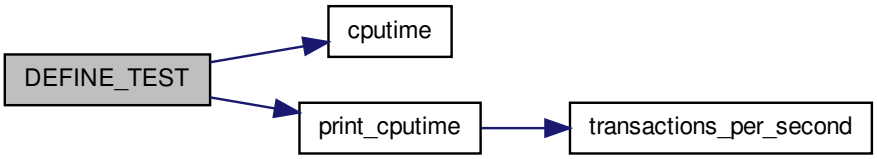
Check for badness with multithreaded access, this is more of a stress test than an empirical test.

Definition at line **164** of file **lru_cache.cpp**.

References **cputime()**, **print_cputime()**, and **unit_pass**.

```
                                {
        L4 = new unit_lru_type2( 20 );
        boost::thread_group thrds;
        double t0 = cputime();
        for (int i=0; i < THREAD_COUNT; ++i)
                thrds.create_thread(&insert_junk);
        thrds.join_all();
        double t1 = cputime();
        print_cputime( "(int,int) multithreaded inserts", t1-t0, THREAD_TRANS*THR
      EAD_COUNT*4 );
        delete L4;
        unit_pass();
}
```

Here is the call graph for this function:



7.2 lru_cache.cpp

```
00001 /*****************************************************************************
00002  *   Copyright (C) 2004-2011 by Patrick Audley                               *
00003  *   paudley@blackcat.ca                                                     *
00004  *   http://patrickaudley.com                                                *
00005  *                                                                           *
00006  *****************************************************************************/
00011 #include "lru_cache.h"
00012
00013 #ifdef UNITTEST
00014 #include "unit_test.h"
00015 #include <string>
00016 #include <stdlib.h>
00017
00019 typedef LRUCache<std::string,std::string> unit_lru_type;
00021 typedef LRUCache<int,int> unit_lru_type2;
00023 class test_big_data {
00024         public:
00025                 char buffer[1000];
00026 };
00028 typedef LRUCache<int,test_big_data> unit_lru_type3;
00029
00031 std::string dump( unit_lru_type *L ) {
00032         unit_lru_type::Key_List _list( L->get_all_keys() );
00033         std::string ret("");
00034         for( unit_lru_type::Key_List_Iter liter = _list.begin(); liter != _list.e
      nd(); liter++ ) {
00035                 ret.append( *liter );
00036                 ret.append( ":" );
00037                 ret.append( L->fetch( *liter, false ) );
00038                 ret.append( "\n" );
00039         }
00040         //std::cout << "Dump--" << std::endl << ret << "----" << std::endl;
00041         return ret;
00042 }
```

```
00043
00045 unit_lru_type3* L3;
00046
00047 UNIT_TEST_DEFINES
00048
00050 DEFINE_TEST( lru_cache_1cycle ) {
00051         const std::string unit_data_1cycle_a("foo:4\n");
00052         const std::string unit_data_1cycle_b("bar:flower\nfoo:4\n");
00053         const std::string unit_data_1cycle_c("foo:4\nbar:flower\n");
00054         const std::string unit_data_1cycle_d("foo:moose\nbaz:Stalin\nbar:flower\n
      ");
00055         const std::string unit_data_1cycle_e("foo:moose\nbar:flower\n");
00056         const std::string unit_data_1cycle_f("quz:xyzzy\nbaz:monkey\nfoo:moose\n"
      );
00057         const std::string unit_data_1cycle_g("coat:mouse\npants:cat\nsocks:bear\n
      ");
00058
00059         unit_lru_type *L = new unit_lru_type(3);
00060         unit_assert( "size==0", (L->size() == 0) );
00061         unit_assert( "maxsize==3", (L->max_size() == 3) );
00062
00063         // Checking a bogus key shouldn't alter the cache.
00064         L->exists( "foo" );
00065         unit_assert( "exists() doesn't increase size", (L->size() == 0) );
00066
00067         // Check insert() and exists()
00068         L->insert( "foo", "4" );
00069         unit_assert( "size==1 after insert(foo,4)", (L->size() == 1) );
00070         unit_assert( "check exists(foo)", L->exists( "foo" ) );
00071         unit_assert( "contents check a)", unit_data_1cycle_a.compare( dump( L ) )
       == 0 );
00072
00073         // Check second insert and ordering
00074         L->insert( "bar", "flower" );
00075         unit_assert( "size==2 after insert(bar,flower)", (L->size() == 2) );
00076         unit_assert( "contents check b)", unit_data_1cycle_b.compare( dump( L ) )
       == 0 );
00077
00078         // Check touching
00079         L->touch( "foo" );
00080         unit_assert( "contents check c)", unit_data_1cycle_c.compare( dump( L ) )
       == 0 );
00081
00082         // Insert of an existing element should result in only a touch
00083         L->insert( "bar", "flower" );
00084         unit_assert( "verify insert touches", unit_data_1cycle_b.compare( dump( L
       ) ) == 0 );
00085
00086         // Verify that fetch works
00087         unit_assert( "verify fetch(bar)", ( std::string("flower").compare( L->
      fetch("bar") ) == 0 ) );
00088
00089         // Insert of an existing element with new data should replace and touch
00090         L->insert( "baz", "Stalin" );
00091         L->insert( "foo", "moose" );
00092         unit_assert( "verify insert replaces", unit_data_1cycle_d.compare( dump(
      L ) ) == 0 );
```

```
00093
00094         // Test removal of an existing member.
00095         L->remove( "baz" );
00096         unit_assert( "verify remove works", unit_data_1cycle_e.compare( dump( L )
       ) == 0 );
00097
00098         // Test LRU removal as we add more members than max_size()
00099         L->insert( "baz", "monkey" );
00100         L->insert( "quz", "xyzzy" );
00101         unit_assert( "verify LRU semantics", unit_data_1cycle_f.compare( dump( L
      ) ) == 0 );
00102
00103         // Stress test the implementation a little..
00104         const char *names[10] = { "moose", "dog", "bear", "cat", "mouse", "hat",
      "mittens", "socks", "pants", "coat" };
00105         for( int i = 0; i < 50; i++ ) {
00106                 L->insert( names[ i % 10 ], names[ i % 9 ] );
00107         }
00108         unit_assert( "stress test a little", unit_data_1cycle_g.compare( dump( L
      ) ) == 0 );
00109
00110         // Setup a little for the third test which verifies that scoped reference
      s inserted into the cache don't disappear.
00111         L3 = new unit_lru_type3(2);
00112         for( int i = 0; i < 10; i++ ) {
00113                 test_big_data B;
00114                 snprintf( B.buffer, 1000, "%d\n", i );
00115                 L3->insert( i, B );
00116         }
00117
00118         unit_pass();
00119 }
00120
00121 #define TRANSACTIONS 50000
00122
00123 DEFINE_TEST( lru_cache_stress ) {
00124         // Stress test the implementation a little more using no objects
00125         unit_lru_type2 *L2 = new unit_lru_type2(5);
00126         double t0 = cputime();
00127         for( int i = 0; i < TRANSACTIONS; i++ ) {
00128                 L2->insert( i, i-1 );
00129         }
00130         double t1 = cputime();
00131         delete L2;
00132         print_cputime( "(int,int) inserts", t1-t0, TRANSACTIONS );
00133         unit_pass();
00134 }
00135
00137 DEFINE_TEST( lru_cache_scope_check ) {
00138         test_big_data* B = L3->fetch_ptr( 9 );
00139         unit_assert( "scope check element L3[1]", ( strncmp( B->buffer, "9\n", 10
      00 ) == 0 ) );
00140         B = L3->fetch_ptr( 8 );
00141         unit_assert( "scope check element L3[2]", ( strncmp( B->buffer, "8\n", 10
      00 ) == 0 ) );
00142         delete L3;
00143         unit_pass();
```

```
00144 }
00145
00146 #ifdef _REENTRANT
00147 #include <boost/thread/thread.hpp>
00148
00149 #define THREAD_TRANS 20000
00150 #define THREAD_COUNT 10
00151
00152 unit_lru_type2 *L4;
00153
00154 void insert_junk(){
00155         for( int i = 0; i < THREAD_TRANS; i++ ) {
00156                 L4->insert( i, i+1 );
00157                 L4->remove( i-5 );
00158                 L4->fetch( i-3 );
00159                 L4->touch( i-10 );
00160         }
00161 }
00162
00164 DEFINE_TEST( lru_cache_threads ) {
00165         L4 = new unit_lru_type2( 20 );
00166         boost::thread_group thrds;
00167         double t0 = cputime();
00168         for (int i=0; i < THREAD_COUNT; ++i)
00169                 thrds.create_thread(&insert_junk);
00170         thrds.join_all();
00171         double t1 = cputime();
00172         print_cputime( "(int,int) multithreaded inserts", t1-t0, THREAD_TRANS*THR
      EAD_COUNT*4 );
00173         delete L4;
00174         unit_pass();
00175 }
00176
00177 #endif
00178
00179 UNIT_TEST_RUN( "LRU Cache" );
00180         ADD_TEST( lru_cache_1cycle );
00181         ADD_TEST( lru_cache_stress );
00182         ADD_TEST( lru_cache_scope_check );
00183 #ifdef _REENTRANT
00184         ADD_TEST( lru_cache_threads );
00185 #endif
00186 UNIT_TEST_END;
00187
00188 #endif
```

7.3 lru_cache.h File Reference

Classes

- class **LRUCache< Key, Data, Sizefn >**

  Template cache with an LRU removal policy.

Defines

- #define **SCOPED_MUTEX** boost::mutex::scoped_lock lock(this->_mutex);

  If we are reentrant then use a BOOST scoped mutex where neccessary.

7.3.1 Detailed Description

Template cache with an LRU removal policy

Author

Patrick Audley

Version

1.3

Date

May 2011

This cache is thread safe if compiled with _REENTRANT defined. It uses the BOOST scientific computing library to provide the thread safety mutexes.

Thanks to `graydon@pobox.com` for the size counting functor.

Definition in file **lru_cache.h**.

7.4 lru_cache.h

```
00001 /*****************************************************************************
00002  *   Copyright (C) 2004-2011 by Patrick Audley                               *
00003  *   paudley@blackcat.ca                                                     *
00004  *   http://patrickaudley.com                                                *
00005  *                                                                           *
00006  *****************************************************************************/
00045 #include <map>
00046 #include <list>
00047 #include <vector>
00048 #ifdef _REENTRANT
00049 #include <boost/thread/mutex.hpp>
00051 #define SCOPED_MUTEX  boost::mutex::scoped_lock lock(this->_mutex);
00052 #else
00053
```

```
00054 #define SCOPED_MUTEX
00055 #endif
00056
00057 template < class T >
00058 struct Countfn {
00059                 unsigned long operator()( const T &x ) { return 1; }
00060 };
00061
00062
00073 template< class Key, class Data, class Sizefn = Countfn< Data > > class LRUCache
      {
00074         public:
00075                 typedef std::list< std::pair< Key, Data > > List;
00076                 typedef typename List::iterator List_Iter;
00077                 typedef typename List::const_iterator List_cIter;
00078                 typedef std::vector< Key > Key_List;
00079                 typedef typename Key_List::iterator Key_List_Iter;
00080                 typedef typename Key_List::const_iterator Key_List_cIter;
00081                 typedef std::map< Key, List_Iter > Map;
00082                 typedef std::pair< Key, List_Iter > Pair;
00083                 typedef typename Map::iterator Map_Iter;

00084                 typedef typename Map::const_iterator Map_cIter;
00085
00086         private:
00087                 List _list;
00088                 Map _index;
00089                 unsigned long _max_size;
00090                 unsigned long _curr_size;
00091
00092 #ifdef _REENTRANT
00093                 boost::mutex _mutex;
00094 #endif
00095
00096         public:
00097
00101                 LRUCache( const unsigned long Size ) :
00102                                 _max_size( Size ),
00103                                 _curr_size( 0 )
00104                                 {}
00105
00107                 ~LRUCache() { clear(); }
00108
00112                 inline const unsigned long size( void ) const { return _curr_size
      ; }
00113
00117                 inline const unsigned long max_size( void ) const { return _max_s
      ize; }
00118
00120                 void clear( void ) {
00121                         SCOPED_MUTEX;
00122                         _list.clear();
00123                         _index.clear();
00124                 };
00125
00130 #ifdef _REENTRANT
00131                 inline bool exists( const Key &key ) {
```

```
00132                           SCOPED_MUTEX;
00133 #else
00134                   inline bool exists( const Key &key ) const {
00135 #endif
00136                           return _index.find( key ) != _index.end();
00137                   }
00138
00142                   inline void remove( const Key &key ) {
00143 #ifdef _REENTRANT
00144                           SCOPED_MUTEX;
00145 #endif
00146                           Map_Iter miter = _index.find( key );
00147                           if( miter == _index.end() ) return;
00148                           _remove( miter );
00149                   }
00150
00154                   inline void touch( const Key &key ) {
00155                           SCOPED_MUTEX;
00156                           _touch( key );
00157                   }
00158
00164                   inline Data *fetch_ptr( const Key &key, bool touch = true ) {
00165                           SCOPED_MUTEX;
00166                           Map_Iter miter = _index.find( key );
00167                           if( miter == _index.end() ) return NULL;
00168                           _touch( key );
00169                           return &(miter->second->second);
00170                   }
00171
00177                   inline Data fetch( const Key &key, bool touch_data = true ) {
00178                           SCOPED_MUTEX;
00179                           Map_Iter miter = _index.find( key );
00180                           if( miter == _index.end() )
00181                                   return Data();
00182                           Data tmp = miter->second->second;
00183                           if( touch_data )
00184                                   _touch( key );
00185                           return tmp;
00186                   }
00187
00194                   inline bool fetch( const Key &key, Data &data, bool touch_data =
      true ) {
00195                           SCOPED_MUTEX;
00196                           Map_Iter miter = _index.find( key );
00197                           if( miter == _index.end() ) return false;
00198                           if( touch_data )
00199                             _touch( key );
00200                           data = miter->second->second;
00201                           return true;
00202                   }
00203
00209                   inline void insert( const Key &key, const Data &data ) {
00210                           SCOPED_MUTEX;
00211                           // Touch the key, if it exists, then replace the content.

00212                           Map_Iter miter = _touch( key );
00213                           if( miter != _index.end() )
```

```
00214                                   _remove( miter );
00215
00216                           // Ok, do the actual insert at the head of the list
00217                           _list.push_front( std::make_pair( key, data ) );
00218                           List_Iter liter = _list.begin();
00219
00220                           // Store the index
00221                           _index.insert( std::make_pair( key, liter ) );
00222                           _curr_size += Sizefn()( data );
00223
00224                           // Check to see if we need to remove an element due to ex
      ceeding max_size
00225                           while( _curr_size > _max_size ) {
00226                                   // Remove the last element.
00227                                   liter = _list.end();
00228                                   --liter;
00229                                   _remove( liter->first );
00230                           }
00231                   }
00232
00236                   inline const Key_List get_all_keys( void ) {
00237                           SCOPED_MUTEX;
00238                           Key_List ret;
00239                           for( List_cIter liter = _list.begin(); liter != _list.end
      (); liter++ )
00240                                   ret.push_back( liter->first );
00241                           return ret;
00242                   }
00243
00244           private:
00249                   inline Map_Iter _touch( const Key &key ) {
00250                           Map_Iter miter = _index.find( key );
00251                           if( miter == _index.end() ) return miter;
00252                           // Move the found node to the head of the list.
00253                           _list.splice( _list.begin(), _list, miter->second );
00254                           return miter;
00255                   }
00256
00261                   inline void _remove( const Map_Iter &miter ) {
00262                           _curr_size -= Sizefn()( miter->second->second );
00263                           _list.erase( miter->second );
00264                           _index.erase( miter );
00265                   }
00266
00270                   inline void _remove( const Key &key ) {
00271                           Map_Iter miter = _index.find( key );
00272                           _remove( miter );
00273                   }
00274 };
```

7.5 unit_test.h File Reference

Defines

- #define **UNIT_TEST_DEFINES**

  Start of inline Unit Test definitions Use this to start the list of unit tests. This should be followed by one or more DEFINE_TEST entries.

- #define **DEFINE_TEST**(test_name) bool unit_test_##test_name (void)

  Start a new test definition.

- #define **ADD_TEST**(test_name) add_test(&unit_test_##test_name);

  Adds a defined test to test run.

- #define **UNIT_TEST_RUN**(suite)

  Start a Unit test run section.

- #define **unit_assert**(msg, cond)

  Use within a Unit Test to verify a condition.

- #define **unit_pass**() return true;

  Use to end a unit test in success.

- #define **unit_fail**() return false;

  Use to end a unit test in failure.

- #define **UNIT_TEST_END**

  Finish a Unit Test run section.

Typedefs

- typedef bool(∗ **test_func**)(void)

  typedef for unittest functions

- typedef std::vector< **test_func** > **test_vector**

  typedef for vectors of unittest functions

Functions

- double **cputime** (void)

  Gets the current CPU time with microsecond accuracy.

- double **transactions_per_second** (double run_time, unsigned long transactions)

  Calculates the transactions rate.

- void **print_cputime** (const char ∗msg, double run_time, unsigned long transactions=0)

  Prints to stdout the results of timing an event.

7.5.1 Detailed Description

Unit Testing framework for C++

Author

Patrick Audley

Date

December 2004

See full documentation for this framework in **Unit Testing Framework** (p. 3)

Definition in file **unit_test.h**.

7.5.2 Define Documentation

7.5.2.1 #define DEFINE_TEST(*test_name*) bool unit_test_##test_name (void)

Start a new test definition.

Parameters

test_name Name of the test - must be unique in this unit test suite.

Definition at line **184** of file **unit_test.h**.

7.5.2.2 #define ADD_TEST(*test_name*) add_test(&unit_test_##test_name);

Adds a defined test to test run.

Parameters

test_name Test name of a previously defined test to add the the current suite.

See also

DEFINE_TEST (p. 28) **UNIT_TEST_RUN** (p. 39) This should be called after **UNIT_TEST_RUN** (p. 39) for each defined test.

Definition at line **191** of file **unit_test.h**.

7.5.2.3 #define UNIT_TEST_RUN(*suite*)

Value:

```
int main(void) { \
  bool result = true; \
  std::cout << "---[ " << suite << " ]--- " << std::endl;
```

Start a Unit test run section.

Parameters

suite Name for this test suite.

Note

Must be terminated with an UNIT_TEST_END statement.

Definition at line **197** of file **unit_test.h**.

7.5.2.4 #define unit_assert(*msg, cond*)

Value:

```
std::cout << "  " << msg << ": " << std::flush; \
  if( !cond ) { std::cout << "FAILED" << std::endl; return false; } \
  std::cout << "PASSED" << std::endl;
```

Use within a Unit Test to verify a condition.

Warning

Terminates test on failure.

Definition at line **205** of file **unit_test.h**.

Referenced by **DEFINE_TEST**().

7.5.2.5 #define unit_pass( ) return true;

Use to end a unit test in success.

Note

Either unit_pass or unit_fail should end every test.

Definition at line **213** of file **unit_test.h**.

Referenced by **DEFINE_TEST**().

7.5.2.6 #define unit_fail( ) return false;

Use to end a unit test in failure.

Note

Either unit_pass or unit_fail should end every test.

Definition at line **218** of file **unit_test.h**.

7.5.3 Function Documentation

7.5.3.1 double cputime (void) [inline]

Gets the current CPU time with microsecond accuracy.

Returns

microseconds since UNIX epoch

Definition at line **139** of file **unit_test.h**.

Referenced by **DEFINE_TEST**().

```
                                {
  getrusage( RUSAGE_SELF, &ruse );
        return ( ruse.ru_utime.tv_sec + ruse.ru_stime.tv_sec + 1e-6 * (ruse.ru_ut
      ime.tv_usec + ruse.ru_stime.tv_usec ) );
}
```

7.5.3.2 double transactions_per_second (double *run_time*, unsigned long *transactions*) [inline]

Calculates the transactions rate.

Parameters

run_time microsecond resolution run time

transactions number of transactions handled in run_time seconds This is useful if you want to guarantee minimun transactional throughputs in unit tests.

Warning

This code is obviously very test platform dependent.

Definition at line **149** of file **unit_test.h**.

Referenced by **print_cputime**().

```
        {
        return (double)transactions / run_time;
}
```

7.5.3.3 void print_cputime (const char * *msg*, double *run_time*, unsigned long *transactions* = *0*) [inline]

Prints to stdout the results of timing an event.

Parameters

msg to print with the numbers

run_time microsecond resolution run time

transactions number of transactions handled in run_time seconds, if 0 then transactional output is suppressed

Warning

This code is obviously very test platform dependent.

Definition at line **158** of file **unit_test.h**.

References **transactions_per_second**().

Referenced by **DEFINE_TEST**().

```
                {
        printf("  -> %s:  %7.3f seconds CPU time", msg, run_time );
        if( transactions != 0 )
                printf( "  (%7.3f transactions/second)", transactions_per_second(
       run_time, transactions ) );
        printf( "\n" );
}
```

Here is the call graph for this function:

7.6 unit_test.h

```
00001 /*************************************************************************
00002  *   Copyright (C) 2004-2011 by Patrick Audley                           *
00003  *   paudley@blackcat.ca                                                 *
00004  *   http://patrickaudley.com                                            *
00005  *************************************************************************/
00122 #ifndef _UNIT_TEST_H
00123 #define _UNIT_TEST_H
00124
00125 #ifdef UNITTEST
00126 #include <iostream>
00127 #include <vector>
00128 #include <sys/time.h>
00129 #include <sys/resource.h>
00130 #include <sys/types.h>
00131 #include <time.h>
00132
00133
00134 struct rusage ruse;
00135 extern int getrusage();
00139 inline double cputime( void ) {
00140   getrusage( RUSAGE_SELF, &ruse );
00141         return ( ruse.ru_utime.tv_sec + ruse.ru_stime.tv_sec + 1e-6 * (ruse.ru_ut
      ime.tv_usec + ruse.ru_stime.tv_usec ) );
00142 }
00149 inline double transactions_per_second( double run_time, unsigned long transaction
      s ) {
00150         return (double)transactions / run_time;
00151 }
```

```
00158 inline void print_cputime( const char* msg, double run_time, unsigned long transa
      ctions = 0 ) {
00159         printf("  -> %s:  %7.3f seconds CPU time", msg, run_time );
00160         if( transactions != 0 )
00161                 printf( "  (%7.3f transactions/second)", transactions_per_second(
       run_time, transactions ) );
00162         printf( "\n" );
00163 }
00164
00166 typedef bool(*test_func)(void);
00168 typedef std::vector< test_func > test_vector;
00169
00174 #define UNIT_TEST_DEFINES \
00175   test_vector * add_test( test_func x ) { \
00176     static test_vector unit_tests; \
00177     if( x != NULL ) unit_tests.push_back( x ); \
00178     return &unit_tests; \
00179   }
00180
00184 #define DEFINE_TEST(test_name) bool unit_test_##test_name (void)
00185
00191 #define ADD_TEST(test_name) add_test( &unit_test_##test_name );
00192
00197 #define UNIT_TEST_RUN( suite ) \
00198 int main(void) { \
00199   bool result = true; \
00200   std::cout << "---[ " << suite << " ]--- " << std::endl;
00201
00205 #define unit_assert( msg, cond ) \
00206   std::cout << "  " << msg << ": " << std::flush; \
00207   if( !cond ) { std::cout << "FAILED" << std::endl; return false; } \
00208   std::cout << "PASSED" << std::endl;
00209
00213 #define unit_pass() return true;
00214
00218 #define unit_fail() return false;
00219
00222 #define UNIT_TEST_END \
00223   test_vector *T = add_test( NULL ); \
00224   for( unsigned short i = 0; i < T->size(); i++ ) { \
00225     bool testresult = (*(*T)[i])(); \
00226     if( result == true && testresult == false ) { result = false; } \
00227   } \
00228   return !result; \
00229 }
00230
00231 #endif // UNITTEST
00232
00233 #endif // _UNIT_TEST_H
```

Chapter 8

Example Documentation

8.1 lru_example.cpp

```
#include "lru_cache.h"
#include <string>
#include <iostream>

int main(void) {
        // Typedef our template for easy of readability and use.
        typedef LRUCache<int,std::string> string_cache_t;

        // Instantiate a string cache with at most three elements.
        string_cache_t *cache = new string_cache_t(3);

        // Insert data into the cache.
        std::string quote_1 = "Number is the within of all things. -Pythagoras";
        cache->insert( 4, quote_1 );

        // Fetch it out.
        std::cout << cache->fetch( 4 ) << std::endl;
}
```

Index