# LRU cache implementation in C++

©2010-2012 Tim Day
timday@timday.com
www.timday.com

**Abstract**

A key-value container providing caching with a least-recently-used replacement strategy is a useful tool in any programmer's performance optimisation toolkit; however, with no ready-to-use implementations provided in the standard library or the widely used boost libraries, C++ developers are likely resort to inefficient or incorrect approximations to the logic. This document describes a couple of simple implementations built on top of the C++ standard library's map types and on the boost library's "bimap" types in the hope of making this useful pattern more accessible to coders not already familiar with it.

## Contents

## 1 Document history

- May 2012: Minor typo fixed in text.

- October 2011: Updated for C++0x; added templating to permit use of both hash-based "unordered" maps and conventional tree-based maps. Updated performance results.
- September 2011: Bug fixed in std-based code (in operator(), erroneous and unnecessary iterator assignment after splice - special thanks to Marek Fort to drawing this to the author's attention).
- March 2011: Source code licensing clarified.
- December 2010: Initial version.

## 2 Source code licensing

While the general aim of this document is to provide its audience with enough information to implement their own LRU-caching classes, readers are welcome to cut-and-paste the code listings in this document under the terms of the Internet Systems Consortium (ISC) license (an OSI-approved BSD-alike license). The following text should be considered to apply to all code in this document, and added as a comment in any code copied directly from it:

Readers simply looking for templated LRU cache code for instant usage may wish to turn straight to Listing 1 and the comments immediately preceding it.

The sources are also available from a Bitbucket-hosted Mercurial repository.

## 3 The problem

The need for caching behaviour sometimes arises during system development. Generally the desire is to preserve some expensive-to-obtain results so they can be reused "for free" without repeating the expensive operation in future. Typically the expense arises because a complex calculation is needed to obtain the result, or because it must be obtained via a time consuming I/O operation. If the total number of such results dealt with over the lifetime of the system does not consume excessive memory, it may suffice to store them in a simple key-value container (for example, a `std::map`), with the key being the input to the expensive function and the value being the result. This is often referred to as "memoisation" of a function.

However, for most applications, this approach would quickly consume too much memory to be of any practical value. The memory consumption issue can be addressed by limiting the maximum number of items stored in the cache or, if the items have a variable size, limiting the aggregate total stored. Initially the cache is empty and records (key-value pairs) can be stored in it freely. After some further usage, it will fill up. Once full, the question arises of what to do with subsequent additional records which it seems desirable to cache, but for which there is no space (given the limited capacity constraint) without taking action to remove some other records from the store. Assuming the records most recently added to the cache are those most likely to be accessed again (ie assuming some temporal coherence in the access sequence), a good general strategy is to make way for a new record by deleting the record in the cache which was "least recently used". This is called an LRU replacement strategy.

Despite the general utility of a LRU-replacement cache component, one is not provided as such in C++'s Standard Library or in the well known Boost libraries. However, it is reasonably easy logic to construct using either library as a basis.

## 4 Implementations

The following two sections describe possible implementations, both of them providing caching up to a limited number of records of a function with signature `V f(K)`, where `K` and `V` are the "key type" and the "value type". The only real difference in the API to either implementation is that the boost-based one passes the cached function as a `boost::function` rather than a function pointer, providing slightly more flexibility through compatibility with `boost::bind`.

The code presented has been tested on g++ 4.4.5. The `-std=c++0x` option is required in order to use the `std::unordered_map` hash-based map container which appeared with the TR1 additions to the C++ Standard Library, and variadic template arguments are also used to bypass some templated-template issues associated with the different type arguments to the tree-based and hash-based container types.

For brevity the code explanations below will generally refer to the `std::map` and `boost::bimaps::set_of` versions of the code; however both cache implementations work equally well (or better; see subsection 7.2) with `std::unordered_map` and `boost::bimaps::unordered_set_of` containers and the code presented is fully intended to support both, subject to the key type supporting the necessary concepts (ordered comparison in the case of `std::map` and `boost::bimaps::set_of`, hashability and equality testing in the case of the unordered variants).

## 5 Standard Library based LRU-cache

### 5.1 Design

Users of the C++ Standard Library needing an LRU-replacement cache generally gravitate towards `std::map` or `std::unordered_map`, because of their support for efficient keyed value accesses ($O(\log n)$ or $O(1)$ access complexity). The problem then is how to implement the eviction strategy.

The most obvious naive solution is to use an

```
std::map<K,std::pair<timestamp_type,V> >
```

The `timestamp_t` holds a scalar quantity, the ordering of which indicates when the value was last accessed relative to other values; typically some sort of incrementing serial number is used rather than an actual clock-derived time, to ensure a one-to-one mapping from timestamps to records. Keys then give efficient access to values and timestamps, and timestamps can be updated without the need to adjust the map's tree-structure (as this depends entirely on the key values). However, to determine the minimum timestamp in order to evict a record, it is necessary to perform a $O(n)$ search over all the records to determine the oldest one.

As a solution to eviction being expensive, it might be tempting to implement the cache as a

```
std::list<std::pair<K,V> >
```

Moving any item accessed to the tail of the list (a cheap operation for lists), ensures the least-recently-used item can trivially be obtained (for erasure) at the list head by `begin()`. However, this just moves the problem elsewhere as it is now necessary to resort to an $O(n)$ search simply to look up a key in the cache.

While either naive solution can be got to work (and may well be a simple pragmatic solution to caching a few tens of items) certainly neither can be considered scalable or efficient due to the $O(n)$ behaviour associated with either identifying eviction targets or accessing values by key.

However, it is possible to implement an LRU-replacement cache with efficient eviction *and* access using a pair of maps:

```
typedef std::map<timestamp_type,K> timestamp_to_key_type;
typedef std::map<
  K,
  std::pair<V,timestamp_type>
> key_to_value_type;
```

On accessing key_to_value by a key, we obtain access to both the value required and the timestamp, which can be updated in both the accessed record and, by lookup, timestamp_to_key. When an eviction is required, the lowest timestamp in timestamp_to_key provides the key to the record which must be erased.

However, this can actually be optimised slightly:

```
typedef std::map<timestamp_type,K> timestamp_to_key_type;
typedef std::map<
  K,
  std::pair<V,timestamp_to_key_type::iterator>
> key_to_value_type;
```

This has the slight advantage that updating the timestamp on a record access avoids an keyed lookup into timestamp_to_key, replacing it with a (presumably more efficient) direct iterator access.

Pedants will observe that further slight improvement would also have the timestamp_to_key_type map's value be an iterator into the key_to_value_type but this introduces a circular type definition. It might be tempting to try to break the dependency by using void* in place of the iterator, but iterators cannot portably be cast to pointers. In any case, the first iterator optimisation mentioned benefits the updating of timestamps needed during cache hits whereas this second iterator optimisation benefits the eviction associated with cache misses. Since in a well functioning cached system cache hits should be far more common than misses, this second optimisation is likely of much less value than the first. Another consideration is that whatever expensive operation is required to generate a new result following a cache miss is likely to be hugely more expensive than any keyed access. Therefore this second optimisation is not considered further.

In fact there *is* one final powerful optimisation possible. The only operations actually done on timestamp_to_key are to access its head (lowest timestamp, least recently used) element, or to move elements to the (most recently used) tail. Therefore it can be replaced by a std::list; this also eliminates the need for any actual instances of timestamp_type (and therefore any concerns about the timestamp possibly overflowing). In previous testing the author has found list-and-map implementations to be almost twice as fast as a version (not shown) using a pair of maps.

Finally, before showing the complete implementation, there is one other detail to consider: it would seem to be desirable to template the cache class code on the underlying container type used (std::map or std::unordered_map) rather than provide two completely separate implementations which actually only differ in minor details. This is slightly complicated by the different type argument signatures of the container types own templates (providing the optional capability to override default comparator or hasher/equality predicate logic as appropriate), but a simple workround is to declare the map type as templated on variadic type arguments `template<typename...> class MAP` and then the actual container instance required can simply be instantiated on the necessary key and value types. Generalising the code to support overriding of the default comparator (or the unordered map type equivalent) or allocators is left as an exercise for the reader.

## 5.2 Implementation

See Listing 1 for a complete example using the pair of containers

```
typedef std::list<K> key_tracker_type;
typedef MAP<
  K,std::pair<V,typename key_tracker_type::iterator>
> key_to_value_type;
```

where MAP is expected to be one of `std::map` or `std::unordered_map`; for example an LRU cache from `std::string` to `std::string` using hashing would be declared as

```
lru_cache_using_std<std::string,std::string,std::
    unordered_map>
```

while an int-keyed cache of Foo objects using the “classic” `std::map` would be declared as

```
lru_cache_using_std<int,std::shared_ptr<Foo>,std::map>
```

Note that use of a reference counted smart-pointer for heap-allocated value types is strongly advised as the very nature of caches easily leads to situations where value object ownership must be considered shared between the cache and a client which has retrieved the value from the cache.

Listing 1: lru_cache_using_std.h

```
#ifndef _lru_cache_using_std_
#define _lru_cache_using_std_

#include <cassert>
#include <list>

// Class providing fixed-size (by number of records)
// LRU-replacement cache of a function with signature
// V f(K).
// MAP should be one of std::map or std::unordered_map.
// Variadic template args used to deal with the
// different type argument signatures of those
// containers; the default comparator/hash/allocator
// will be used.
template <
  typename K,
  typename V,
  template<typename...> class MAP
  > class lru_cache_using_std
{
public:

  typedef K key_type;
  typedef V value_type;

  // Key access history, most recent at back
  typedef std::list<key_type> key_tracker_type;

  // Key to value and key history iterator
  typedef MAP<
    key_type,
    std::pair<
      value_type,
      typename key_tracker_type::iterator
      >
  > key_to_value_type;

  // Constuctor specifies the cached function and
  // the maximum number of records to be stored
  lru_cache_using_std(
    value_type (*f)(const key_type&),
    size_t c
  )
    :_fn(f)
    ,_capacity(c)
  {
    assert(_capacity!=0);
  }

  // Obtain value of the cached function for k
  value_type operator()(const key_type& k) {

    // Attempt to find existing record
    const typename key_to_value_type::iterator it
      =_key_to_value.find(k);

    if (it==_key_to_value.end()) {
      // We don't have it:

      // Evaluate function and create new record
      const value_type v=_fn(k);
      insert(k,v);

      // Return the freshly computed value
      return v;

    } else {

      // We do have it:

      // Update access record by moving
      // accessed key to back of list
      _key_tracker.splice(
        _key_tracker.end(),
        _key_tracker,
        (*it).second.second
      );

      // Return the retrieved value
      return (*it).second.first;
    }
  }

  // Obtain the cached keys, most recently used element
  // at head, least recently used at tail.
  // This method is provided purely to support testing.
  template <typename IT> void get_keys(IT dst) const {
    typename key_tracker_type::const_reverse_iterator src
        =_key_tracker.rbegin();
    while (src!=_key_tracker.rend()) {
      *dst++ = *src++;
    }
  }

private:

  // Record a fresh key-value pair in the cache
  void insert(const key_type& k,const value_type& v) {

    // Method is only called on cache misses
    assert(_key_to_value.find(k)==_key_to_value.end());

    // Make space if necessary
    if (_key_to_value.size()==_capacity)
      evict();
    // Record k as most-recently-used key
    typename key_tracker_type::iterator it
      =_key_tracker.insert(_key_tracker.end(),k);

    // Create the key-value entry,
    // linked to the usage record.
    _key_to_value.insert(
      std::make_pair(
        k,
        std::make_pair(v,it)
      )
    );
    // No need to check return,
    // given previous assert.
  }

  // Purge the least-recently-used element in the cache
  void evict() {

    // Assert method is never called when cache is empty
    assert(!_key_tracker.empty());

    // Identify least recently used key
    const typename key_to_value_type::iterator it
      =_key_to_value.find(_key_tracker.front());
    assert(it!=_key_to_value.end());

    // Erase both elements to completely purge record
    _key_to_value.erase(it);
    _key_tracker.pop_front();
  }

  // The function to be cached
  value_type (*_fn)(const key_type&);

  // Maximum number of key-value pairs to be retained
  const size_t _capacity;

  // Key access history
  key_tracker_type _key_tracker;

  // Key-to-value lookup
  key_to_value_type _key_to_value;
};

#endif
```

Readers interested in the performance impact of choosing `std::map` vs. `std::unordered_map` will find some results in subsection 7.2.

## 6 Boost bimap-based LRU-cache

### 6.1 Design

Assuming usage of the boost C++ libraries is an option, `boost::bimap` provides an excellent basis on which to construct LRU-replacement cache logic. Unlike the better known `std::map`, which has an asymmetric relation between key and value (that is, a key can index a value but a value can't index a key), `boost::bimap` supports a bidirectional mapping between a pair of keys. This is very convenient for indexing a cache's collection using either the cached-function argument key or the key-access history as required, and much of the complication of the standard-library version described in section 5 can be avoided.

As with the standard library based implementation above, it is not actually necessary to record timestamps explicitly and it suffices for the access-history view of the bimap to be a `list_of`, which can maintain the ordering of keys accessed. By default `bimap` inserts new records at the tail of the list view, and records can be efficiently relocated there on access.

There are several ways in which the boost::bimap container to be used might be instantiated. The most obvious one is

```
typedef boost::bimaps::bimap<
  boost::bimaps::set_of<K>,
  boost::bimaps::list_of<V>
> container_type;
```

with the left view providing the key-to-value semantics, and the ordering on the right view used to maintain a record of the key-value-pair access history. Replacing the `set_of` with `unordered_set_of` to use hashing instead of trees is another choice.

However, there also exists the possibility of declaring

```
typedef boost::bimaps::bimap<
  boost::bimaps::set_of<K>,
  boost::bimaps::list_of<dummy_type>,
  boost::bimaps::with_info<V>
> container_type;
```

with the cached value stored using bimap's support for "additional information" and `dummy_type` never storing a meaningful value. In the past the author has found some peculiar performance differences between implementations using a `with_info` version of the container and those without, but this seems to no longer be the case so only the simpler version without `with_info` is considered further here.

## 6.2 Implementation

See Listing 2 for a complete implementation. This is similar to the standard library-based version in that it is templated on the key view template to be used which is expected be one of `boost::bimaps::set_of` or `boost::bimaps::unordered_set_of`. For example an LRU cache from `std::string` to `std::string` using hashing would be declared as

```
lru_cache_using_boost<std::string,std::string,boost::
    bimaps::unordered_set_of>
```

Listing 2: lru_cache_using_boost.h

```
#ifndef _lru_cache_using_boost_
#define _lru_cache_using_boost_

#include <boost/bimap.hpp>
#include <boost/bimap/list_of.hpp>
#include <boost/bimap/set_of.hpp>
#include <boost/function.hpp>
#include <cassert>

// Class providing fixed-size (by number of records)
// LRU-replacement cache of a function with signature
// V f(K).
// SET is expected to be one of boost::bimaps::set_of
// or boost::bimaps::unordered_set_of
template <
  typename K,
  typename V,
  template <typename...> class SET
  > class lru_cache_using_boost
{
 public:

  typedef K key_type;
  typedef V value_type;

  typedef boost::bimaps::bimap<
    SET<key_type>,
    boost::bimaps::list_of<value_type>
    > container_type;

  // Constuctor specifies the cached function and
  // the maximum number of records to be stored.
  lru_cache_using_boost(
    const boost::function<value_type(const key_type&)>& f,
    size_t c
```

```
)
  :_fn(f)
  ,_capacity(c)
{
  assert(_capacity!=0);
}

// Obtain value of the cached function for k
value_type operator()(const key_type& k) {

  // Attempt to find existing record
  const typename container_type::left_iterator it
    =_container.left.find(k);

  if (it==_container.left.end()) {

    // We don't have it:

    // Evaluate function and create new record
    const value_type v=_fn(k);
    insert(k,v);

    // Return the freshly computed value
    return v;

  } else {

    // We do have it:

    // Update the access record view.
    _container.right.relocate(
      _container.right.end(),
      _container.project_right(it)
    );

    // Return the retrieved value
    return it->second;
  }
}

// Obtain the cached keys, most recently used element
// at head, least recently used at tail.
// This method is provided purely to support testing.
template <typename IT> void get_keys(IT dst) const {
  typename container_type::right_const_reverse_iterator
      src
       =_container.right.rbegin();
  while (src!=_container.right.rend()) {
    *dst++=(*src++).second;
    }
  }

 private:

  void insert(const key_type& k,const value_type& v) {

    assert(_container.size()<=_capacity);

    // If necessary, make space
    if (_container.size()==_capacity) {
      // by purging the least-recently-used element
      _container.right.erase(_container.right.begin());
    }

    // Create a new record from the key and the value
    // bimap's list_view defaults to inserting this at
    // the list tail (considered most-recently-used).
    _container.insert(
      typename container_type::value_type(k,v)
    );
  }

  const boost::function<value_type(const key_type&)> _fn;
  const size_t _capacity;
  container_type _container;
};

#endif
```

See subsection 7.2 for some comparative performance testing between the tree and hash based versions, and between bimap-based caches and conventional standard library map based caches.

## 7 Testing

Both test codes below make use of the header Listing 3 which brings together both standard library and boost bimap based implementations and defines some helper types. This permits, for example

```
  lru_cache_using_boost<std::string,std::string,boost::
      bimaps::unordered_set_of>
```

to be written

```
  lru_cache_using_boost_unordered_set<std::string,std::
      string>::type
```

Listing 3: lru_cache.h

```
#ifndef _lru_cache_h_
#define _lru_cache_h_

// Bring all the necessary includes together,
// and define some type helpers.

#include <map>
#include <unordered_map>
#include "lru_cache_using_std.h"

#include <boost/bimap/set_of.hpp>
#include <boost/bimap/unordered_set_of.hpp>
#include "lru_cache_using_boost.h"

// See http://www.gotw.ca/gotw/079.htm for why we can't
// just use a templated typedef.

template <typename K,typename V> struct
    lru_cache_using_std_map {
  typedef lru_cache_using_std<K,V,std::map> type;
};

template <typename K,typename V> struct
    lru_cache_using_std_unordered_map {
  typedef lru_cache_using_std<K,V,std::unordered_map> type;
};

template <typename K,typename V> struct
    lru_cache_using_boost_set {
  typedef lru_cache_using_boost<K,V,boost::bimaps::set_of>
      type;
};

template <typename K,typename V> struct
    lru_cache_using_boost_unordered_set {
  typedef lru_cache_using_boost<K,V,boost::bimaps::
      unordered_set_of> type;
};

#endif
```

## 7.1 Unit test

Listing 4 contains templated `boost::test` code to exercise all four cache implementations and check basic behaviour is as expected.

Listing 4: lru_test.cpp

```
#define BOOST_TEST_DYN_LINK
```

```
#define BOOST_TEST_MODULE lru_test

#include <iostream>
#include <string>
#include <vector>

#include <boost/test/unit_test.hpp>
#include <boost/test/test_case_template.hpp>
#include <boost/mpl/list.hpp>

#include "lru_cache.h"

namespace {size_t count_evaluations=0;}

// Dummy function we want to cache
std::string fn(const std::string& s)
{
  ++count_evaluations;
  std::string r;
  std::copy(s.rbegin(),s.rend(),std::back_inserter(r));
  return r;
}

typedef boost::mpl::list<
  lru_cache_using_std_map<std::string,std::string>::type,
  lru_cache_using_std_unordered_map<std::string,std::string
     >::type,
  lru_cache_using_boost_set<std::string,std::string>::type,
  lru_cache_using_boost_unordered_set<std::string,std::
     string>::type
  > test_types;

BOOST_AUTO_TEST_CASE_TEMPLATE
(
  lru_test,
  CACHE,
  test_types
)
{
  count_evaluations=0;

  CACHE lru(fn,5);

  // Some initial accesses to prime state
  BOOST_CHECK_EQUAL(lru("first"),"tsrif");
  BOOST_CHECK_EQUAL(lru("second"),"dnoces");
  BOOST_CHECK_EQUAL(lru("third"),"driht");
  BOOST_CHECK_EQUAL(lru("fourth"),"htruof");
  BOOST_CHECK_EQUAL(lru("fifth"),"htfif");
  BOOST_CHECK_EQUAL(count_evaluations,5);
  BOOST_CHECK_EQUAL(lru("sixth"),"htxis");
  BOOST_CHECK_EQUAL(count_evaluations,6);

  // This should be retrieved from cache
  BOOST_CHECK_EQUAL(lru("second"),"dnoces");
  BOOST_CHECK_EQUAL(count_evaluations,6);

  // This will have been evicted
  BOOST_CHECK_EQUAL(lru("first"),"tsrif");
  BOOST_CHECK_EQUAL(count_evaluations,7);

  // Cache contents by access time
  // (most recent to least recent)
  // should now be:
  // first,second,sixth,fifth,fourth
  {
    std::vector<std::string> expected;
    expected.push_back("first");
    expected.push_back("second");
    expected.push_back("sixth");
    expected.push_back("fifth");
    expected.push_back("fourth");
    std::vector<std::string> actual;
    lru.get_keys(std::back_inserter(actual));
    BOOST_CHECK(actual==expected);
  }

  // So check fourth is retrieved
  BOOST_CHECK_EQUAL(lru("fourth"),"htruof");
  BOOST_CHECK_EQUAL(count_evaluations,7);

  // That will have moved up "fourth" to the head
  // so this will evict fifth
  BOOST_CHECK_EQUAL(lru("seventh"),"htneves");
  BOOST_CHECK_EQUAL(count_evaluations,8);

  // Check fifth was evicted as expected
  BOOST_CHECK_EQUAL(lru("fifth"),"htfif");
  BOOST_CHECK_EQUAL(count_evaluations,9);
}
```

## 7.2 Performance testing

Listing 5 contains test code to examine the performance of all four cache implementations. It makes use of a timer utility class which is provided in Listing 6 (NB this makes use of the TR1 “chrono” addition to the standard library).

In order to test only pure cache performance, a trivial operation is used for the cached function. The code tests a 1024 element capacity cache with a series of random accesses over a smaller or larger range. When the the load is less than or equal to $100\%$, accesses are guaranteed to find a record already in the cache, and the performance reflects the cost of simply finding it and updating the access timestamp. When the load is over $100\%$, the excess determines the expected proportion of accesses which will result in cache misses; these have a higher cost as deletion of a previous record and creation of a new one is required and this is more complex than simply adjusting a timestamp. Hence in the $112.5\%$ load test, one in nine accesses are expected to be a miss, and in the $200\%$ load case, on average half of all accesses should miss.

Two key-value types are tested:

- `int` to `int`: Random keys over the full integer range, with the cached function simply being the identity.
- `std::wstring` to `int`: Random 256 character strings (of `wchar_t`), with the cached function simply being the string length.

Table 1 and Table 2 shows performance results obtained on a 2.66GHz Intel Q9450 Core2 running Debian Squeeze (32bit) when compiled with

```
g++ -std=c++0x -march=native -O3 -DNDEBUG -o lru_perf
    lru_perf.cpp -lboost_unit_test_framework
```

Software versions are g++ 4.4.5, boost 1.42.

Table 1: int-to-int cache performance test results (Mega-accesses per second)

| Cache load | Standard Library container type | | boost bimap key view type | |
|---|---|---|---|---|
| | `map` | `unordered_map` | `set_of` | `unordered_set_of` |
| 50% | 14.4 | 41.8 | 13.4 | 36.3 |
| 87.5% | 11.6 | 35.1 | 11.9 | 34.4 |
| 100% | 11.6 | 34.4 | 11.9 | 34.8 |
| 112.5% | 7.78 | 17.2 | 8.91 | 23.2 |
| 200% | 3.72 | 7.53 | 4.87 | 11.6 |

The main conclusions which can be drawn from these results are:

- Performance differences between hash-based and tree-based containers can be significant. This should hardly be surprising given the different algorithmic complexity ($O(\log n)$ vs. $O(1)$) of key lookup.
- The string-key results demonstrate that hash based caches are not automatically better. In fact this test case was constructed to

Table 2: 256-character std::wstring-to-int cache performance test results (Mega-accesses per second)

| Cache load | Standard Library container type | | boost bimap key view type | |
|---|---|---|---|---|
| | map | unordered_map | set_of | unordered_set_of |
| 50% | 1.66 | 0.515 | 1.56 | 1.1 |
| 87.5% | 1.58 | 0.515 | 1.49 | 1.09 |
| 100% | 1.57 | 0.513 | 1.49 | 1.09 |
| 112.5% | 1.4 | 0.422 | 1.49 | 1.02 |
| 200% | 0.996 | 0.261 | 1.48 | 0.815 |

probe the “cross over point” at which a tree-based container becomes more efficient than caching; for shorter string lengths the balance shifts back in favour of hashing.

- Performance differences between the boost and standard library based implementations are not so simple to declare one or the other as obviously superior. It might be possible to make the generalisation that the standard library based performs better when lightly loaded (few cache misses) while the boost implementation performs better under heavy load (high proportion of cache misses), but this is not actually true over all results obtained.

Of course these results should not be taken as a substitute for readers carrying out performance testing on their own systems using realistic test cases relevant to their own usage.

Listing 5: lru_perf.cpp

```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE lru_perf

#include <boost/format.hpp>
#include <boost/lexical_cast.hpp>
#include <boost/test/unit_test.hpp>
#include <boost/test/test_case_template.hpp>
#include <boost/mpl/list.hpp>
#include <boost/random.hpp>
#include <iostream>
#include <set>
#include <vector>

#include "lru_cache.h"
#include "scoped_timer.h"

// Random number source for generating test data
```

```
boost::mt19937 rng(/*seed=*/23);
boost::random_number_generator<
  boost::mt19937,size_t
  > rnd(rng);

// Functions to create collections of test keys

// General pattern
template <typename K> std::vector<K> make_keys(size_t);

// Integer specialization:
// generate n unique integer keys
template <> std::vector<int> make_keys<int>(size_t n)
{
  std::set<int> r;
  do {
    r.insert(static_cast<int>(rng()));
  } while (r.size()<n);
  return std::vector<int>(r.begin(),r.end());
}

// String specialization:
// generate n unique long strings
template <> std::vector<std::wstring> make_keys<std::
    wstring>(size_t n)
{
  // Sufficiently long key lengths favour comparator
  // based containers over hashed
  const size_t keylength=256;

  std::set<std::wstring> r;
  do {
    std::wstring s;
    for (size_t i=0;i<keylength;++i)
      s+=static_cast<wchar_t>(1+rnd(65535));
    r.insert(s);
  } while(r.size()<n);
  return std::vector<std::wstring>(r.begin(),r.end());
}

// Extract relevant part of __PRETTY_FUNCTION__ text
std::string shrink_pretty_function_text(
  const std::string& s
) {
  std::string r;
  int c=0;
  for (size_t i=s.find("=")+2;i<s.size()-1;++i) {
    if (c<=1) r+=s[i];
    if (s[i]=='<') ++c;
    if (s[i]=='>') {if (c==2) r+="...>";--c;}
  }
  return r;
}

// Test the cache object (capacity n) with k keys
// and a accesses.  Benchmark adapts to CACHE::key_type,
// but expects an int-like CACHE::value_type
template <typename CACHE> void benchmark(
  CACHE& lru,
  size_t n,
  size_t k,
  size_t a
) {

  typedef typename CACHE::key_type key_type;

  // The set of m keys to be used for testing
  std::vector<key_type> keys=make_keys<key_type>(k);

  // Create a random access sequence plus some "primer"
  std::vector<key_type> seq;
  for (size_t i=0;i<a+n;++i)
    seq.push_back(keys[i%k]);
  std::random_shuffle(seq.begin(),seq.end(),rnd);

  // Prime the cache so timed results reflect
  // "steady state", not ramp-up period.
  for (size_t i=0;i<n;++i)
    lru(seq[i]);

  // Setup timing
  scoped_timer timer(
    boost::str(
      boost::format("%1%␣load␣%|2$4.1f|%%")
      % shrink_pretty_function_text(__PRETTY_FUNCTION__)
      % (k*100.0/n)
    ),
    "Maccesses",
    a/static_cast<float>(1<<20)
  );

  // Run the access sequence
  int t=0;
  for (
    typename std::vector<key_type>::const_iterator it=
      seq.begin()+n;
    it!=seq.end();
    ++it
  )
    t+=lru(*it);

  // Avoid "optimised away"
  volatile int force=0;
  force=t;
}

// Classes we want to test
typedef boost::mpl::list<
  lru_cache_using_std_map<int,int>::type,
  lru_cache_using_std_unordered_map<int,int>::type,
  lru_cache_using_boost_set<int,int>::type,
  lru_cache_using_boost_unordered_set<int,int>::type,

  lru_cache_using_std_map<std::wstring,int>::type,
  lru_cache_using_std_unordered_map<std::wstring,int>::type
      ,
  lru_cache_using_boost_set<std::wstring,int>::type,
  lru_cache_using_boost_unordered_set<std::wstring,int>::
      type
> test_types;

// Some lightweight functions to cache
// for the key types of interest

template <typename K> int fn(const K&);

template <> int fn<int>(const int& x) {
  return x;
}

template <> int fn<std::wstring>(const std::wstring& x) {
  return x.size();
}

BOOST_AUTO_TEST_CASE_TEMPLATE
(
  lru_perf,
  CACHE,
  test_types
) {
  const size_t n=1024;

  CACHE lru(fn<typename CACHE::key_type>,n);

  // Test a variety of cache load levels
  benchmark(lru,n,n/2    ,1024*n);
  benchmark(lru,n,(7*n)/8,1024*n);
  benchmark(lru,n,n      ,1024*n);
  benchmark(lru,n,(9*n)/8,1024*n);
  benchmark(lru,n,2*n    ,1024*n);
  std::cout << std::endl;
}
```

## 8 Variations

There are numerous ways in which basic caching behaviour can be extended. Here are a few suggestions.

### 8.1 Definition of "capacity"

In many applications for caches, it is required to define the cache capacity not simply as a maximum number of cached elements, but rather in terms of the total of some measure on the elements. For example, a utility for browsing a collection of image files might include a cache of loaded raster images keyed by filename so as to avoid re-loading the most recently viewed images if the user revisits them. A cache which simply stored a fixed number of images would not have very predictable (ie bounded) memory consumption due to the enormous variation in image sizes which could be encountered. However, by taking the size-in-bytes of the stored images into account, the maximum total memory consumed by cached objects could be controlled and the number of images cached automatically adjusted in response to the aggregate size.

Implementation of such a cache is left as an exercise for the reader. Hint: It will be necessary to provide some mechanism (another function object?) for determining the capacity consumed by each stored record, to track the aggregate total currently cached, and to call the `evict()` method within a loop to possibly free up multiple elements in order to make sufficient space for insertion of a single large elements.

### 8.2 Thread safety

It should be obvious how to trivially make a cache implementation thread-safe by addition of a suitable mutex as a member and locking it for the scope of the access operation. Bear in mind that caches can easily become a serialisation bottleneck because, unlike simpler containers with no state mutated on read, multiple readers cannot simply ignore each other as even a cache-hitting read access must update the access record. Possibly some benefit might be gained from using an upgradeable lock to limit the time for which exclusive access is required to the minimum. For cache hits, exclusivity can be limited to the access history update, but for cache misses the situation is more complicated because several non-exclusive threads could cache miss the same key simultaneously and it is probably desirable for the exclusive-scope code to handle this case efficiently and ideally only compute the corresponding value once.

In some cases the best solution might be for multiple threads to each maintain their own independent cache: the inefficiency introduced by duplication of some cache-miss calculations must be weighed against the overheads introduced by sharing a cache between multiple threads.

## 8.3 Compressed stored values

Depending on the nature of the stored result values in the cache, it may be possible to reduce the amount of memory used by the cache (or, more usefully, cache more records in the same amount of memory) by compressing the result values. Storing a value in the cache will incur a compression overhead, and retrieving each value will require a decompression, but if the costs of these is small compared with the original operation required to obtain the value then it may be worth the expense to obtain an overall improvement in system performance through a better cache hit rate.

A further refinement might be two levels of caching; one holding compressed values, and a smaller one holding the most recent decompressed values in anticipation of near-term reuse.

See also `boost::flyweight`, which may be of use as a cache-size reducer in situations where many records in the cache actually hold the same value (this would imply the cached function is “many-to-one” in character).

## 8.4 Alternative strategies to LRU

For some specialist use cases, alternative strategies might be appropriate. For example, a most-recently-used (MRU) replacement strategy evicts the record most recently added to the cache; this might be relevant where cached results generated earliest in the lifetime of a program are most likely to be reused, or it can be a useful strategy to switch to in cases where a program repeatedly loops over a sequence of records slightly too large for the cache to store fully. Alternatively, a cache might have access to additional information (a so-called “clairvoyant algorithm”) which predicts the cached values to be used in future and thereby be able to select records for eviction so as to minimise the amount of recalculation subsequently needing to be done.

## 8.5 Pooling

Consider a fully populated cache, which should be considered to be the normal state for caches after they have been in existence for some time. Cache misses result in the deletion of an evicted stored result value, followed by the immediate creation of a new one. If the value type is particularly expensive to create there may be a potential optimisation in recycling evicted values through a simple object pool for more efficient reuse.

## A Timer utility class

Listing 6 contains the timer utility class used by the performance testing code in Listing 5.

Listing 6: scoped_timer.h

```
#ifndef _scoped_timer_h_
#define _scoped_timer_h_

#include <boost/noncopyable.hpp>
#include <chrono>
#include <iostream>

// Utility class for timing and logging rates
// (ie "things-per-second").
// NB _any_ destructor invokation (including early return
// from a function or exception throw) will trigger an
// output which will assume that whatever is being measured
// has completed successfully and fully.
class scoped_timer : boost::noncopyable
{
public:

  typedef std::chrono::high_resolution_clock clock_type;

  scoped_timer(
    const std::string& what,
    const std::string& units,
    double n
  )
    :_what(what)
    ,_units(units)
    ,_how_many(n)
    ,_start(clock_type::now())
  {}

  ~scoped_timer() {
    clock_type::time_point stop=clock_type::now();
    const double t=
      1e-9*std::chrono::duration_cast<
        std::chrono::nanoseconds
      >(stop-_start).count();
    std::cout << (
      boost::format(
        "%1%:␣%|2$-5.3g|␣%|3$|/s␣(%|4$-5.3g|s)"
      ) % _what % (_how_many/t) % _units % t
    ) << std::endl;
  }

private:

  const std::string _what;
  const std::string _units;
  const double _how_many;
  const clock_type::time_point _start;
};

#endif
```