ORDERED VERSUS UNORDERED MAP FOR PRIMITIVE DATA TYPES

Tom Nealis

September 2015

U.S. ARMY ARMAMENT RESEARCH, DEVELOPMENT AND ENGINEERING CENTER
Weapons and Software Engineering Center
Picatinny Arsenal, New Jersey

Approved for public release; distribution is unlimited.
The views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

The citation in this report of the names of commercial firms or commercially available products or services does not constitute official endorsement by or approval of the U.S. Government.

Destroy this report when no longer needed by any method that will prevent disclosure of its contents or reconstruction of the document. Do not return to the originator.
A map in programming is an associative container consisting of some key value mapped to some element. C++ provides two types of map containers within the standard template library, the std::map and the std::unordered_map classes. As the name implies, the containers main functional difference is that the elements in the std::map are ordered by the key, and the std::unordered_map are not ordered based on their key. The std::unordered_map elements are placed into “buckets” based on a hash value computed for their key. This report will concentrate on the performance difference of these two containers using a primitive data type for the key.
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Methodology</td>
<td>1</td>
</tr>
<tr>
<td>Conclusions</td>
<td>7</td>
</tr>
<tr>
<td>Distribution List</td>
<td>9</td>
</tr>
</tbody>
</table>
INTRODUCTION

A map in computer science programming is an associative container consisting of a key value mapped to some element. The C++ programming language provides two types of map containers within the standard template library, the std::map and the std::unordered_map classes. The main difference between these two containers is the way in which the elements are stored. In the std::unordered_map, the elements are not stored in order by the key. The std::unordered_map elements are placed into “buckets” based on a hash value computed for their key.

The underlying structure of a std::map is typically a binary search tree. The std::map is thought to be generally slower than unordered maps but certainly have their use if ordered access is necessary. The std::unordered_map is stored in a hash table. This allows for faster access to elements based on a hash computation done on the key value. This computed value is then used to look up the location of the element.

METHODOLOGY

In order to test the performance of these containers for comparison, a small program was written to measure and then log the execution time of certain tasks involving the containers. The main concentration was on some of the more common tasks such as inserting, finding, erasing, and traversal. The following is the program written to accomplish this:

```cpp
int _tmain(int argc, _TCHAR* argv[]) 
{ 
    LARGE_INTEGER frequency; 
    QueryPerformanceFrequency(&frequency); 

    LARGE_INTEGER starting_time, ending_time, elapsed_microseconds; 

    srand(static_cast<unsigned int>(time(nullptr))); 

    std::ofstream a_file("outfile.txt"); 

    std::map<unsigned int, unsigned int> ordered_map; 
    std::unordered_map<unsigned int, unsigned int> unordered_map; 

    std::vector<unsigned int> my_ints; 
    std::vector<unsigned int> find_ints; 

    auto intergers = 0u; 
    for(auto j = 0u; j < intergers; ++j) 
    { 
        intergers += 100000; 

        //clear all vectors and arrays 
        my_ints.clear(); 
        find_ints.clear(); 
        ordered_map.clear(); 
        unordered_map.clear(); 

        //populate vector 
        for(auto i = 0u; i < intergers; ++i) 
            my_ints.push_back(rand() % intergers); 
    }
}
```
//add 10 elements to find
for(auto i = 0; i < 10; ++i)
    find_ints.push_back(rand() % intergers);

//populate ordered map
QueryPerformanceCounter(&starting_time);
for(const auto &j : my_ints)
    ordered_map.insert(std::pair<unsigned int, unsigned int>(j, j));
QueryPerformanceCounter(&ending_time);
elapsed_microseconds.QuadPart = ending_time.QuadPart - starting_time.QuadPart;

auto time_elapsed_ord_insert = static_cast<double>((elapsed_microseconds.QuadPart * 1000000.0) / frequency.QuadPart);
//printf("Ordered Map Insert took: %4.2f microseconds\n", time_elapsed_ord_insert);

//populate unordered map
QueryPerformanceCounter(&starting_time);
for(const auto &j : my_ints)
    unordered_map.insert(std::pair<unsigned int, unsigned int>(j, j));
QueryPerformanceCounter(&ending_time);
elapsed_microseconds.QuadPart = ending_time.QuadPart - starting_time.QuadPart;

auto time_elapsed_unordered_insert = static_cast<double>((elapsed_microseconds.QuadPart * 1000000.0) / frequency.QuadPart);
//printf("Unordered Map Insert took: %4.2f microseconds\n", time_elapsed_unordered_insert);

//find some ints in ordered map
QueryPerformanceCounter(&starting_time);
for(const auto &j : find_ints)
    auto it = ordered_map.find(j);
QueryPerformanceCounter(&ending_time);
elapsed_microseconds.QuadPart = ending_time.QuadPart - starting_time.QuadPart;

auto time_elapsed_ordered_find = static_cast<double>((elapsed_microseconds.QuadPart * 1000000.0) / frequency.QuadPart);
//printf("Ordered Map Find took: %4.2f microseconds\n", time_elapsed_ordered_find);

//find the same ints in unordered map
QueryPerformanceCounter(&starting_time);
for(const auto &j : find_ints)
    auto it = unordered_map.find(j);
QueryPerformanceCounter(&ending_time);
elapsed_microseconds.QuadPart = ending_time.QuadPart - starting_time.QuadPart;
auto time_elapsed_unordered_find = static_cast<double>((elapsed_microseconds.QuadPart * 1000000.0) / frequency.QuadPart);
//printf("Unordered Map Find took: %.2f microseconds\n", time_elapsed_unordered_find);

// erase some ints in ordered map
QueryPerformanceCounter(&starting_time);
for(const auto &j : find_ints)
  auto it = ordered_map.erase(j);
QueryPerformanceCounter(&ending_time);
elapsed_microseconds.QuadPart = ending_time.QuadPart - starting_time.QuadPart;
auto te_ordered_delete = static_cast<double>((elapsed_microseconds.QuadPart * 1000000.0) / frequency.QuadPart);

// erase the same ints in unordered map
QueryPerformanceCounter(&starting_time);
for(const auto &j : find_ints)
  auto it = unordered_map.erase(j);
QueryPerformanceCounter(&ending_time);
elapsed_microseconds.QuadPart = ending_time.QuadPart - starting_time.QuadPart;
auto te_unordered_delete = static_cast<double>((elapsed_microseconds.QuadPart * 1000000.0) / frequency.QuadPart);

// range ordered map
QueryPerformanceCounter(&starting_time);
auto it = ordered_map.begin();
for(const auto &j : ordered_map) ;
QueryPerformanceCounter(&ending_time);
elapsed_microseconds.QuadPart = ending_time.QuadPart - starting_time.QuadPart;
auto te_ordered_range = static_cast<double>((elapsed_microseconds.QuadPart * 1000000.0) / frequency.QuadPart);

// range unordered map
QueryPerformanceCounter(&starting_time);
for(const auto &j : unordered_map) ;
QueryPerformanceCounter(&ending_time);
elapsed_microseconds.QuadPart = ending_time.QuadPart - starting_time.QuadPart;
auto te_unordered_range = static_cast<double>((elapsed_microseconds.QuadPart * 1000000.0) / frequency.QuadPart);
This program first populates two standard vectors with random numbers. The first one stores integers that will be inserted into the containers. The second one consists of 10 random numbers used for finding values in the maps. The next step was to measure how long it took to insert all the values into the containers. Then, the time it took to find all 10 values was recorded, and then the time it took for the removal of all 10 of those values was recorded. Lastly, the traversal of the entire map was timed. Figures 1 through 4 show the measurements that were logged.
Figure 1
Map insert

Figure 2
Map find
Figure 3
Map erase

Figure 4
Map range access
CONCLUSIONS

The std::unordered_map outperformed the std::map in all categories measured. If it is not necessary to have the elements sorted, using the std::unordered_map class is suggested. Using the std::unordered_map class will make your program faster and more efficient.
REVIEW AND APPROVAL OF ARDEC TECHNICAL REPORTS

Ordered Vs. Unordered Map for Primitive Data Types

Title

Date received by LCSD

Thomas M. Nealis
Author/Project Engineer

Report number (to be assigned by LCSD)

X8048 31

RDAR-WSF-M

Extension Building

Author's/Project Engineers Office
(Division, Laboratory, Symbol)

PART 1. Must be signed before the report can be edited.

a. The draft copy of this report has been reviewed for technical accuracy and is approved for editing.

b. Use Distribution Statement A_X__ B__ C__ D__ E__ F__ or X__ for the reason checked on the continuation of this form. Reason: ___ Operational Use___________

1. If Statement A is selected, the report will be released to the National Technical Information Service (NTIS) for sale to the general public. Only unclassified reports whose distribution is not limited or controlled in any way are released to NTIS.

2. If Statement B, C, D, E, F, or X is selected, the report will be released to the Defense Technical Information Center (DTIC) which will limit distribution according to the conditions indicated in the statement.

c. The distribution list for this report has been reviewed for accuracy and completeness.

Patricia Alameda
Division Chief

9/10/05

(Part 2)

To be signed either when draft report is submitted or after review of reproduction copy.

This report is approved for publication.

Patricia Alameda
Division Chief

9/10/05

Andrew Pskowski

RDAR-CIS

9/10/05

LCSD 49 supersedes SMCAI Form 49, 20 Dec 06