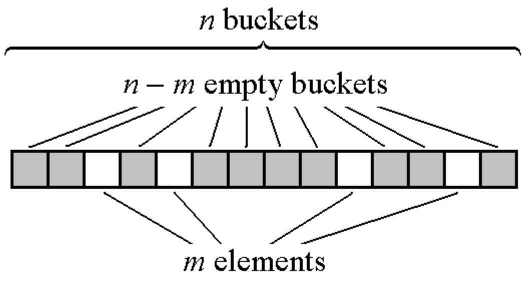
Document no: N2023=06-0093
Date: 2006-06-13
Project: Programming Language C++
Reply to: Joaquín Mª López Muñoz
joaquin@tid.es

`erase(iterator)` for unordered containers should not return an iterator

In section 6.3.1, Table 21 of the TR1 specification[1], it is required that the expression `a.erase(q)`, where `q` is an iterator to an element of the unordered container `a`, return the iterator immediately following `q` prior to the erasure. A similar condition exists for `const_iterator`s. We argue that this requirement makes it impossible to achieve average $O(1)$ complexity on the erase operation for, at least, a common implementation of unordered containers.

In unordered containers using singly linked buckets, incrementing an iterator at the end of a bucket implies traversing the internal hash table until the next nonempty bucket is reached. Going from one bucket to the other requires then a number of "hops" through the hash table determined by the distance between both buckets. Let us calculate the amortized number of hops due to iterator increment performed when executing the operation `a.erase(q)`. We begin with a container of size n where elements are randomly erased one by one until the container is empty, and calculate the expected total number of hops $h(n)$ resulting from the n erasure operations; the amortized complexity of erasure is then lower bounded by the average number of hops $h(n)/n$. For simplicity, we assume that the initial load factor is 1.0 and that the hashing scheme is perfect and there are no collisions.



Consider the general situation where there are m randomly distributed elements left, and $n - m$ empty buckets (see figure above). The hash table has then $m + 1$ empty intervals (including those of length zero between adjacent elements) with an average length $d = (n - m)/(m + 1)$. Erasing a randomly chosen element and incrementing to the next one (or the end of the container) implies reaching for the next nonempty bucket in $1 + d = (n + 1)/(m + 1)$ hops. The expected total number of hops $h(n)$ is then

[1] Matthew Austern, "Draft Tecnhical Report on C++ Library Extensions," WG21 Document N1836=05-0096, 2005.

$$h(n) = \sum_{m=1}^{n} \frac{n+1}{m+1} = (n+1)\sum_{m=1}^{n} \frac{1}{m+1} = (n+1)(H_{n+1} - 1) \approx$$
$$\approx \frac{1}{2} + (n+1)(\ln(n+1) + \gamma - 1),$$

where H_n is the n-th *harmonic number*[2] and $\gamma = 0.5772\ldots$ is the *Euler-Mascheroni constant*[3]. This formula has been empirically verified by computer simulation. The average number of hops $h(n)/n$ is then $O(\log n)$, which constitutes a lower bound for the complexity of `a.erase(q)`, in contradiction with the average $O(1)$ requirement in TR1. In pathological cases, $h(n)$ can be as bad as ½ $n(n + 1)$: this happens when the elements are erased in reverse order as they are arranged in the hash table:

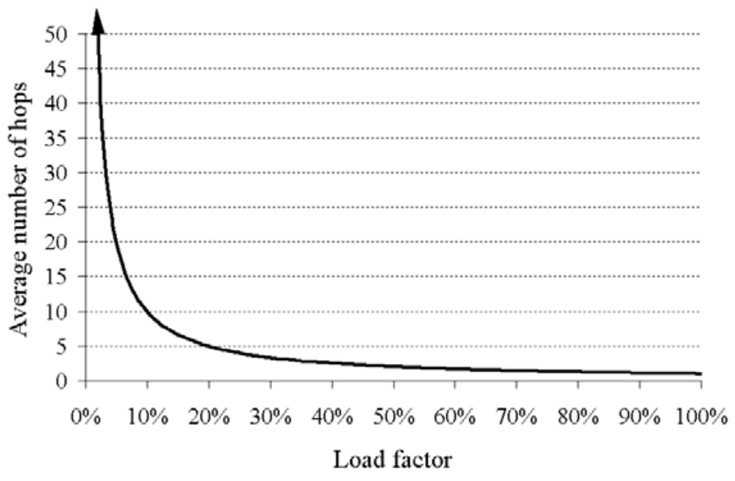
```
unordered_set<unsigned int> s;
for(unsigned int m=0; m<n; ++m) s.insert(m);

// quadratic behavior in some implementations
for(unsigned int m=n; m--;) s.erase(s.find(m));
```

Impact on actual performance
The formulas above show that the average number of hops per erasure operation can be quite high, which hints at the possibility that hopping has a measurable impact on the performance of actual programs.

Maximum size of the container	Average number of hops per erasure operation
1,000	6.5
10,000	8.8
100,000	11.1
1,000,000	13.4

However, hopping overhead is unevenly distributed with respect to the load factor, which decreases as the table empties, so that most of that overhead concentrates on the area where the table is nearly void, as depicted in the figure.



[2] Eric W. Weisstein et al. "Harmonic Number." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/HarmonicNumber.html
[3] Eric W. Weisstein. "Euler-Mascheroni Constant." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/Euler-MascheroniConstant.html

It can be alleged then that under normal operating conditions, i.e. when the table is almost full, hopping overhead is negligible. To ascertain this point, a test program has been written in which, under conditions simulating those of a real application, an unordered container is randomly depleted down to load factors around 10%. This program has been run against two versions of unordered container, differing only in that the first one returns an iterator on erasure and the second one does not; the results show that the latter version runs around 8% faster than the former.

Proposed resolution 1

The problem can be eliminated by omitting the requirement that `a.erase(q)` return an iterator. This is, however, in contrast with the equivalent requirements for other standard containers.

Proposed resolution 2

`a.erase(q)` can be made to compute the next iterator only when explicitly requested: the technique consists in returning a proxy object implicitly convertible to `iterator`, so that

```
iterator q1=a.erase(q);
```

works as expected, while

```
a.erase(q);
```

does not ever invoke the conversion-to-iterator operator, thus avoiding the associated computation. To allow this technique, some sections of TR1 along the line "return value is an iterator…" should be changed to "return value is an unspecified object implicitly convertible to an iterator…" Although this trick is expected to work transparently, it can have some collateral effects when the expression `a.erase(q)` is used inside generic code.