Audience: LEWG, SG14
Document number: P0447R2
Date: 2017-02-11
Project: Introduction of std::colony to the standard library
Reply-to: Patrice Roy <patricer@gmail.com>, Guy Davidson <guy.davidson@hatcat.com>, Matthew Bentley <mattreecebentley@gmail.com>

Introduction of std::colony to the standard library

I. Introduction

The following is an initial draft based on the positive response to functionality and performance of the reference implementation. This has been receiving feedback for the past 1.5 years from Boost, SG14 and individual developers. On the basis of support from SG14 we are looking for feedback on the existing proposal. The word "link" is used within this text to refer to any method of referencing elements within a container, ie. indexes/pointers/iterators/id's.

Colony is a higher-performance container for situations where the ratio of insertions/erasures to iterations is relatively high. It is unordered but sortable, allows for both fast iteration and direct pointer linkage, and does not invalidate pointers/iterators to non-erased elements during insertion or erasure. It also frees or reuses unused memory on the fly, and is it's own abstract data type; similar to a "bag", "bucket-array" or "unordered multiset" but without key values. On the basis of a reference implementation (plf::colony) the following attributes have been established:

Colonies have better overall performance than any standard library container or container modification when:

a. Insertion order is unimportant

b. Insertions and erasures to the container occur frequently in performance-critical code, ***and***

c. Links to non-erased container elements may not be invalidated by insertion or erasure.

The performance characteristics compared to other containers, for the most-commonly-used operations in the above circumstance are:

- *Singular (non-fill, unordered) insertion*: better than any std:: library container except some implementations of deque.
- *Erasure (from random location)*: better or equal to any std:: library container.
- *Iteration*: better than any std:: library container capable of preserving pointer validity post-erasure (eg. map, multiset, list). Where pointer validity is unimportant, better than any std:: library container except deque and vector.

There are some vector/deque modifications/workarounds which can outperform colony for iteration while maintaining link validity, but which have slower insertion and erasure speeds, and also typically have a cost to usability or memory requirements. Under scenarios involving high levels of modification colony will outperform these as well. These results, along with performance comparisons to the unmodified std:: containers, are explored in detail in the Appendix B Benchmarks.

II. Motivation and Scope

Sometimes there are situations where data is heavily interlinked, iterated over frequently, and changing often. An example might be a central 'entity' class in a video game engine. These are 'has a'-style objects rather than 'is a'-style objects, which reference other shared resources like sprites, sounds and suchforth. These resources are typically located in separate containers/arrays. The entities are, in turn, referenced by other structures like quadtrees/octrees, level structures and the like. The entities could be erased at any time (for example, a wall gets destroyed and no longer requires processing) and new entities may be inserted (for example, a new enemy is created). All the while, inter-linkages between entities, resources and superstructures such as levels and quadtrees, are required to stay valid in order for the game to run. The order of the entities and resources themselves within the containers is, in the context of a game, typically unimportant.

What is needed is a container (or modification of a container) such that these insertions and erasures may occur without invalidating all the interlinkages between elements within containers. Unfortunately the container with the best iteration performance in the standard library, vector[1], also loses pointer validity to elements within it upon insertion, and also pointer/index validity upon erasure. This tends to lead to sophisticated, and often restrictive, workarounds when developers attempt to utilize vector or similar containers under the above circumstances.

While these situations above are common across multiple domains, for the benefit of those unfamiliar with any specific scenarios, I will present some more specific requirements with regards to game engines. When working on game engines we are predominantly dealing with situations where:

a. Elements within data collections refer to elements within other data collections (through a variety of methods - indices, pointers, etc). These references must stay valid throughout the course of the game/level. Any container which causes pointer or index invalidation can create difficulties or necessitate workarounds.

b. Order is unimportant for the most part. The majority of data is simply iterated over, transformed, referred to and utilized with no regard to order.

c. Erasing or otherwise "deactivating" objects occurs frequently in performance-critical code. For this reason methods of erasure which create strong performance penalties are avoided.

d. Inserting new objects in performance-critical code (during gameplay) is also common - for example, a tree drops leaves every so often, or a player spawns in a multiplayer game.

e. It is not always clear in advance how many elements there will be in a container at the beginning of development, or even at the beginning of a level during play. Genericized game engines in particular have to adapt to considerably different user requirements and scopes. For this reason extensible containers which can expand and contract in realtime are often necessary.

f. For modern performance reasons, memory storage which is more-or-less contiguous is preferred.

g. Memory waste is avoided.

Unfortunately, std::vector in it's default state, does not meet these requirements due to:

1. Poor (non-fill) singular insertion performance (regardless of insertion position) due to the need for reallocation upon reaching capacity
2. Insert invalidates pointers/iterators to all elements
3. Erase invalidates pointers/iterators/indexes to all elements afer the erased element

Hence game developers tend to either simply develop custom solutions for each scenario or implement a workaround for vector. Workarounds vary, but the most common are most likely:

1. Using a boolean flag or similar to indicate the inactivity of an object (as opposed to actually erasing from the vector). Elements flagged as inactive are skipped during iteration.

   Advantages: Fast "deactivation".
   Disadvantages: Slow to iterate due to branching.

2. Using a vector of data and a secondary vector of indexes. When erasing, the erasure occurs only in the vector of indexes, not the vector of data. When iterating it iterates over the vector of indexes and accesses the data from the vector of data via the remaining indexes.

   Advantages: Fast iteration.
   Disadvantages: Erasure still incurs some reallocation cost which can increase jitter.

3. Combining a swap-and-pop approach to erasure with some form of dereferenced lookup system to enable contiguous element iteration (sometimes called a 'packed array': http://bitsquid.blogspot.ca/2011/09/managing-decoupling-part-4-id-lookup.html). When iterating over the data we simply iterate through the vector of elements. When erasing, the back element of the vector is swapped with the element being erased, then the new back element is popped. A typical dereferencing system follows:
   To maintain valid external references to elements (which may be swapped from the back to an erased element location), a vector of indexes (V1) is maintained and updated when erasures or insertions occur. External objects referring to elements within the container store a pointer (or index) to the entry in V1 corresponding to the element in question. In addition the data vector (Vd) has a second vector of indexes (V2). This is used by the structure to update the entry in V1 upon movement of an element. Lastly, a free-list of erased entries in V1 is maintained (entries in V1 are erased when their corresponding elements in Vd are erased) and these locations are reused upon subsequent insertions.

   Advantages: Iteration is at standard vector speed.
   Disadvantages: Erase could be slow if objects are large and/or non-trivially copyable, thereby making swap cost large. All references to elements incur additional costs due to the dereferencing system.

Each of these techniques has the disadvantage of slow singular insertions, and the first two will also continually expand memory usage when erasing and inserting over periods of time. Colony is an attempt to bring a more generic solution to these situations. It has the advantage of good iteration speed, a similar erasure speed to the boolean technique described above, and an insertion speed which is much faster than a vector's. It also never causes pointer invalidation to non-erased elements during erasure or insertion and the memory locations of erased elements are either reused during subsequent insertions, or released on-the-fly when the element's memory block becomes empty.

III. Impact On the Standard

This is a pure library addition, no changes necessary to the standard asides from the introduction of the colony container.
A reference implementation of colony is available for download and use: http://www.plflib.org/colony.htm

IV. Design Decisions

The key technical features of this container are:

- Unordered non-associative data
- Never invalidates pointers/iterators to non-erased elements
- Reuses or frees memory from erased elements (memory is freed if a memory block becomes empty)
- In a high-modification context ie. where the ratio of insertion/erasure to iteration is high, it should be faster than any std:: library container or container workaround

The abstract requirements needed to support these features are:

- It must use multiple memory-blocks (this prevents element reallocation on insertion and associated pointer invalidation)
- Memory blocks must be freed once empty (otherwise O(1) ++ traversal is impossible)
- Memory blocks must be removable with low performance cost and without invalidating pointers to elements in other memory blocks
- There must be a mechanism for reusing erased element memory locations upon subsequent insertions
- Erased elements must also be recorded in a skipfield and skipped over during iteration (this prevents the necessity for element reallocation upon erasure)
- Skipfield design must allow for O(1) ++ and -- iteration

To summarise the above we can say that there are three aspects to a colony which define any implementation:

1. A multiple-memory-block based allocation pattern which allows for the fast removal of memory blocks when they become empty of elements without causing element pointer invalidation in other memory blocks.
2. A skipfield to enable the skipping over of erased elements during iteration.
3. A mechanism for reusing erased element locations upon subsequent insertions.

Obviously these three things can be achieved in a variety of different ways. We'll examine how the reference implementation achieves them now:

Memory blocks - chained group allocation pattern

This is essentially a doubly-linked intrusive list of nodes (groups) containing (a) memory blocks, (b) memory block metadata and (c) skipfields. The memory blocks themselves have a growth factor of 2, which reduces the number of memory allocations necessary for the container. The metadata in question includes information necessary for an iterator to iterate over colony elements, such as the last insertion point within the memory block, and other information useful to specific functions, such as the total number of non-erased elements in the node.

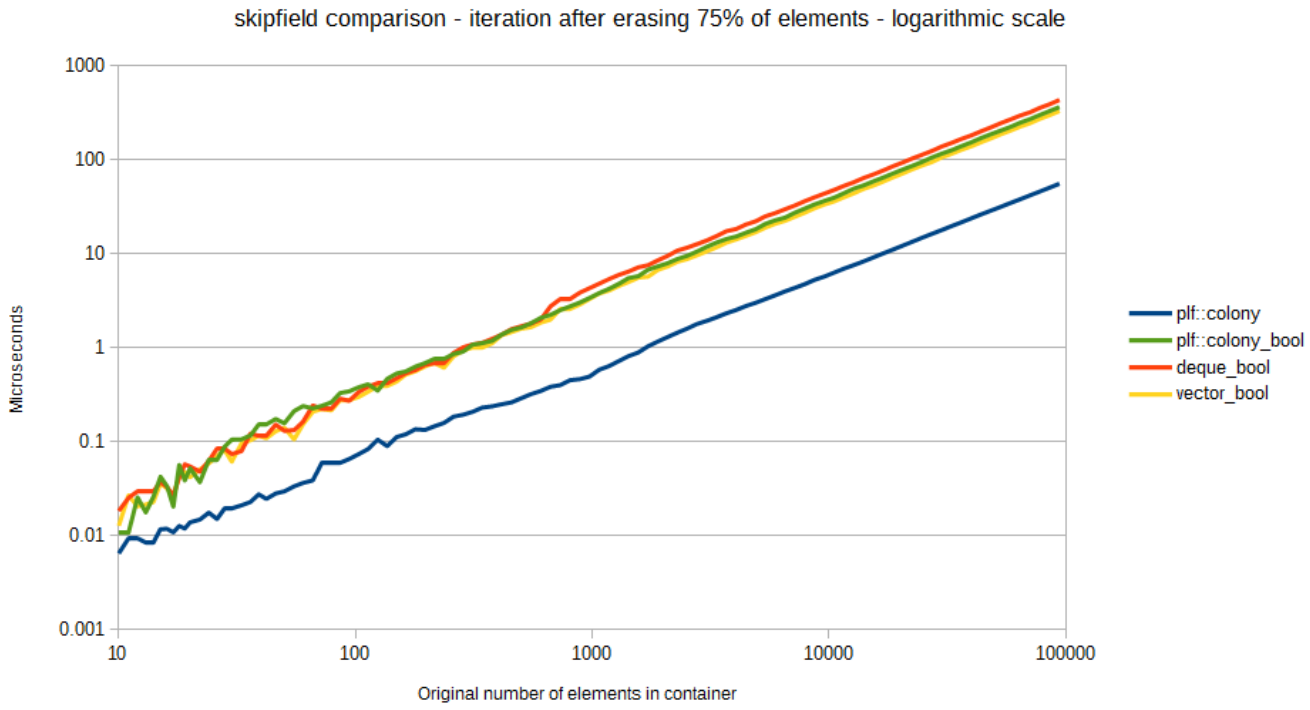
This approach keeps the operation of freeing empty memory blocks from the colony structure at O(1) time complexity; compared to, say, a vector of memory blocks. Further information is available here:
http://www.plflib.org/chained_group_allocation_pattern.htm

Alternative approaches of using a vector of pointers to dynamically-allocated groups, or keeping metadata separate, are possible, however this may cause jitter due to the additional overhead when non-back groups are removed.

Skipfield - jump-counting skipfield pattern

This numeric pattern allows for O(1) time complexity iterator operations and fast iteration compared to alternative skipfields. It stores and modifies (during insertion and erasure) numbers which correspond to the number of elements in runs of consecutive erased elements, and during iteration adds these numbers to the current iterator location. This avoids the looping branching code necessary for iteration with a boolean skipfield (as an aside, a colony could not actually use a boolean skipfield because ++ iteration becomes O(random), which violates the C++ iterator specifications). The full reference paper for the jump-counting skipfield pattern is available here: http://www.plflib.org/the_jump_counting_skipfield_pattern.pdf),

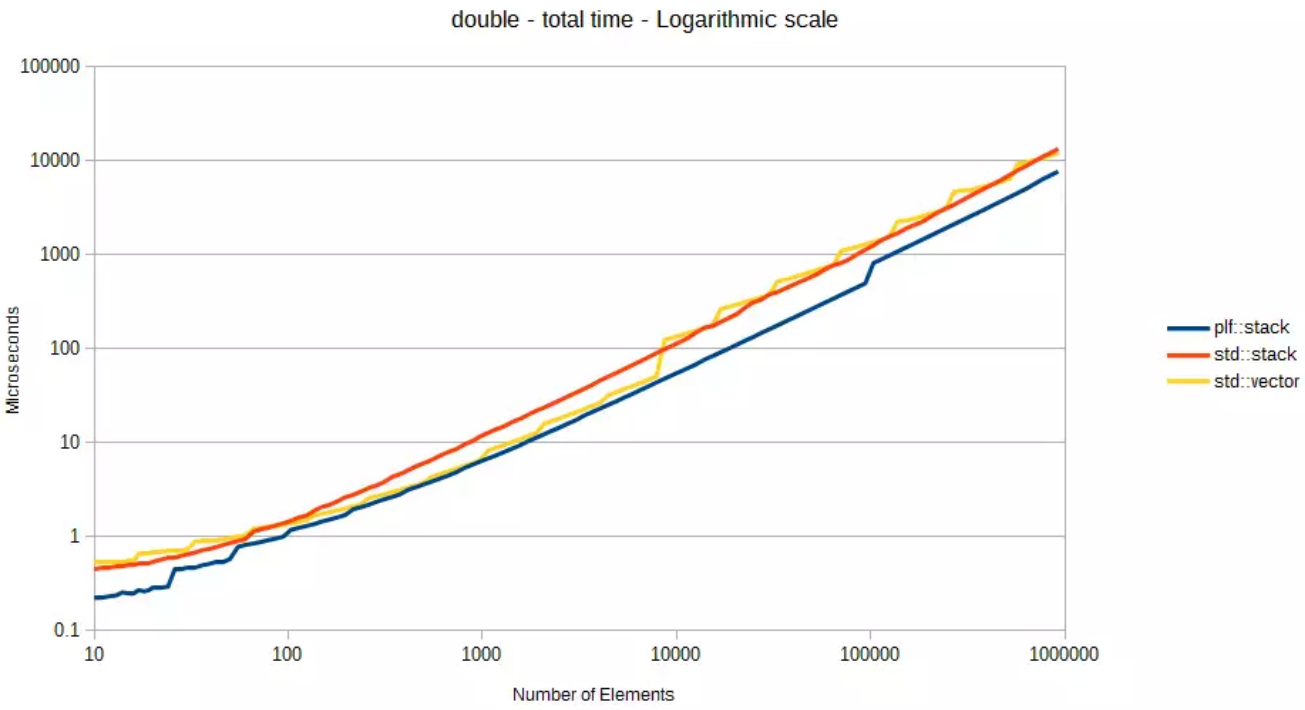
Iterative performance after having previously erased 75% of all elements in the containers at random, for a std::vector, std::deque and a colony with boolean skipfields, versus a colony with a jump-counting skipfield (GCC 5.1 x64) - storing a small struct type:



Memory re-use mechanism - reduced_stack

This is a stripped-down custom internal stack class based on plf::stack (http://www.plflib.org/stack.htm), which outperforms all other std:: containers in a stack context, for all datatypes and across compilers. plf::stack uses the same chained-group allocation pattern as plf::colony. The stack stores erased element memory locations and these are popped during subsequent colony insertions. If a memory block becomes empty and is subsequently removed, the stack is processed internally and any memory locations from within that block are removed.

Time to push all elements then read-and-pop all elements, plf::stack vs std::stack (std::deque) and std::vector (GCC 5.1 x64) - storing type double:



An alternative approach, using a "free list" (explanation of this concept: http://bitsquid.blogspot.ca/2011/09/managing-decoupling-part-4-id-lookup.html) to record the erased element memory locations, has been explored and the following issues identified:

1. A colony element could be smaller in size than a pointer and thus a union with such would dramatically increase the amount of wasted space in circumstances with low numbers of erasures - moreso if the pointer type supplied by the allocator is non-trivial.
2. Because a free list will jump between colony memory blocks continuously, consolidating a free list when a memory block becomes empty would result in an operation filled with cache misses. In the context of jitter-sensitive work this would become unacceptable.

V. Technical Specifications

Colony meets the requirements of the C++ Container, AllocatorAwareContainer, and ReversibleContainer concepts.

For the most part the syntax and semantics of colony functions are very similar to current existing std:: c++ libraries. Formal description is as follows:

```
template <class T, class Allocator = std::allocator<T>, typename Skipfield_Type = unsigned short> class colony
```

`T` - the element type. In general T must meet the requirements of Erasable, CopyAssignable and CopyConstructible. However, if emplace is utilized to insert elements into the colony, and no functions which involve copying or moving are utilized, T is only required to meet the requirements of Erasable. If move-insert is utilized instead of emplace, T must also meet the requirements of MoveConstructible .

`Allocator` - an allocator that is used to acquire memory to store the elements. The type must meet the requirements of Allocator. The behavior is undefined if `Allocator::value_type` is not the same as T.

`Skipfield_Type` - an unsigned integer type. This type is used to form the skipfield, and the maximum size of element memory blocks is constrained by the type's bit-depth (due to the nature of a jump-counting skipfield). As an example, `unsigned short` on most platforms is 16-bit which thereby constrains the size of individual memory blocks to a maximum of 65535 elements. `unsigned short` has been found to be the optimal type for performance based on benchmarking. However there could be memory-constrained situations where element block allocations of more than 255 elements at a time would not be desirable. In these situations, `unsigned char` could be used as the skipfield type instead, resulting in additional memory saving in allocation of the skipfield. It is unlikely for there to be any circumstances which benefit from a skipfield bit-depth greater than `unsigned short`. Behaviour is undefined if `Skipfield_Type` is not an unsigned integer type.

Basic example of usage

```
#include <iostream>
#include "plf_colony.h"

int main(int argc, char **argv)
{
  plf::colony<int> i_colony;

  // Insert 100 ints:
  for (int i = 0; i != 100; ++i)
  {
    i_colony.insert(i);
  }

  // Erase half of them:
  for (plf::colony<int>::iterator it = i_colony.begin(); it != i_colony.end(); ++it)
  {
    it = i_colony.erase(it);
  }

  // Total the remaining ints:
  int total = 0;

  for (plf::colony<int>::iterator it = i_colony.begin(); it != i_colony.end(); ++it)
  {
    total += *it;
  }

  std::cout << "Total: " << total << std::endl;
  std::cin.get();
  return 0;
}
```

Example demonstrating pointer stability

```
#include <iostream>
#include "plf_colony.h"

int main(int argc, char **argv)
{
  plf::colony<int> i_colony;
  plf::colony<int>::iterator it;
  plf::colony<int *> p_colony;
  plf::colony<int *>::iterator p_it;

  // Insert 100 ints to i_colony and pointers to those ints to p_colony:
  for (int i = 0; i != 100; ++i)
  {
    it = i_colony.insert(i);
    p_colony.insert(&(*it));
  }

  // Erase half of the ints:
  for (it = i_colony.begin(); it != i_colony.end(); ++it)
  {
    it = i_colony.erase(it);
  }

  // Erase half of the int pointers:
  for (p_it = p_colony.begin(); p_it != p_colony.end(); ++p_it)
  {
    p_it = p_colony.erase(p_it);
  }

  // Total the remaining ints via the pointer colony:
  int total = 0;

  for (p_it = p_colony.begin(); p_it != p_colony.end(); ++p_it)
  {
    total += *(*p_it);
  }
```

```
  std::cout << "Total: " << total << std::endl;

  if (total == 2500)
  {
    std::cout << "Pointers still valid!" << std::endl;
  }

  std::cin.get();
  return 0;
}
```

Iterator Invalidation

All read-only operations, swap, std::swap	Never
clear, reinitialize, operator =	Always
reserve, shrink_to_fit	Only if capacity is changed
change_group_sizes, change_minimum_group_size, change_maximum_group_size	Only if supplied minimum group size is larger than smallest group in colony, or supplied maximum group size is smaller than largest group in colony.
erase	Only for the erased element
insert, emplace	If an iterator is == end() it may be invalidated by a subsequent insert/emplace. Otherwise it will not be invalidated.

Member types

Member type	Definition
value_type	T
allocator_type	Allocator
size_type	Allocator::size_type (pre-c++11) std::allocator_traits<Allocator>::size_type (post-c++11)
difference_type	Allocator::difference_type (pre-c++11) std::allocator_traits<Allocator>::difference_type (post-c++11)
reference	Allocator::reference (pre-c++11) value_type & (post-c++11)
const_reference	Allocator::const_reference (pre-c++11) const value_type & (post-c++11)
pointer	Allocator::pointer (pre-c++11) value_type & (post-c++11)
const_pointer	Allocator::const_pointer (pre-c++11) std::allocator_traits<Allocator>::const_pointer (post-c++11)
iterator	BidirectionalIterator
const_iterator	Constant BidirectionalIterator
reverse_iterator	BidirectionalIterator
const_reverse_iterator	Constant BidirectionalIterator

Colony iterators may not be random access due to the use of a skipfield. This constrains +, -, += or -= operators into being non-O(1). But member overloads for the standard library functions advance(), next(), prev() and distance() are available in the reference implementation, and are significantly faster than O(n) in the majority of scenarios.

A full list of the current reference implementation's constructors and member functions can be found in Appendix A.

VI. Acknowledgements

Matt would like to thank: Glen Fernandes and Ion Gaztanaga for restructuring advice, Robert Ramey for documentation advice, various Boost and SG14 members for support, Baptiste Wicht for teaching me how to construct decent benchmarks, suggestions

and bug reports including Sean Middleditch, Patrice Roy and Guy Davidson, that jerk from Lionhead for annoying me enough to get me to get around to implementing the skipfield pattern, Jon Blow for initial advice and Mike Acton for some influence.

VII. Appendixes

Appendix A - Reference implementation member functions

Constructors

default	`explicit colony(const allocator_type &alloc = allocator_type())`
fill	`explicit colony(const size_type n, const unsigned short min_group_size = 8, const unsigned short max_group_size = std::numeric_limits<Skipfield_type>::max(), const allocator_type &alloc = allocator_type())` `explicit colony(const size_type n, const value_type &element, const unsigned short min_group_size = 8, const unsigned short max_group_size = std::numeric_limits<Skipfield_type>::max(), const allocator_type &alloc = allocator_type())`
range	`template<typename InputIterator> colony(const InputIterator &first, const InputIterator &last, const unsigned short min_group_size = 8, const unsigned short max_group_size = std::numeric_limits<Skipfield_type>::max(), const allocator_type &alloc = allocator_type())`
copy	`colony(const colony &source)`
move	`colony(colony &&source) noexcept` (C++11 and upwards)
initializer list	`colony(const std::initializer_list<value_type> &element_list, const unsigned short min_group_size = 8, const unsigned short max_group_size = std::numeric_limits<Skipfield_type>::max(), const allocator_type &alloc = allocator_type())`

Some constructor usage examples

- `colony<T> a_colony`

  Default constructor - default minimum group size is 8, default maximum group size is std::numeric_limits<Skipfield_type>::max() (typically 65535). You cannot set the group sizes from the constructor in this scenario, but you can call the change_group_sizes() member function after construction has occurred.
  Example: `plf::colony<int> int_colony;`

- `colony<T, the_allocator<T> > a_colony(const allocator_type &alloc = allocator_type())`

  Default constructor, but using a custom memory allocator eg. something other than std::allocator.
  Example: `plf::colony<int, tbb::allocator<int> > int_colony;`
  Example2:
  ```
  // Using an instance of an allocator as well as it's type
  tbb::allocator<int> alloc_instance;
  plf::colony<int, tbb::allocator<int> > int_colony(alloc_instance);
  ```

- `colony<T> colony(const size_type n, const unsigned short min_group_size = 8, const unsigned short max_group_size = std::numeric_limits<Skipfield_type>::max())`

  Fill constructor with value_type unspecified, so the value_type's default constructor is used. `n` specifies the number of elements to create upon construction. If `n` is larger than `min_group_size`, the size of the groups created will either be `n` and `max_group_size`, depending on which is smaller. `min_group_size` (ie. the smallest possible number of elements which can be stored in a colony group) can be defined, as can the `max_group_size`. Setting the group sizes can be a performance advantage if you know in advance roughly how many objects are likely to be stored in your colony long-term - or at least the rough scale of storage. If that case, using this can stop many small initial groups being allocated.
  Example: `plf::colony<int> int_colony(62);`

- `colony<T> colony(const std::initializer_list<value_type> &element_list, const unsigned short min_group_size = 8, const unsigned short max_group_size = std::numeric_limits<Skipfield_type>::max())`

Using an initialiser list to insert into the colony upon construction.
Example: `std::initializer_list<int> &el = {3, 5, 2, 1000};`
`plf::colony<int> int_colony(el, 64, 512);`

- `colony<T> a_colony(const colony &source)`

Copy all contents from source colony, removes any empty (erased) element locations in the process. Size of groups created is either the total size of the source colony, or the maximum group size of the source colony, whichever is the smaller.
Example: `plf::colony<int> int_colony_2(int_colony_1);`

- `colony<T> a_colony(colony &&source)`

Move all contents from source colony, does not remove any erased element locations or alter any of the source group sizes. Source colony is now void of contents and can be safely destructed.
Example: `plf::colony<int> int_colony_1(50, 5, 512, 512); // Create colony with min and max group sizes set at 512 elements. Fill with 50 instances of int = 5.`
`plf::colony<int> int_colony_2(std::move(int_colony_1)); // Move all data to int_colony_2. All of the above characteristics are now applied to int_colony2.`

Iterators

All iterators are bidirectional but also provide >, <, >= and <= for convenience (for example, for comparisons against end() when doing multiple increments within for loops) and within some functions (`distance()` uses these for example). Functions for iterator, reverse_iterator, const_iterator and const_reverse_iterator follow:

```
operator *
operator ->
operator ++
operator --
operator =
operator ==
operator !=
operator <
operator >
operator <=
operator >=
base() (reverse_iterator and const_reverse_iterator only)
```

All operators have O(1) amortised time-complexity. Originally there were +=, -=, + and - operators, however the time complexity of these varied from O(n) to O(1) depending on the underlying state of the colony, averaging in at O(log n). As such they were not includable in the iterator functions (as per C++ standards). These have been transplanted to colony's advance(), next(), prev() and distance() member functions. Greater-than/lesser-than operator usage indicates whether an iterator is higher/lower in position compared to another iterator in the same colony (ie. closer to the end/beginning of the colony).

Member functions

Insert

single element	`iterator insert (const value_type &val)`
fill	`iterator insert (const size_type n, const value_type &val)`
range	`template <class InputIterator> iterator insert (const InputIterator &first, const InputIterator &last)`
move	`iterator insert (value_type&& val)` (C++11 and upwards)
initializer list	`iterator insert (const std::initializer_list<value_type> &il)`

- `iterator insert(const value_type &element)`

Inserts the element supplied to the colony, using the object's copy-constructor. Will insert the element into a previously erased element slot if one exists, otherwise will insert to back of colony. Returns iterator to location of inserted element. Example:

```
plf::colony<unsigned int> i_colony;
i_colony.insert(23);
```

- `iterator insert (const size_type n, const value_type &val)`

Inserts `n` copies of `val` into the colony. Will insert the element into a previously erased element slot if one exists, otherwise will insert to back of colony. Returns iterator to location of first inserted element. Example:

```
plf::colony<unsigned int> i_colony;
i_colony.insert(10, 3);
```

- `template <class InputIterator> iterator insert (const InputIterator &first, const InputIterator &last)`

Inserts a series of `value_type` elements from an external source into a colony holding the same `value_type` (eg. int, float, a particular class, etcetera). Stops inserting once it reaches `last`. Example:

```
// Insert all contents of colony2 into colony1:
colony1.insert(colony2.begin(), colony2.end());
```

- `iterator insert(value_type &&element)` **C++11 and upwards**

Moves the element supplied to the colony, using the object's move-constructor. Will insert the element in a previously erased element slot if one exists, otherwise will insert to back of colony. Returns iterator to location of inserted element. Example:

```
std::string string1 = "Some text";

plf::colony<std::string> data_colony;
data_colony.insert(std::move(string1));
```

- `iterator insert (const std::initializer_list<value_type> &il)`

Moves the element supplied to the colony, using the object's move-constructor. Will insert the element in a previously erased element slot if one exists, otherwise will insert to back of colony. Returns iterator to location of inserted element. Example:

```
std::initializer_list<int> some_ints = {4, 3, 2, 5};

plf::colony<int> i_colony;
i_colony.insert(some_ints);
```

Erase

single element	`iterator erase(const iterator &it)`
range	`void erase(const iterator &first, const iterator &last)`

- `iterator erase(const iterator &it)`

Removes the element pointed to by the supplied iterator, from the colony. Returns an iterator pointing to the next non-erased element in the colony (or to end() if no more elements are available). This must return an iterator because if a colony group becomes entirely empty, it will be removed from the colony, invalidating the existing iterator. Attempting to erase a previously-erased element results in undefined behaviour (this is checked for via an assert in debug mode). Example:

```
plf::colony<unsigned int> data_colony(50);
plf::colony<unsigned int>::iterator an_iterator;
an_iterator = data_colony.insert(23);
an_iterator = data_colony.erase(an_iterator);
```

- `void erase(const iterator &first, const iterator &last)`

Erases all contents of a given colony from `first` to the element before the `last` iterator. Example:

```
plf::colony<int> iterator1 = colony1.begin();
colony1.advance(iterator1, 10);
plf::colony<int> iterator2 = colony1.begin();
colony1.advance(iterator2, 20);
colony1.erase(iterator1, iterator2);
```

Other functions

- `iterator emplace(Arguments ...parameters)` **C++11 and upwards**

  Constructs new element directly within colony. Will insert the element in a previously erased element slot if one exists, otherwise will insert to back of colony. Returns iterator to location of inserted element. "...parameters" are whatever parameters are required by the object's constructor. Example:

```
class simple_class
{
private:
int number;
public:
simple_class(int a_number): number (a_number) {};
};

plf::colony<simple_class> simple_classes;
simple_classes.emplace(45);
```

- `bool empty()`

  Returns a boolean indicating whether the colony is currently empty of elements.
  Example: `if (object_colony.empty()) return;`

- `size_type size()`

  Returns total number of elements currently stored in container.
  Example: `std::cout << i_colony.size() << std::endl;`

- `size_type max_size()`

  Returns the maximum number of elements that the allocator can store in the container. This is an approximation as it does attempt to measure the memory overhead of the container's internal memory structures. It is not possible to measure the latter because a copy operation may change the number of groups utilized for the same amount of elements, if the maximum or minimum group sizes are different in the source container.
  Example: `std::cout << i_colony.max_size() << std::endl;`

- `size_type capacity()`

  Returns total number of elements currently able to be stored in container without expansion.
  Example: `std::cout << i_colony.capacity() << std::endl;`

- `void shrink_to_fit()`

  Reduces container capacity to the amount necessary to store all currently stored elements. If the total number of elements is larger than the maximum group size, the resultant capacity will be equal to `((total_elements / max_group_size) + 1) * max_group_size` (rounding down at division). Invalidates all pointers, iterators and references to elements within the container.
  Example: `i_colony.shrink_to_fit();`

- `void reserve(unsigned short reserve_amount)`

  Preallocates memory space sufficient to store the number of elements indicated by `reserve_amount`. The maximum size for this number is limited to the maximum group size of the colony and will be truncated if necessary. The default maximum group size is 65535 on the majority of platforms.
  Example: `i_colony.reserve(15);`

- `void clear()`

  Empties the colony and removes all elements and groups.
  Example: `object_colony.clear();`

- `void change_group_sizes(const unsigned short min_group_size, const unsigned short max_group_size)`

  Changes the minimum and maximum internal group sizes, in terms of number of elements stored per group. If the colony is not empty and either min_group_size is larger than the smallest group in the colony, or max_group_size is smaller than the largest group in the colony, the colony will be internally copy-constructed into a new colony which uses the new group sizes, invalidating all pointers/iterators/references.
  Example: `object_colony.change_group_sizes(1000, 10000);`

- `void change_minimum_group_size(const unsigned short min_group_size)`

  Changes the minimum internal group size only, in terms of minimum number of elements stored per group. If the colony is not empty and min_group_size is larger than the smallest group in the colony, the colony will be internally copy-constructed into a new colony which uses the new minimum group size, invalidating all pointers/iterators/references.
  Example: `object_colony.change_minimum_group_size(100);`

- `void change_maximum_group_size(const unsigned short min_group_size)`

  Changes the maximum internal group size only, in terms of maximum number of elements stored per group. If the colony is not empty and either max_group_size is smaller than the largest group in the colony, the colony will be internally copy-constructed into a new colony which uses the new maximum group size, invalidating all pointers/iterators/references.
  Example: `object_colony.change_maximum_group_size(1000);`

- `void reinitialize(const unsigned short min_group_size, const unsigned short max_group_size)`

  Semantics of function are the same as "clear(); change_group_sizes(min_group_size, max_group_size);", but without the copy-construction code of the change_group_sizes() function - this means it can be used with element types which are non-copy-constructible, unlike change_group_sizes().
  Example: `object_colony.reinitialize(1000, 10000);`

- `void swap(colony &source)`

  Swaps the colony's contents with that of `source`.
  Example: `object_colony.swap(other_colony);`

- `friend void swap(colony &A, source &B)`

  External friend function, swaps the colony A's contents with that of colony B (assumes both stacks have same element type).
  Example: `swap(object_colony, other_colony);`

- `colony & operator = (const colony &source)`

  Copy the elements from another colony to this colony, clearing this colony of existing elements first.
  Example:
  ```
  // Manually swap data_colony1 and data_colony2 in C++03
  data_colony3 = data_colony1;
  data_colony1 = data_colony2;
  data_colony2 = data_colony3;
  ```

- `colony & operator = (const colony &&source)` **`C++11 only`**

  Move the elements from another colony to this colony, clearing this colony of existing elements first. Source colony becomes invalid but can be safely destructed without undefined behaviour.
  Example:
  ```
  // Manually swap data_colony1 and data_colony2 in C++11
  data_colony3 = std::move(data_colony1);
  data_colony1 = std::move(data_colony2);
  data_colony2 = std::move(data_colony3);
  ```

- `bool operator == (const colony &source)`

  Compare contents of another colony to this colony. Returns a boolean as to whether they are equal.
  Example: `if (object_colony == object_colony2) return;`

- `bool operator != (const colony &source)`

  Compare contents of another colony to this colony. Returns a boolean as to whether they are not equal.
  Example: `if (object_colony != object_colony2) return;`

- `iterator begin(), iterator end(), const_iterator cbegin(), const_iterator cend()`

  Return iterators pointing to, respectively, the first element of the colony and the element one-past the end of the colony (as per standard STL guidelines).

- `reverse_iterator rbegin(), reverse_iterator rend(), const_reverse_iterator cbegin(), const_reverse_iterator cend()`

  Return reverse iterators pointing to, respectively, the last element of the colony and the element one-before the first element of the colony (as per standard STL guidelines).

- `iterator get_iterator_from_pointer(const element_pointer_type the_pointer)` **`(slow)`**

  Getting a pointer from an iterator is simple - simply dereference it then grab the address ie. "`&(*the_iterator);`". Getting an iterator from a pointer is typically not so simple. This function enables the user to do exactly that. This is expected to be useful in the use-case where external containers are storing pointers to colony elements instead of iterators (as iterators for colonies have 3 times the size of an element pointer) and the program wants to erase the element being pointed to or possibly change the element being pointed to. Converting a pointer to an iterator using this method and then erasing, is about 20% slower on average than erasing when you already have the iterator. This is less dramatic than it sounds, as it is still faster than all std:: container erasure times which it is roughly equal to. However this is generally a slower, lookup-based operation. If the lookup doesn't find a non-erased element based on that pointer, it returns `end()`. Otherwise it returns an iterator pointing to the element in question. Example:

```
plf::colony<a_struct> data_colony;
plf::colony<a_struct>::iterator an_iterator;
a_struct struct_instance;
an_iterator = data_colony.insert(struct_instance);
a_struct *struct_pointer = &(*an_iterator);
iterator another_iterator = data_colony.get_iterator_from_pointer(struct_pointer);
if (an_iterator == another_iterator) std::cout << "Iterator is correct" << std::endl;
```

- `size_type get_index_from_iterator(const iterator/const_iterator &the_iterator)` **(slow)**

  While colony is a container with unordered insertion (and is therefore unordered), it still has a (transitory) order which changes upon any erasure or insertion. *Temporary* index numbers are therefore obtainable. These can be useful, for example, when creating a save file in a computer game, where certain elements in a container may need to be re-linked to other elements in other container upon reloading the save file. Example:

```
plf::colony<a_struct> data_colony;
plf::colony<a_struct>::iterator an_iterator;
a_struct struct_instance;
data_colony.insert(struct_instance);
data_colony.insert(struct_instance);
an_iterator = data_colony.insert(struct_instance);
unsigned int index = data_colony.get_index_from_iterator(an_iterator);
if (index == 2) std::cout << "Index is correct" << std::endl;
```

- `size_type get_index_from_reverse_iterator(const reverse_iterator/const_reverse_iterator &the_iterator)` **(slow)**

  The same as get_index_from_iterator, but for reverse_iterators and const_reverse_iterators. Index is from front of colony (same as iterator), not back of colony. Example:

```
plf::colony<a_struct> data_colony;
plf::colony<a_struct>::reverse_iterator r_iterator;
a_struct struct_instance;
data_colony.insert(struct_instance);
data_colony.insert(struct_instance);
r_iterator = data_colony.rend();
unsigned int index = data_colony.get_index_from_reverse_iterator(r_iterator);
if (index == 1) std::cout << "Index is correct" << std::endl;
```

- `iterator get_iterator_from_index(const size_type index)` **(slow)**

  As described above, there may be situations where obtaining iterators to specific elements based on an index can be useful, for example, when reloading save files. This function is basically a shorthand to avoid typing `"iterator it = colony.begin(); colony.advance(it, 50);"`. Example:

```
plf::colony<a_struct> data_colony;
plf::colony<a_struct>::iterator an_iterator;
a_struct struct_instance;
data_colony.insert(struct_instance);
data_colony.insert(struct_instance);
iterator an_iterator = data_colony.insert(struct_instance);
iterator another_iterator = data_colony.get_iterator_from_index(2);
if (an_iterator == another_iterator) std::cout << "Iterator is correct" << std::endl;
```

- `template <iterator_type> void advance(iterator_type iterator, distance_type distance)`

  Increments/decrements the iterator supplied by the positive or negative amount indicated by *distance*. Speed of incrementation will almost always be faster than using the ++ operator on the iterator for increments greater than 1. In some cases it may approximate O(1). The iterator_type can be an iterator, const_iterator, reverse_iterator or const_reverse_iterator.
  Example: `colony<int>::iterator it = i_colony.begin();`
  `i_colony.advance(it, 20);`

- `template <iterator_type> iterator_type next(const iterator_type &iterator, distance_type distance)`

  Creates a copy of the iterator supplied, then increments/decrements this iterator by the positive or negative amount indicated by *distance*.
  Example: `colony<int>::iterator it = i_colony.next(i_colony.begin(), 20);`

- `template <iterator_type> iterator_type prev(const iterator_type &iterator, distance_type distance)`

  Creates a copy of the iterator supplied, then decrements/increments this iterator by the positive or negative amount indicated by *distance*.
  Example: `colony<int>::iterator it2 = i_colony.prev(i_colony.end(), 20);`

- `template <iterator_type> difference_type distance(const iterator_type &first, const iterator_type &last)`

Measures the distance between two iterators, returning the result, which will be negative if the second iterator supplied is before the first iterator supplied in terms of it's location in the colony.
Example:
```
colony<int>::iterator it = i_colony.next(i_colony.begin(), 20);
colony<int>::iterator it2 = i_colony.prev(i_colony.end(), 20);
std::cout "Distance: " i_colony.distance(it, it2) std::endl;
```

Appendix B - plf::colony benchmarks

Last updated 8-2-2017 v3.87

Test machine setup

The test setup is an Intel E8500, 8GB ram, running GCC 5.1 x64 as compiler. OS is a stripped-back Windows 7 x64 SP1 installation with the most services, background tasks (including explorer) and all networking disabled. Build settings are "-O2 -march=native -std=c++11 -fomit-frame-pointer". Results under MSVC 2015 update 3, on an Intel Xeon E3-1241 (Haswell core) can be viewed here: http://www.plflib.org/colony_benchmark_msvc_results.htm. There is no commentary for the MSVC results.

Source code for the benchmarks can be found in the reference implementation's downloads section:
http://www.plflib.org/colony.htm#download

General test design

Tests are based on a sliding scale of number of runs vs number of elements, so a test with only 10 elements in a container may average 100000 runs to guarantee a more stable result average, whereas a test with 100000 elements may only average 10 runs. This tends to give adequate results without overly lengthening test times. I have not included results involving 'reserve()' functions as the differences to overall insertion performance were not adequate.

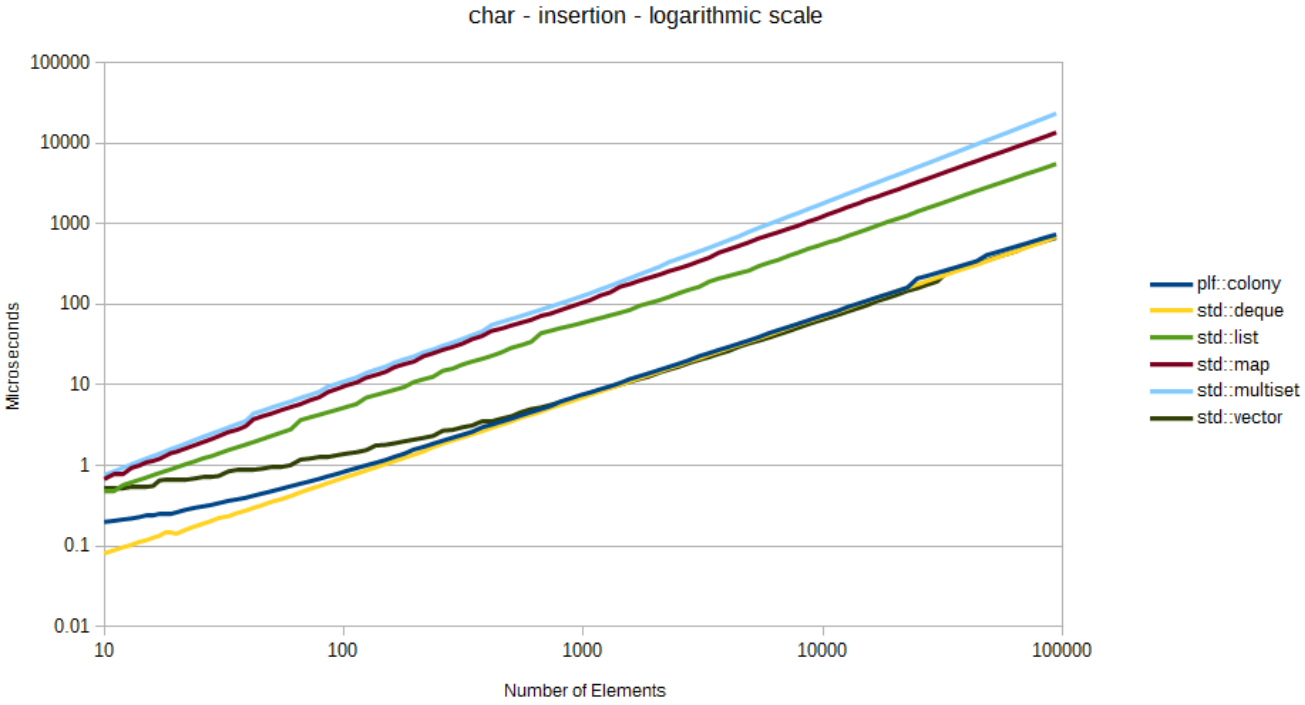
Insertion: is into empty containers for the raw and comparitive tests, entering single elements at a time. For the 'scenario' tests there is also ongoing insertion at random intervals. This matches the use case of colony, where insertion on-the-fly is expected to be one of the driving factors of usage. Insertion is always at the most performant location for the specific container, for example front for list, or back for vector.
Erasure: initially takes place in an iterative fashion for the raw tests, erasing elements at random as we iterate through the container. The exception to this is the tests involving a remove_if pattern (pointer_deque and indexed_vector) which have a secondary pass when using this pattern.
Iteration: is straightforward iteration from the start to end of any containers. Typically there are more runs for iteration than the other tests due to iteration being a much quicker procedure, so more runs deliver a more stable average.

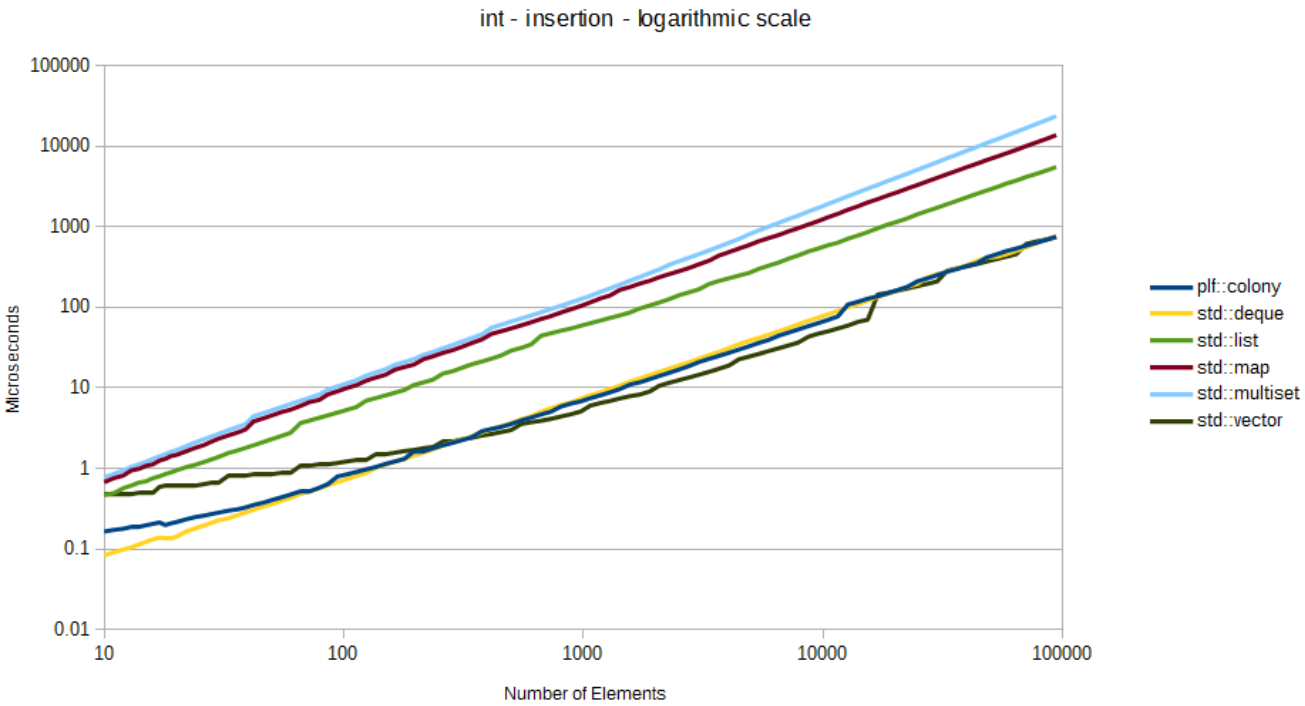
Raw performance tests

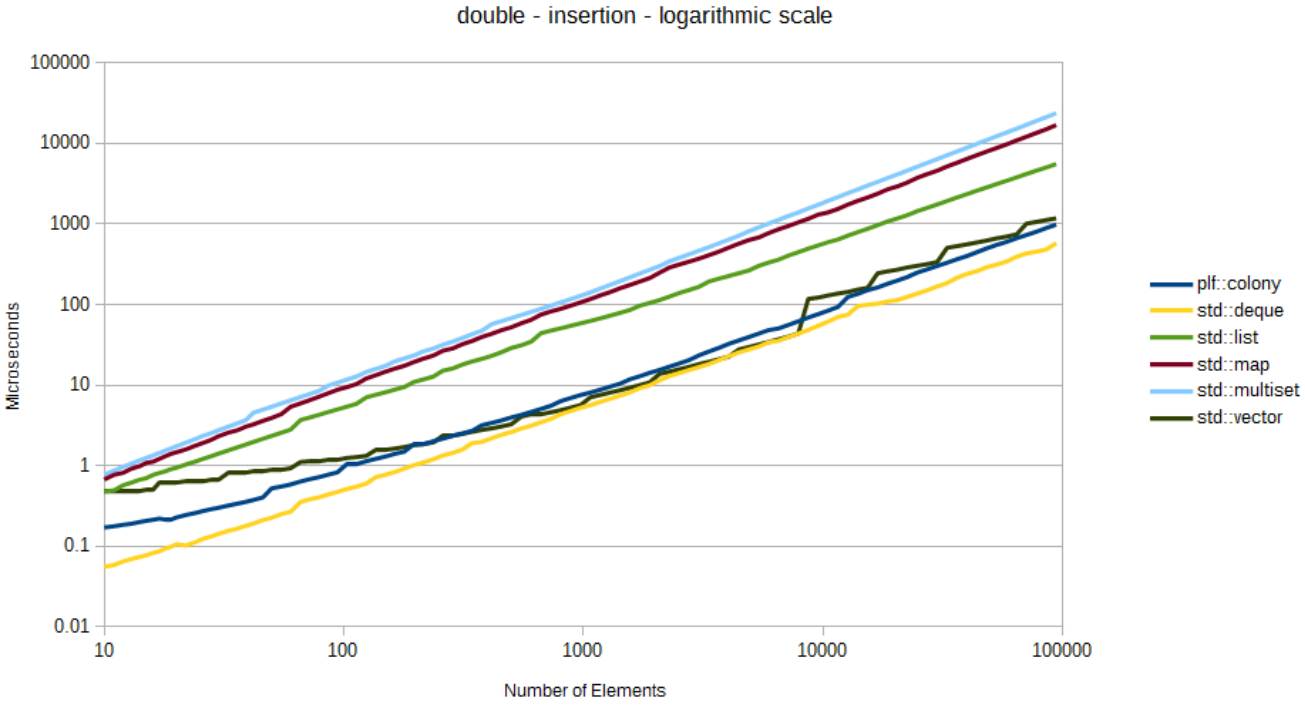
Before we begin measuring colony against containers or container modifications which do not invalidate links on erasure or insertion, we need to identify which containers are good candidates for comparison based on raw performance without regard to linkage invalidation. With that in mind the following tests compare colony against the main standard library containers. Tests are carried out on the following types: (a) a 8-bit type ie. char, (b) a 32-bit type ie. int, (c) a 64-bit type ie. double, (d) a small struct containing two pointers and four scalar types, and (e) a large struct containing 2 pointers, 4 scalar types, a large array of ints and a small array of chars.

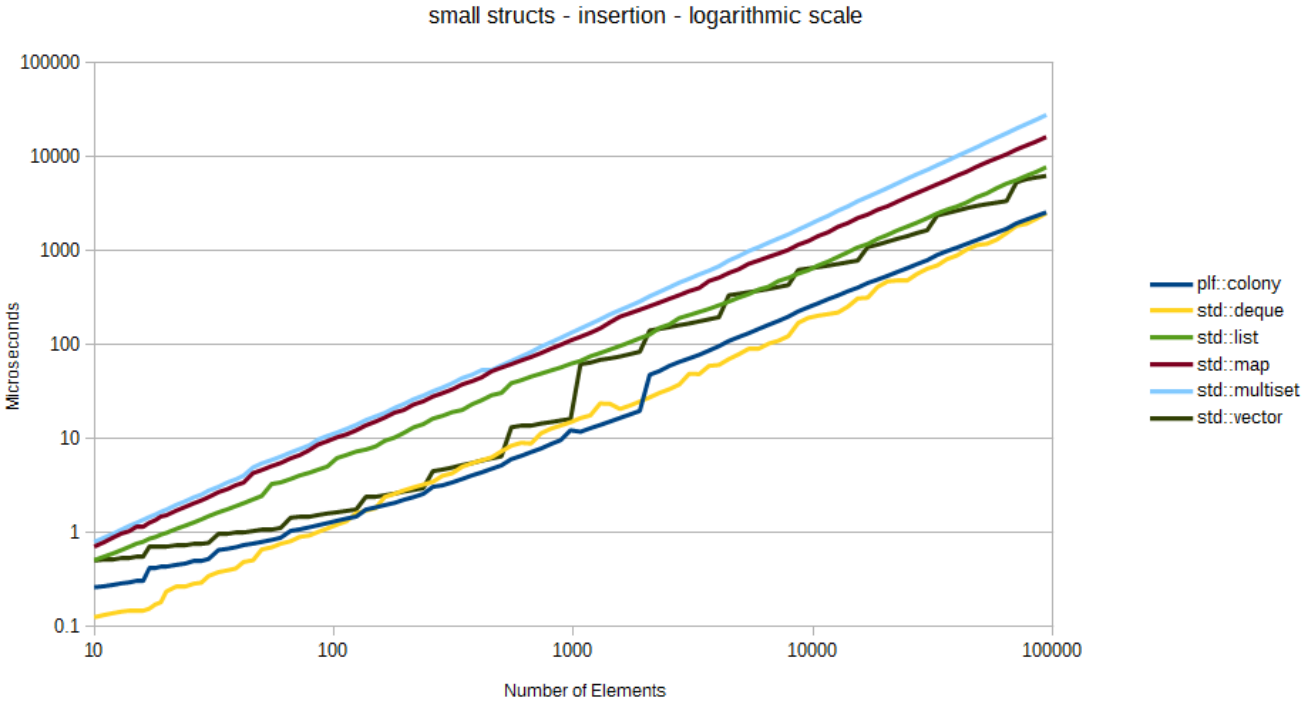
The first test measures time to insert N elements into a given container, the second measures the time taken to erase 25% of those same elements from the container, and the third test measures iteration performance after the erasure has taken place. Erasure tests avoid the remove_if pattern for the moment to show standard random-access erasure performance more clearly (this pattern is explored in the second test). Both linear and logarithmic views of each benchmark are provided in order to better show the performance of lower element amounts.

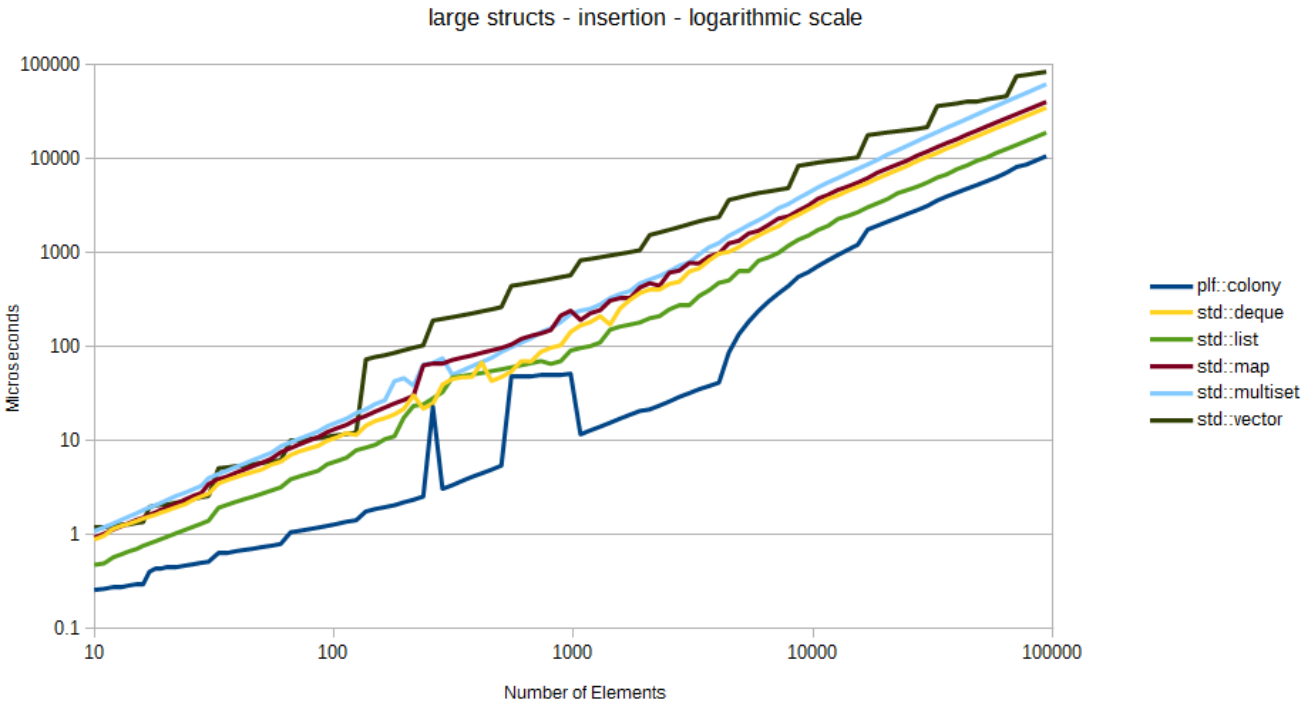
Insertion Performance






double - insertion - logarithmic scale
100000
10000
1000
100
10
1
0.1
0.01
Microseconds
10
100
1000
10000
100000
Number of Elements
plf::colony
std::deque
std::list
std::map
std::multiset
std::vector

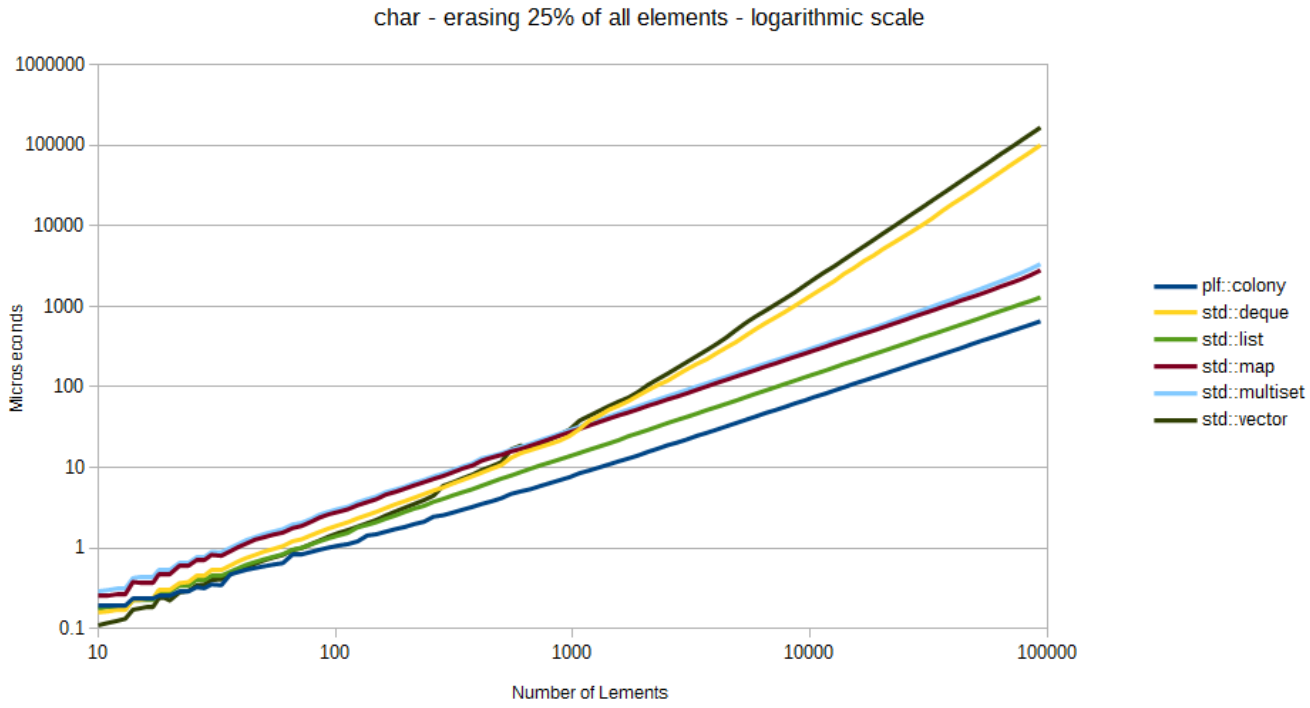

small structs - insertion - logarithmic scale
100000
10000
1000
100
10
1
0.1
Microseconds
10
100
1000
10000
100000
Number of Elements
plf::colony
std::deque
std::list
std::map
std::multiset
std::vector

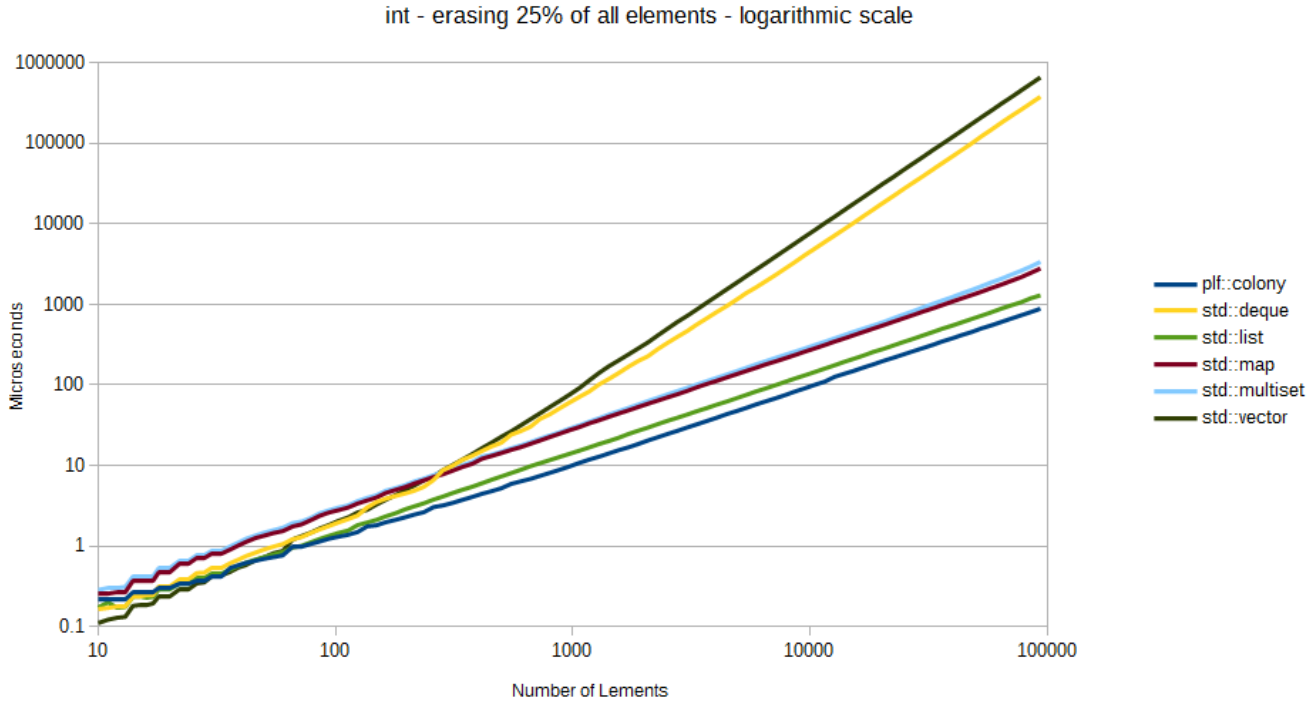


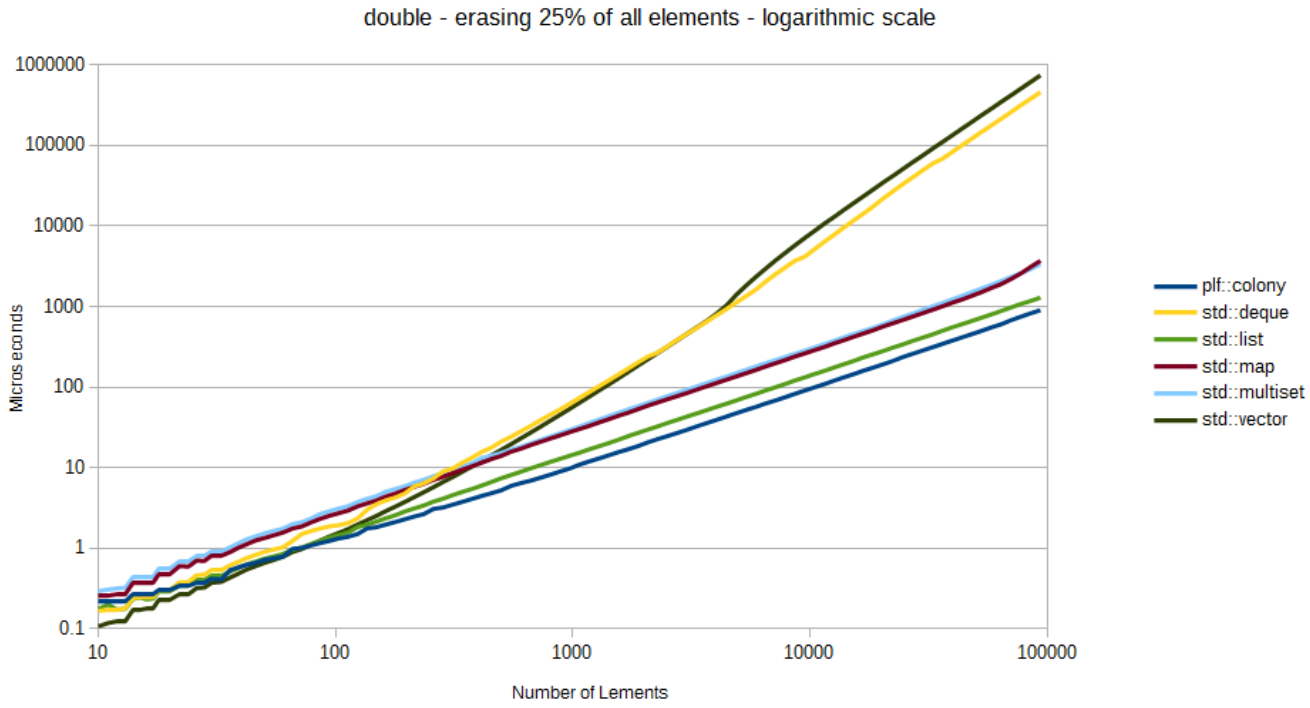
A predictable pattern for all but the large struct test is shown for insertion:
std::deque dominates insertion, with plf::colony equal after about 100 elements, but then for large structs it's performance completely eclipses std::deque. This is because libstdc++'s implementation of deque caps the memory block size at 512 bytes, whereas the large struct in question is ~506 bytes (depending on platform), meaning it, essentially, becomes a std::list but with additional overheads. Colony avoids this downfall due to it's memory allocation pattern of basing memory block sizes on fixed numbers of elements with a growth factor of 2, not fixed numbers of bytes. std::vector is nearly on a par with std::deque for very small element types with element numbers greater than a thousand, but becomes worse and worse the larger the size of the stored type is, and the fewer stored elements there are.
std::list, std::map and std::multiset all perform poorly by contrast, with the exception of large structs, where std::list comes in 2nd place to colony. Overall, plf::colony and std::deque dominate.

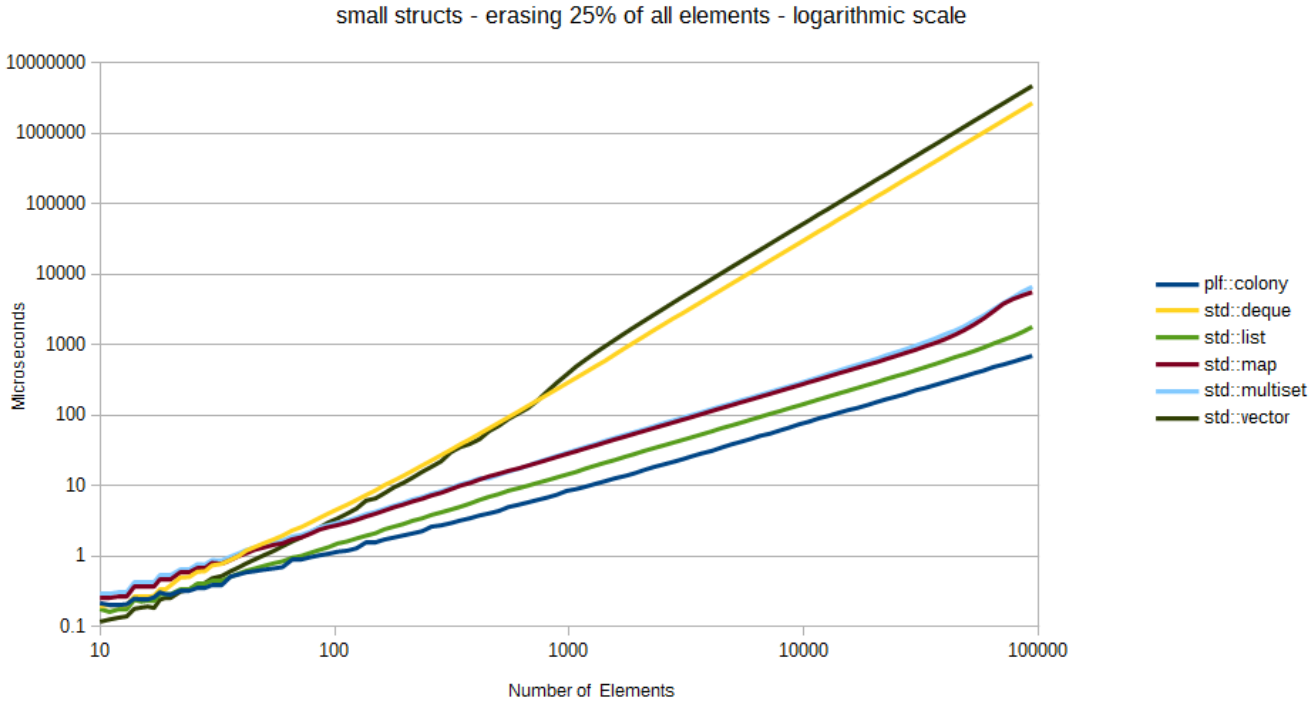
Erase Performance

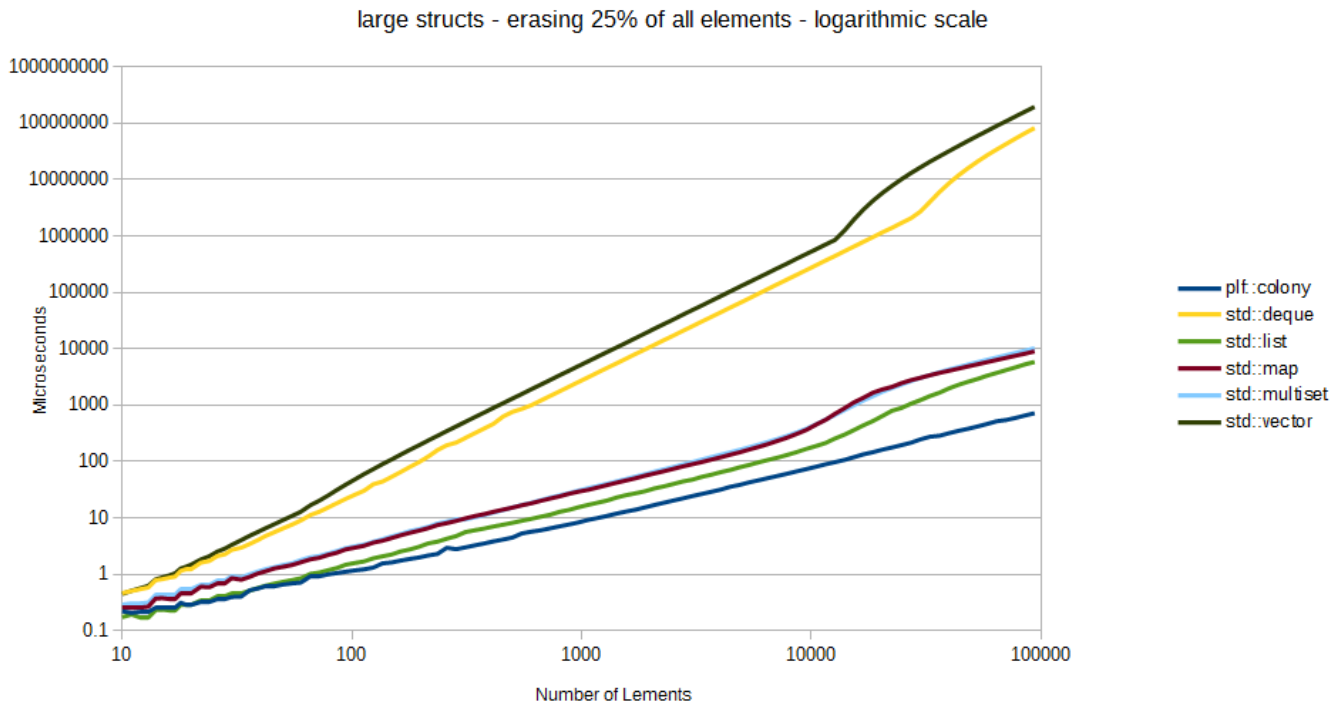
Here we forward-iterate over each container and erase 25% of all elements at random. If (due to the variability of random number generators) 25% of all elements have not been erased by the end of the container iteration, the test will reverse-iterate through the container and randomly erase the remaining necessary number of elements until that 25% has been reached.


char - erasing 25% of all elements - logarithmic scale
1000000
100000
10000
1000
100
10
1
0.1
Micros eco nds
10
100
1000
10000
100000
Number of Lements
plf::colony
std::deque
std::list
std::map
std::multiset
std::vector


int - erasing 25% of all elements - logarithmic scale
1000000
100000
10000
1000
100
10
1
0.1
Micros eco nds
10
100
1000
10000
100000
Number of Lements
plf::colony
std::deque
std::list
std::map
std::multiset
std::vector


double - erasing 25% of all elements - logarithmic scale
1000000
100000
10000
1000
100
10
1
0.1
Microseconds
10
100
1000
10000
100000
Number of Lements
plf::colony
std::deque
std::list
std::map
std::multiset
std::vector

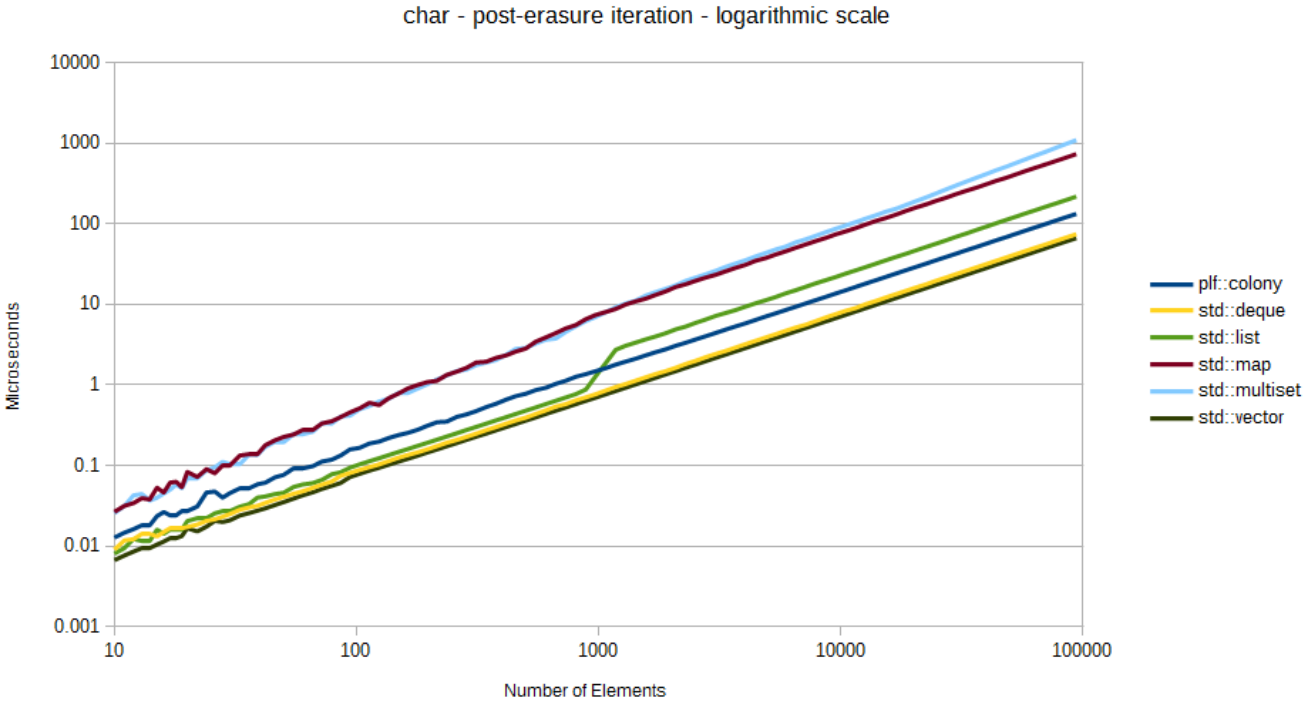

small structs - erasing 25% of all elements - logarithmic scale
10000000
1000000
100000
10000
1000
100
10
1
0.1
Microseconds
10
100
1000
10000
100000
Number of Elements
plf::colony
std::deque
std::list
std::map
std::multiset
std::vector

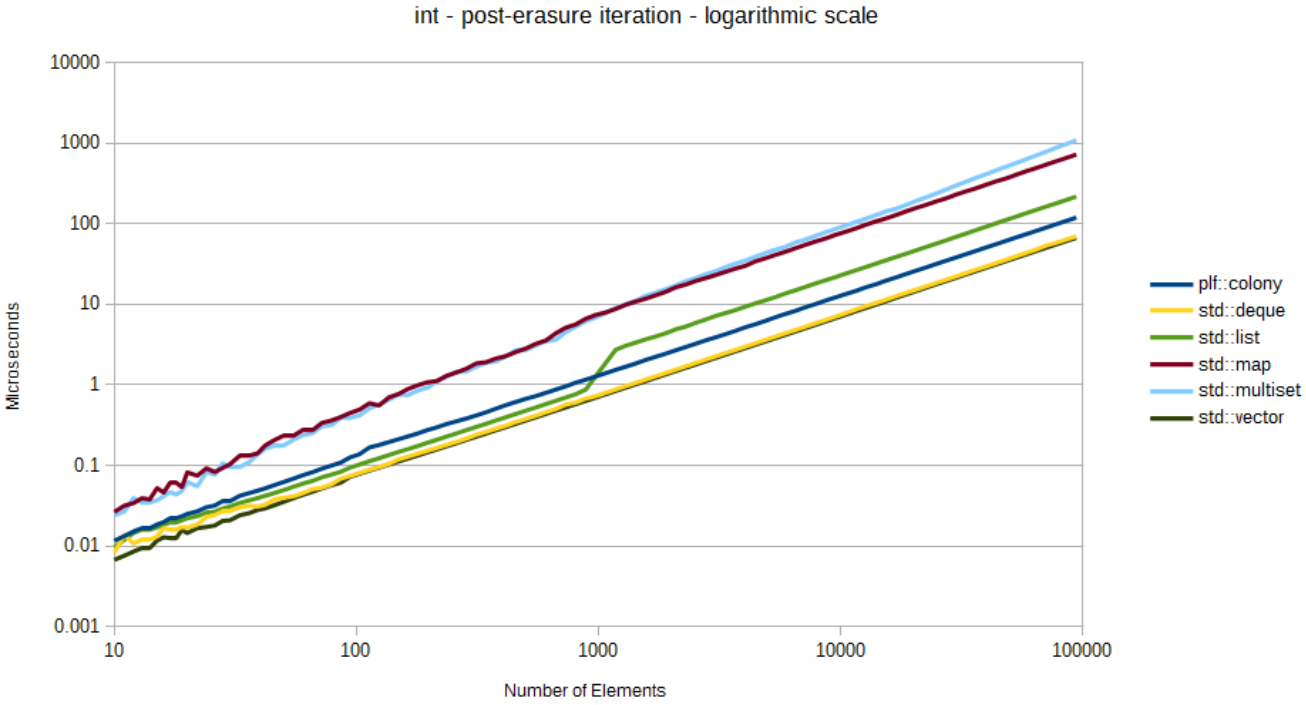


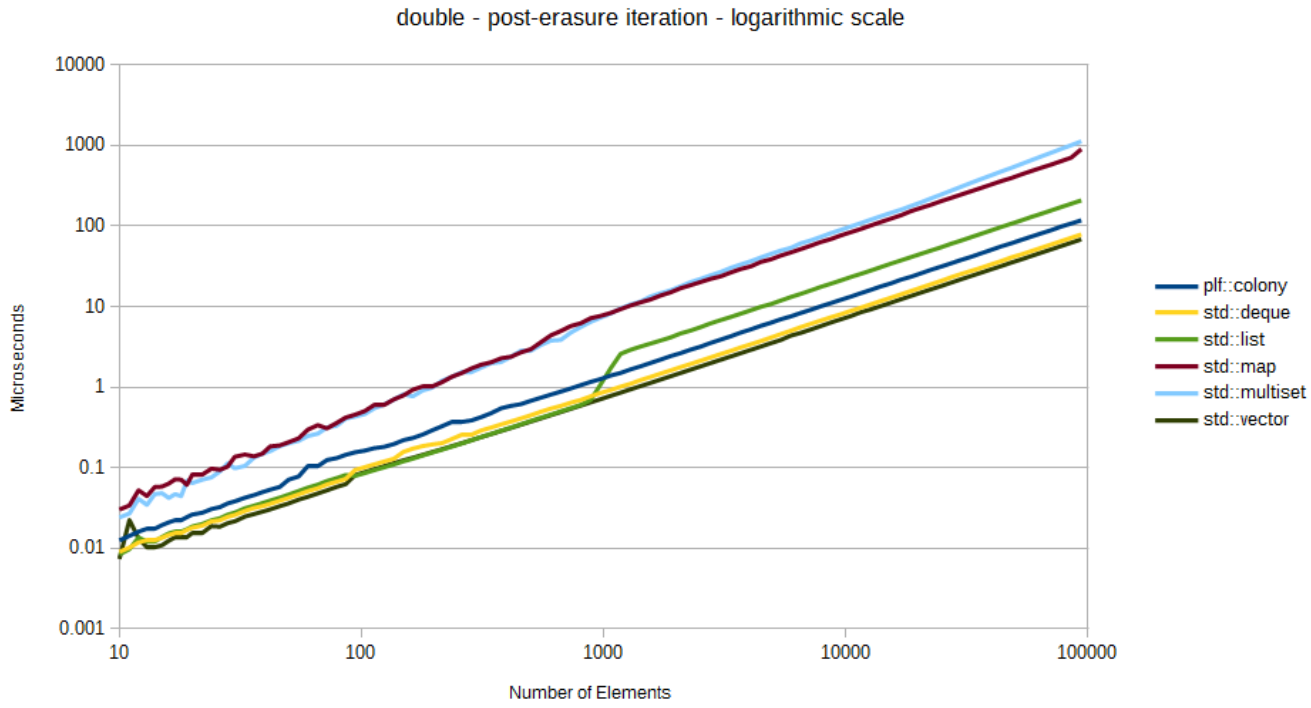
Across all types plf::colony dominates performance, with std::list coming close behind. std::deque and std::vector have predictably poor performance as a remove_if pattern is not being used, as much as 100000x worse than plf::colony and std::list for large numbers of large types.

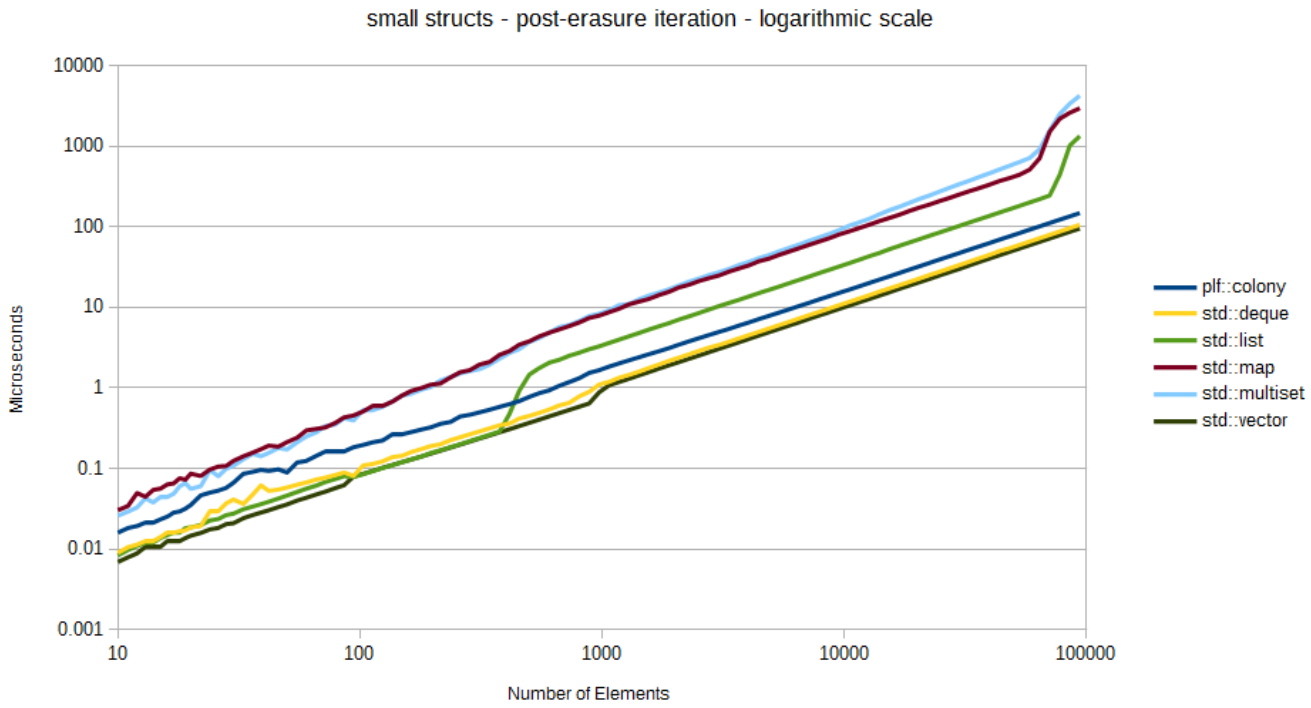
Post-erasure Iteration Performance

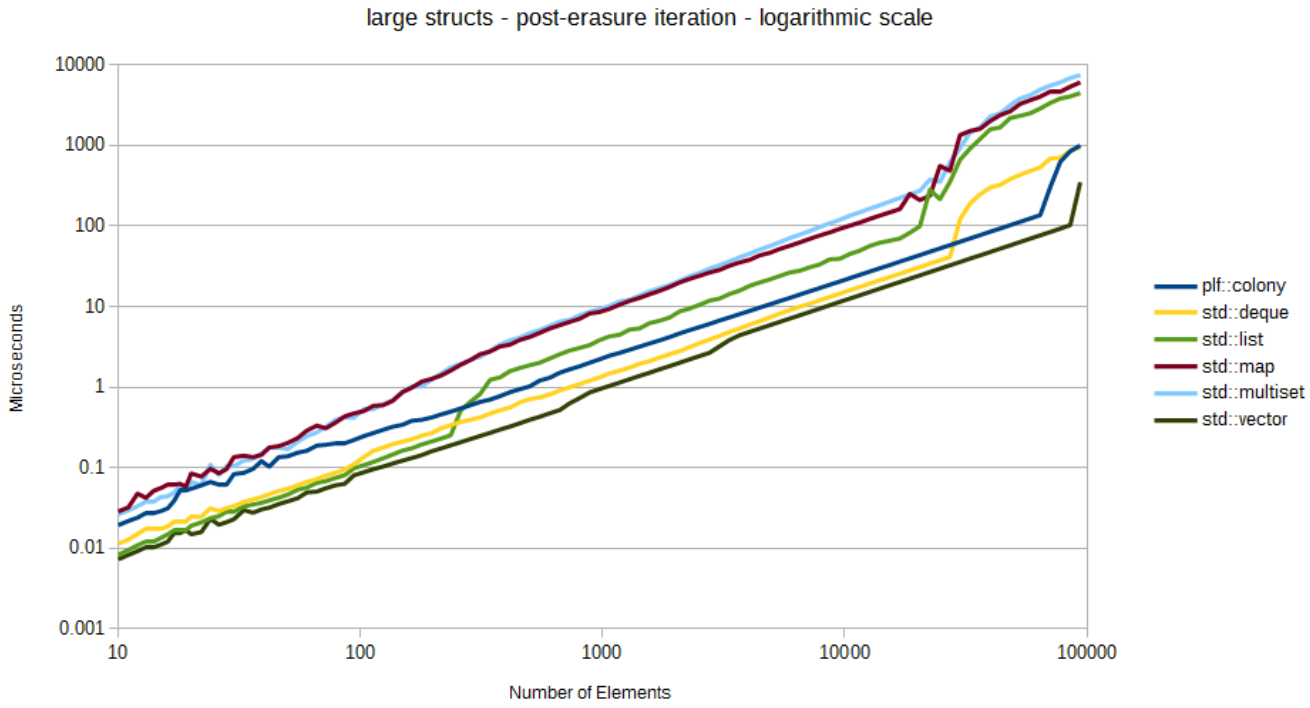
Since data is typically iterated across more than it is erased or inserted, iteration speed is, for many areas, more important than erase or insertion performance, despite the fact that it almost always takes factors of ten less time than either of those two.


char - post-erasure iteration - logarithmic scale
Microseconds
10000
1000
100
10
1
0.1
0.01
0.001
10
100
1000
10000
100000
Number of Elements
plf::colony
std::deque
std::list
std::map
std::multiset
std::vector


int - post-erasure iteration - logarithmic scale
Microseconds
10000
1000
100
10
1
0.1
0.01
0.001
10
100
1000
10000
100000
Number of Elements
plf::colony
std::deque
std::list
std::map
std::multiset
std::vector


double - post-erasure iteration - logarithmic scale
10000
1000
100
10
1
0.1
0.01
0.001
Microseconds
10
100
1000
10000
100000
Number of Elements
plf::colony
std::deque
std::list
std::map
std::multiset
std::vector


small structs - post-erasure iteration - logarithmic scale
10000
1000
100
10
1
0.1
0.01
0.001
Microseconds
10
100
1000
10000
100000
Number of Elements
plf::colony
std::deque
std::list
std::map
std::multiset
std::vector



std::vector and std::deque come in first and second place for most types, with colony in third place - the only place where this does not occur is for large structs, where colony dominates std::deque after approximately 20000 elements are inserted. Once again this is due to the structure of deque as explained in the insertion conclusion above.

For under 1000 elements, std::list is about on par with both std::deque and std::vector, both of which dominate these tests, with std::vector taking 1st place. However the number of elements necessary before this effect occurs on std::list decreases according to how large the stored type is, suggesting that performance in this case is due to some effect of the cpu cache or implementation. Querying the GCC mailing list about this resulted in the following response, which I believe to be accurate due to the correlation between std::list iteration performance and type size: "I suspect that for working sets which fit into the first-level cache of the CPU, the simpler iterators for std::list are faster than std::deque because the additional memory accesses in std::list are cheaper than the fairly complicated iterator implementation in std::deque". What this suggests is that for typical programs, where more than one data set is competing for space in the L1 or L2 caches, std::list performance will not follow the pattern above and generally will be poor.

Raw tests Conclusion

From the above data we can see that std::list is not a good contendor against plf::colony, as it has weaker insertion and erase performance, and the only scenario where it has good iteration performance is where (a) the amount of data in the container is small enough to fit entirely into the cache and (b) where that data set is the only data set being operated on by the program in question, and in fact the computer as a whole. That being a relatively uncommon case, std::list is not a general contendor.

std::deque is a contendor, having strong insertion performance and excellent iteration performance but poor non-remove_if erasure performance - however std::deque invalidates pointers upon erasure, meaning it requires modification to be used in a way comparable to colony. std::vector is a slightly weaker contendor, having weak insertion performance and worse non-remove_if erasure performance than std::deque, however it's iteration performance is always the best, being entirely contiguous in memory, rather than deque which is only partially contiguous. std::vector invalidates pointers on both insertion and erasure, meaning it will also require modification to compare to colony.

Comparative performance tests

Colony is primarily designed for scenarios where good insertion/erasure/iteration performance is required while guarantee'ing linkage stability for outside elements referring to elements within the container, and where ordered insertion is unimportant. The two containers from the raw performance tests which may compare both in performance and usage (after modification) are std::deque and std::vector. std::list does not meet these requirements as it has poor insertion and iteration performance.

pointer_deque and indexed_vector

Because std::deque does not invalidate pointers to elements upon insertion to the back or front, we can guarantee that pointers won't be invalidated during unordered insertion. This means we can use a modification called a 'pointer-to-deque deque', or pointer_deque. Here we take a deque of elements and construct a secondary deque containing pointers to each element in the first deque. The second deque functions as an erasable iteration field for the first deque ie. when we erase we only erase from the pointer deque, and when we iterate, we iterate over the pointer deque and access only those elements pointed to by the pointer deque. In doing so we reduce erase times for larger-than-scalar types, as it is computationally cheaper to reallocate pointers (upon erasure) than larger structs. By doing this we avoid reallocation during erasure for the element deque, meaning pointers to elements within the element deque stay valid.

We cannot employ quite the same technique with std::vector because it reallocates during insertion to the back of the vector upon capacity being reached. But since indexes stay valid regardless of a vector reallocates, we can employ a similar tactic using indexes instead of pointers; which we'll call an indexed_vector. In this case we use a secondary vector of indexes to iterate over the vector of elements, and only erase from the vector of indexes. This strategy has the advantage of potentially lower memory usage, as the bitdepth of the indexes can be reduced to match the maximum known number of elements, but it will lose a small amount of performance due to the pointer addition necessary to utilise indexes instead of pointers. In addition outside objects refering to elements within the indexed_vector must use indexes instead of pointers to refer to the elements, and this means the outside object must know both the index and the container it is indexing; whereas a pointer approach can ignore this and simply point to the element in question.

We will also compare these two container modifications using a *remove_if* pattern for erasure vs regular erasure, by adding an additional boolean field to indicate erasure to the original stored struct type, and utilizing two passes - the first to randomly flag elements as being ready for erasure via the boolean field, the second using the *remove_if* pattern.

vector_bool and deque_bool

A second modification approach, which we'll call a vector_bool, is a very common approach in a lot of game engines - a bool or similar type is added to the original struct or class, and this field is tested against to see whether or not the object is 'active' (true) - if inactive (false), it is skipped over. We will also compare this approach using a deque.

packed_deque

packed_deque is an implementation of a 'packed_array' as described in the motivation section earlier, but using deques instead of vectors or arrays. As we've seen in the raw performance benchmarks, (GCC) libstdc++'s deque is almost as fast as vector for iteration, but about twice as fast for back insertion and random location erasure. It also doesn't invalidate pointers upon insertion, which is also a good thing. These things become important when designing a container which is meant to handle large numbers of insertions and random-location erasures. Although in the case of a packed-array, random location erasures don't really happen, the 'erased' elements just get replaced with elements from the back, so erasure speed is not as critical, but insertion speed is critical as it will always consume significantly more CPU time than iteration.

With that in mind my implementation uses two std::deque's internally: one containing structs which package together the container's element type and a pointer, and one containing pointers to each of the 'package' structs in the first deque. The latter is what is used by the container's 'handle' class to enable external objects to refer to container elements. The pointer in the package itself in turn points to the package's corresponding 'handle' pointer in the second deque. This enables the container to update the handle pointer when and if a package is moved from the back upon an erasure.

Anyone familiar with packed array-style implementations can skip this paragraph. For anyone who isn't, this is how it works when an element is erased from packed_deque, unless the element in question is already at the back of the deque. It:

1. Uses the pointer within the package to find the 'handle' pointer which pointed to the erased element, and adds it to a free list.
2. Moves the package at the back of the container to the location of the package containing the element being erased.
3. Uses the pointer in the package which has just been moved to update the corresponding handle pointer, to correctly point to the package's new location.
4. Pops the back package off the first deque (should be safe to destruct after the move - if it's not, the element's implementation is broke).

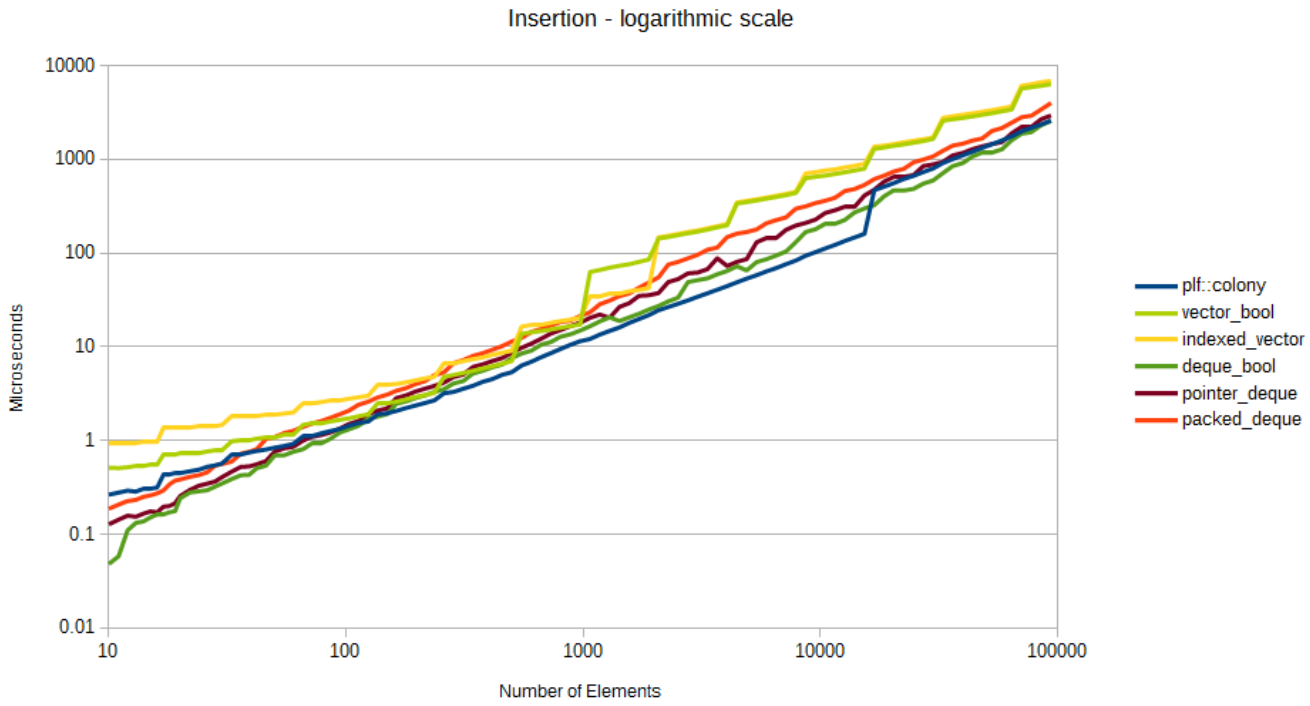
In this way, the data in the first deque stays contiguous and is hence fast to iterate over. And any handles referring to the back element which got moved stay valid after the erasure.

This implementation will not work well under MSVC as MSVC's deque implementation performs badly.

Tests

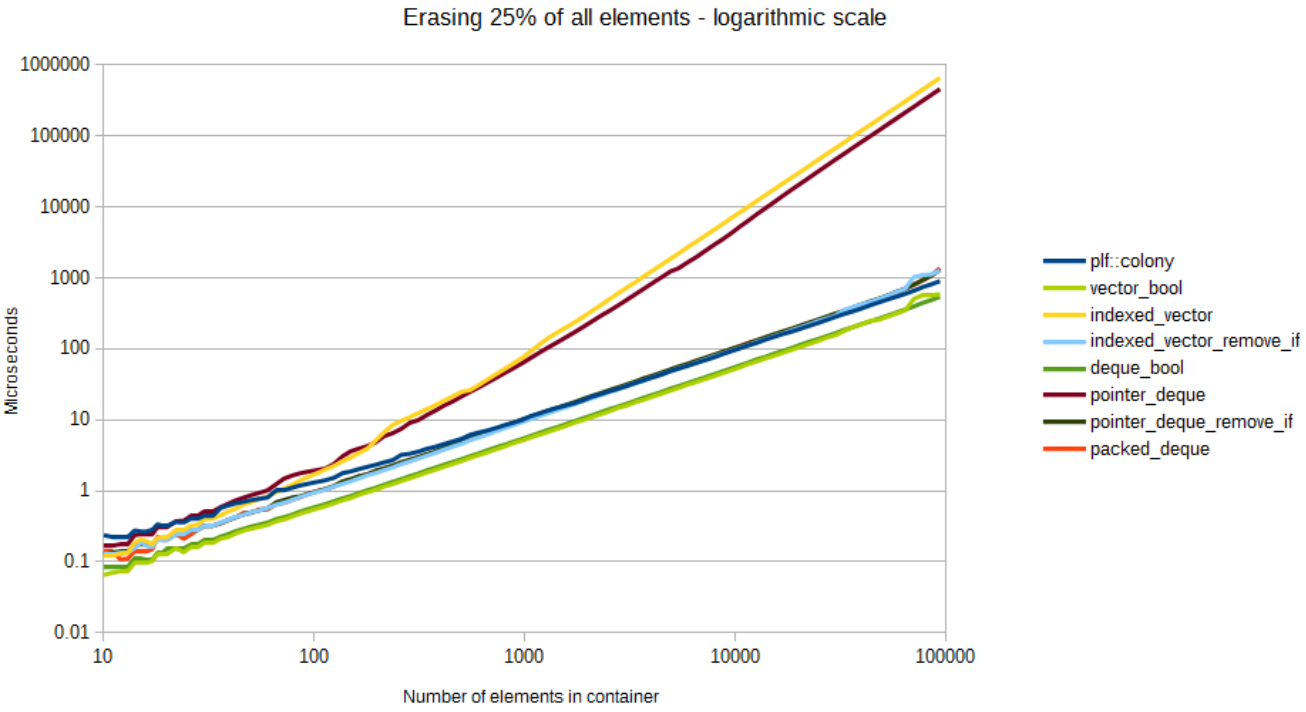
Since neither indexed_vector nor pointer_deque will have erasure time benefits for small scalar types, and because game development is predominantly concerned with storage of larger-than-scalar types, we will only test using small structs from this point onwards. In addition, we will test 4 levels of erasure and subsequent iteration performance: 0% of all elements, 25% of all elements, 50% of all elements, and 75% of all elements.

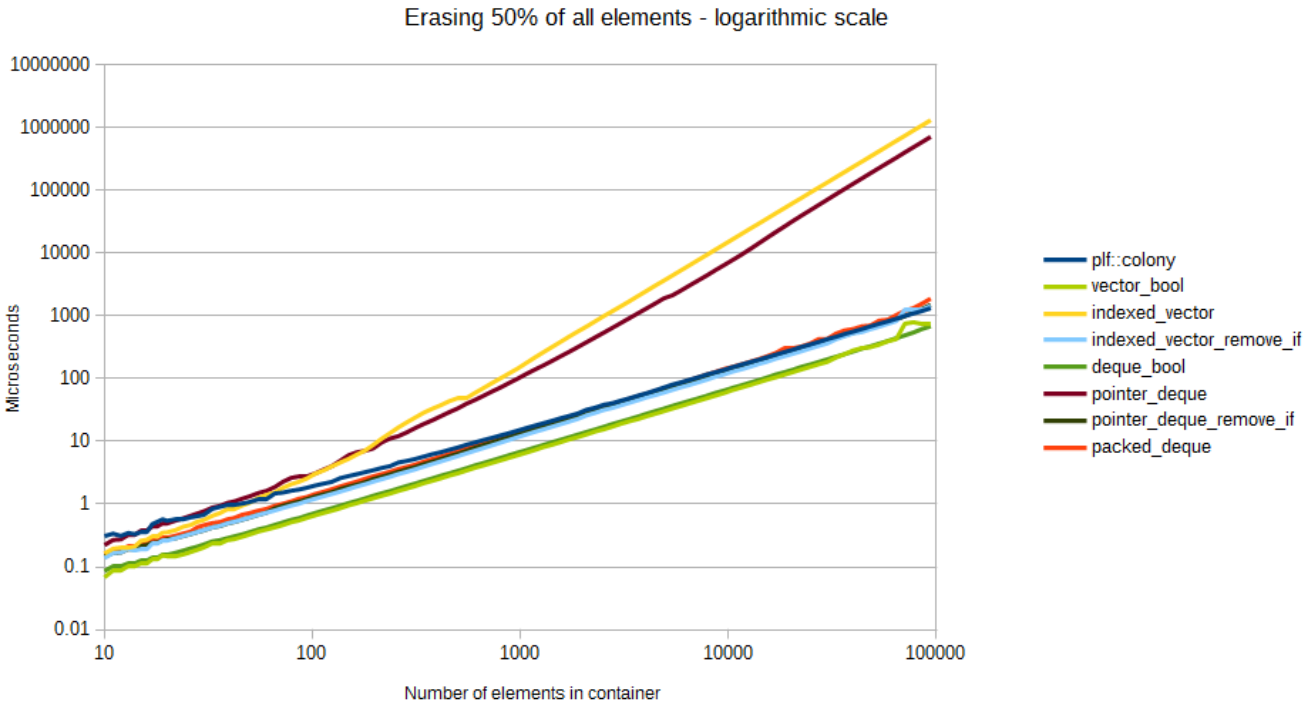
Insertion

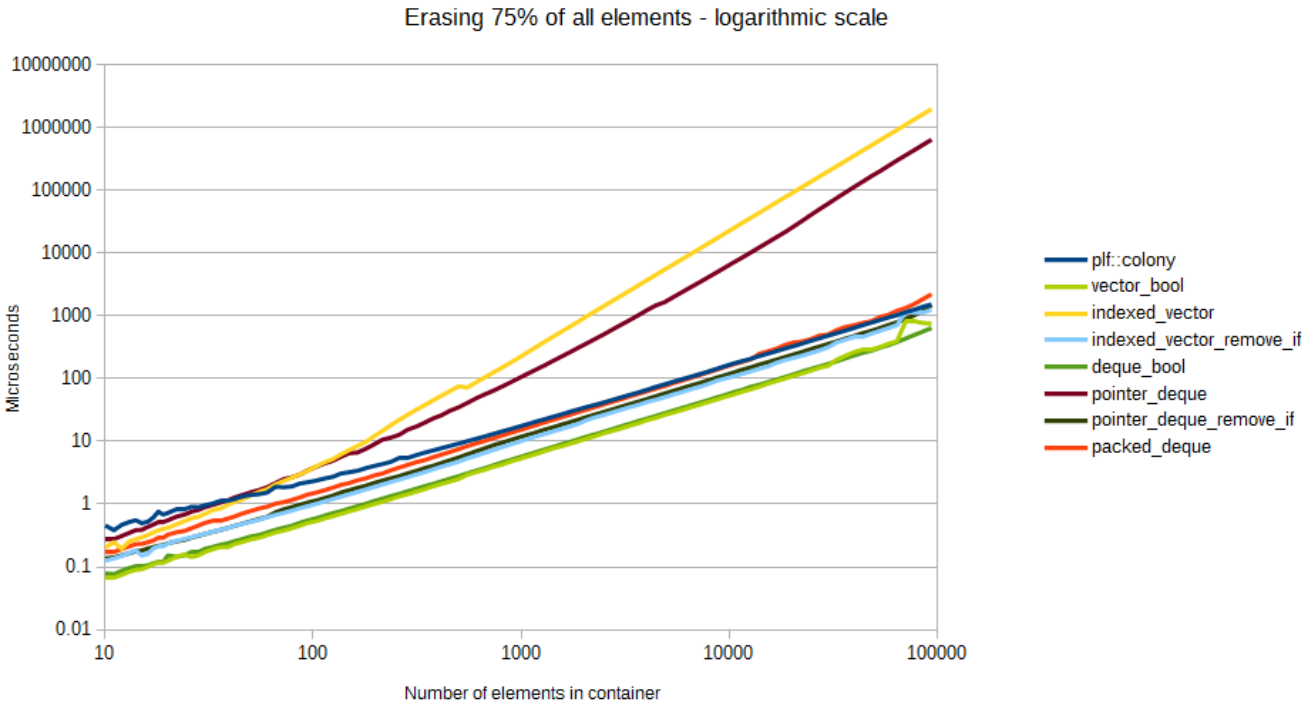


For insertion plf::colony outperforms the others for greater than 100 and less than 20000 elements. Below 100 elements it is outperformed by pointer_deque and deque_bool, and above 20000 elements it is outperformed by deque_bool. packed_deque consistently comes 4th, and both vector methods perform poorly by contrast.

Erasure

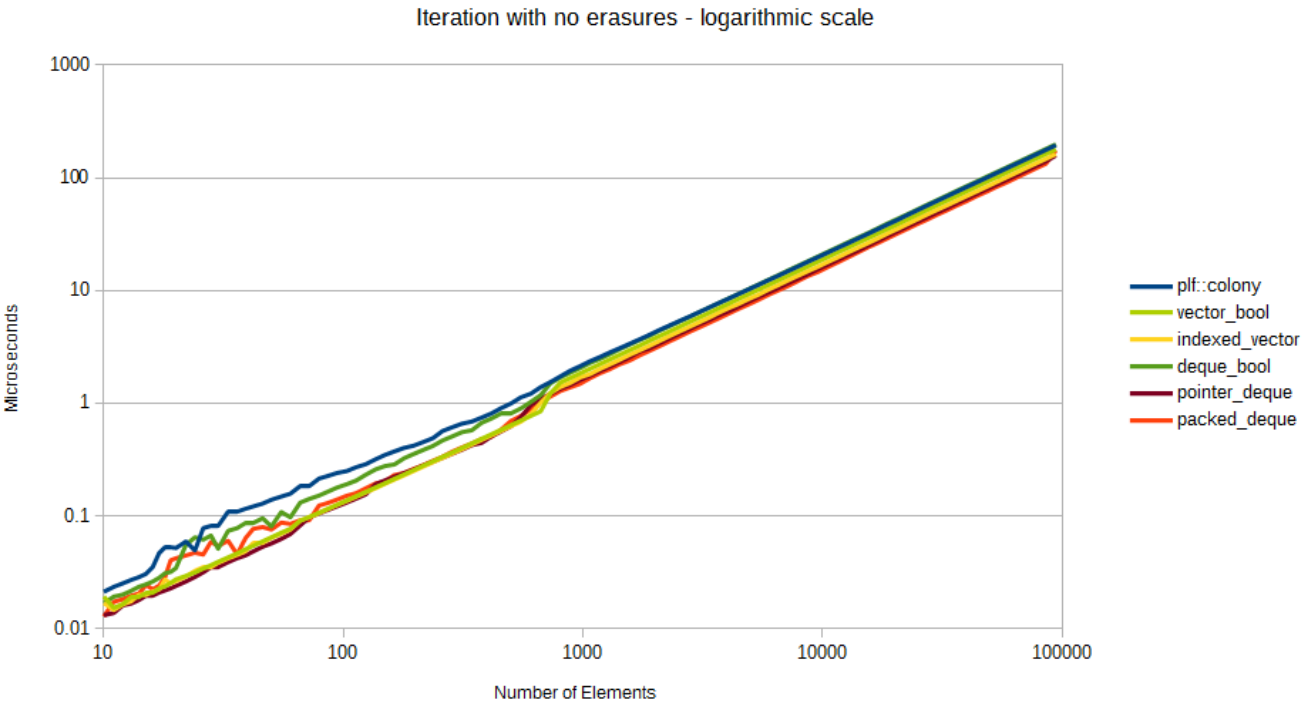

Erasing 25% of all elements - logarithmic scale
Microseconds
1000000
100000
10000
1000
100
10
1
0.1
0.01
10
100
1000
10000
100000
Number of elements in container
plf::colony
vector_bool
indexed_vector
indexed_vector_remove_if
deque_bool
pointer_deque
pointer_deque_remove_if
packed_deque


Erasing 50% of all elements - logarithmic scale
Microseconds
10000000
1000000
100000
10000
1000
100
10
1
0.1
0.01
10
100
1000
10000
100000
Number of elements in container
plf::colony
vector_bool
indexed_vector
indexed_vector_remove_if
deque_bool
pointer_deque
pointer_deque_remove_if
packed_deque



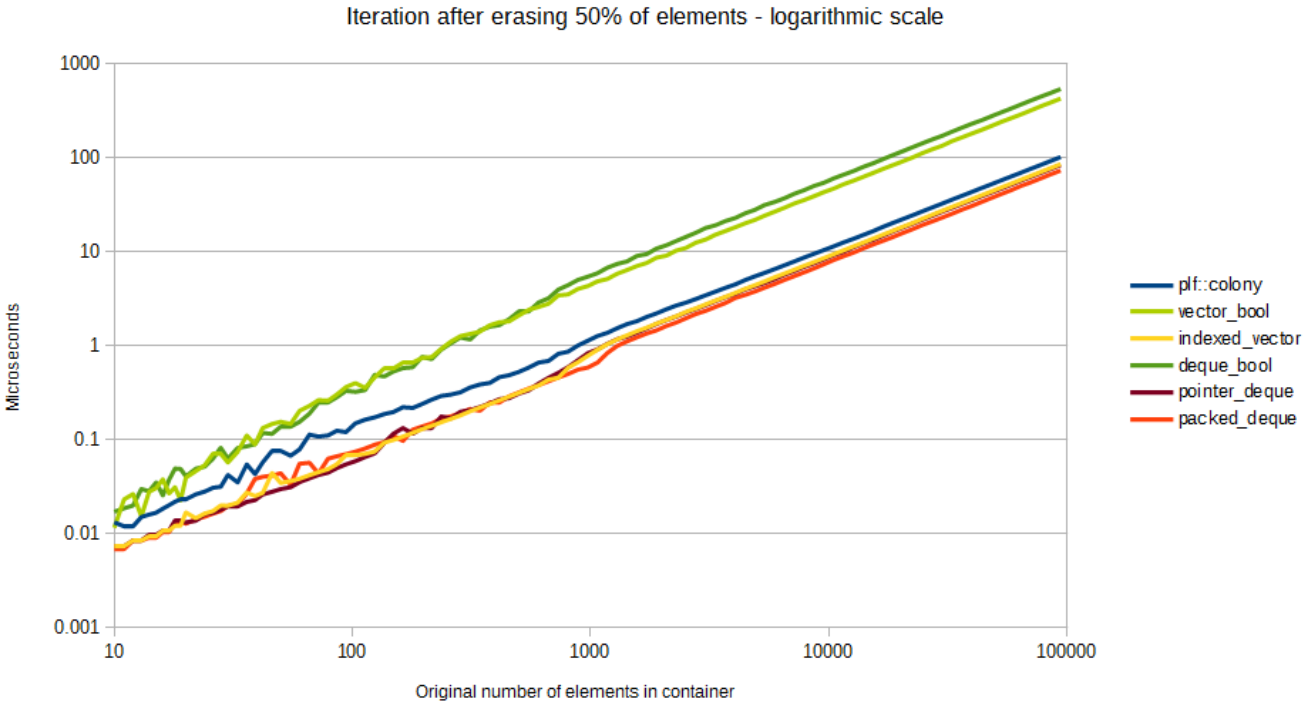
Here the gap consistently widens between the candidates as erasure percentage increases. The two boolean skipfield methods obviously dominate performance, being the easiest and fastest to implement in terms of erasure. Above 25% erasure both of the remove_if variants outperform the others, with packed_deque and colony criss-crossing each other in terms of performance. The non-remove_if variants of pointer_deque and indexed_vector of course perform poorly.

Post-erasure Iteration

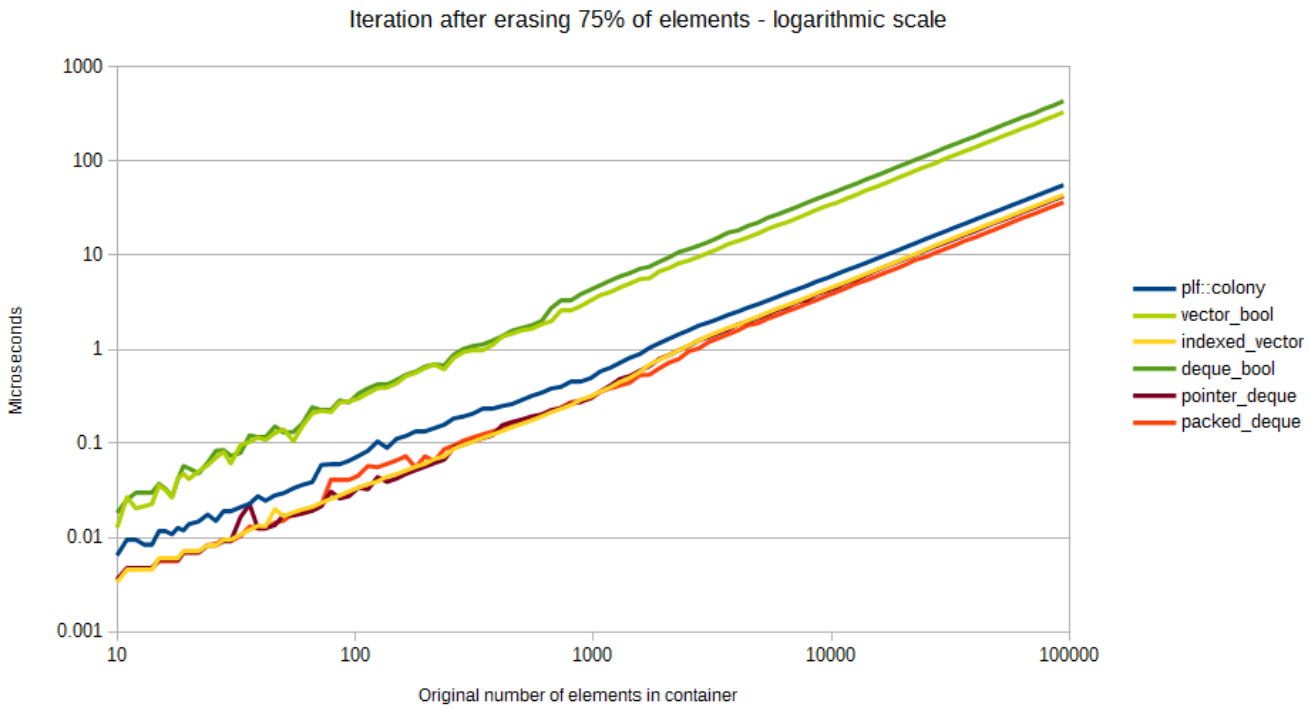


Colony performs the worst out of the lot for iteration with zero erasures, with deque_bool coming in slightly worse only for large numbers of elements. Unsurprisingly packed_deque performs the best out of the lot as this constitutes pure contiguous iterative performance with no skipfield or iteration field. While close, the pointer_deque approach has a slight performance advantage over the indexed_vector.

Now we begin to see the advantage of a jump-counting skipfield over a boolean skipfield. Because boolean skipfields require branching code to iterate over, and 25% of all elements being erased represents a large number of cache misses. Other than that the results are much the same as the 0% test.



At 50% randomised erasures, a CPU's branch prediction cannot work at all, and so the boolean approaches degrade significantly.



At this point we can clearly see that the boolean approaches are not useful in terms of iteration.

Summary: for iteration packed_deque comes 1st, pointer_deque 2nd, indexed_vector 3rd, colony 4th, vector_bool 5th and deque_bool 6th.

'Real-world' scenario testing - low modification

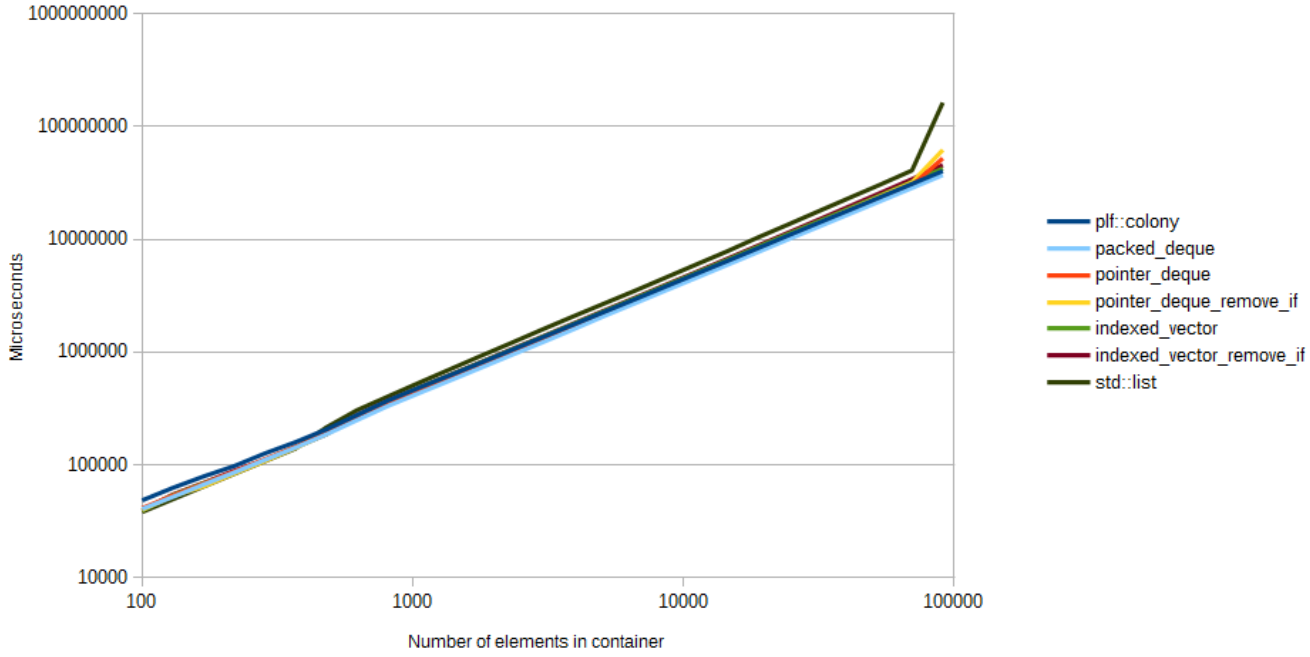
While testing iteration, erasure and insertion separately can be a useful tool, they don't tell us how containers perform under real-world conditions, as under most use-cases we will not be simply inserting a bunch of elements, erasing some of them, then iterating once over the data. To get more valid results, we need to think about the kinds of use-cases we may have for different types of data, in this cas, video-game data.

In this test we simulate the use-case of a container for general video game objects, actors/entities, enemies etc. Initially we insert a number of small structs into the container, then simulate in-game 'frames'. We iterating over container elements every frame, and erase(at random locations)/insert 1% of the original number of elements for every minute of gametime ie. 3600 frames assuming 60 frames-per-second. We measure the total time taken to simulate this scenario for 108000 frames (half an hour of simulated game-time, assuming 60 frames per second), as well as the amount of memory used by the container at the end of the test. We then re-test this scenario with 5% of all elements being inserted/erased, then 10%.

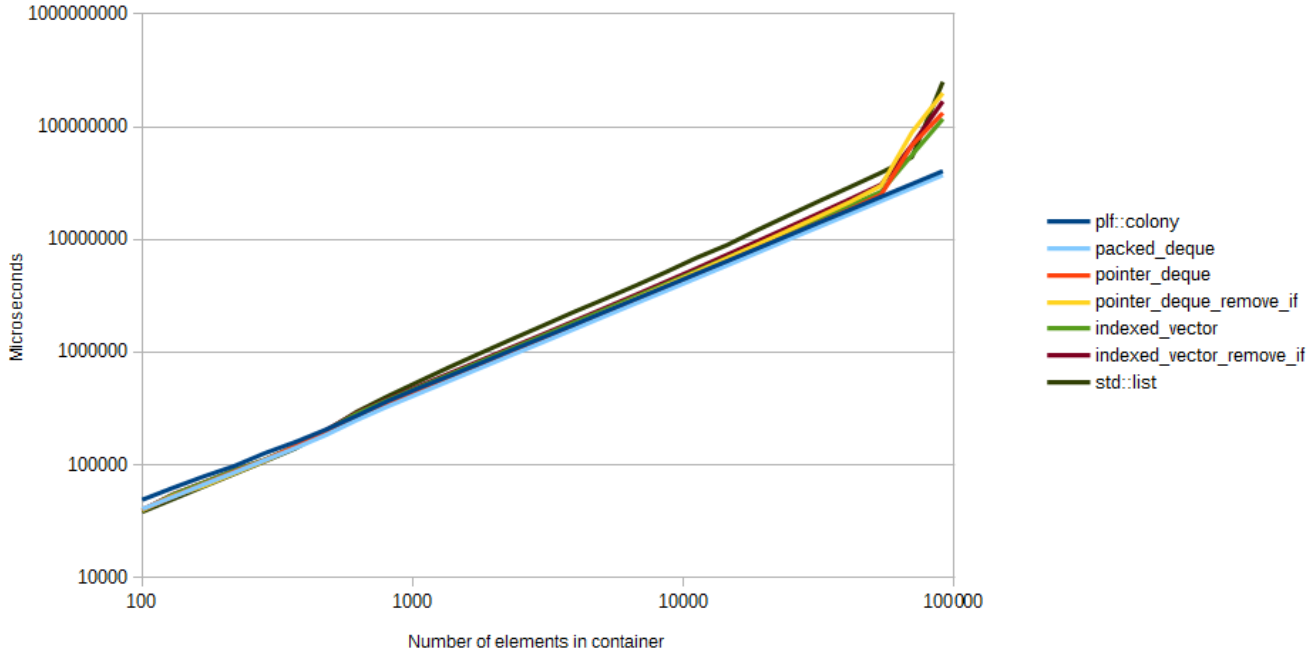
With reluctance I have included the results for std::list in this test and the high modification tests, despite the face that the earlier 'raw' performance tests show beyond a shadow of a doubt that it is not a contendor for colony. This is because some people do not correctly relate the raw performance test outcomes to the expected outcomes of subsequent tests.

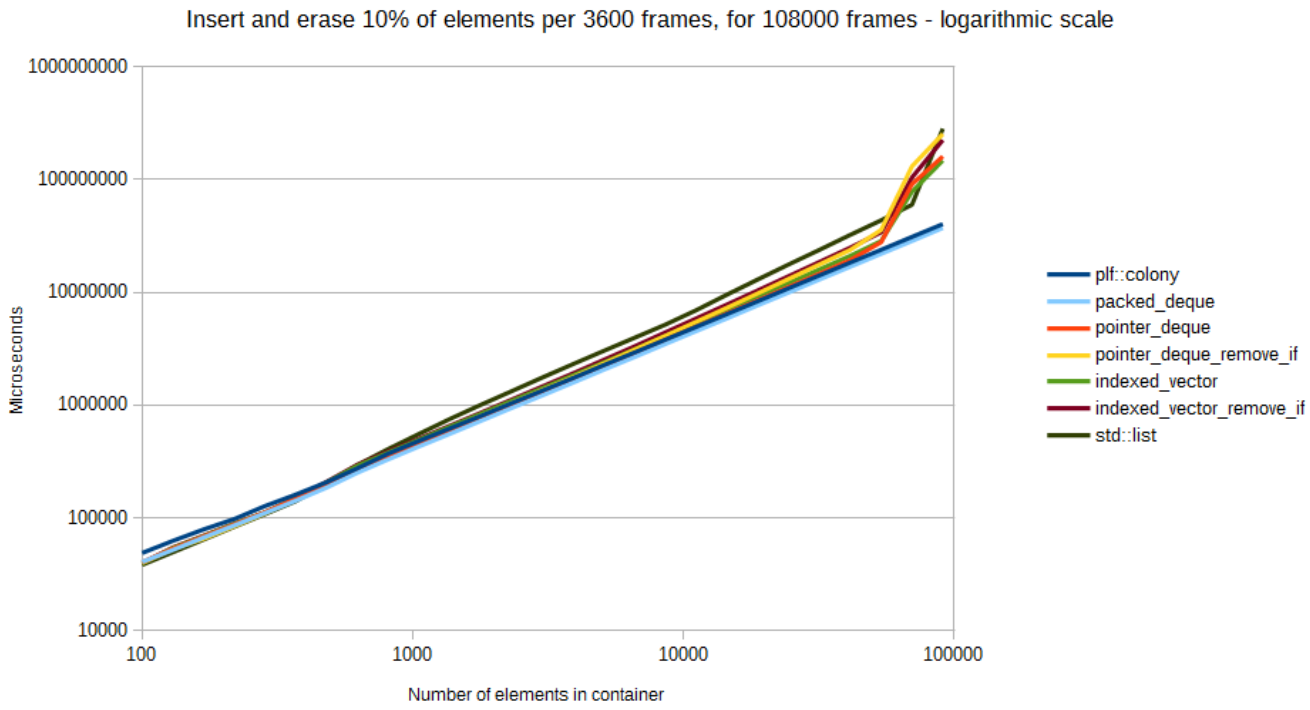
Performance results

Insert and erase 1% of elements per 3600 frames, for 108000 frames - logarithmic scale

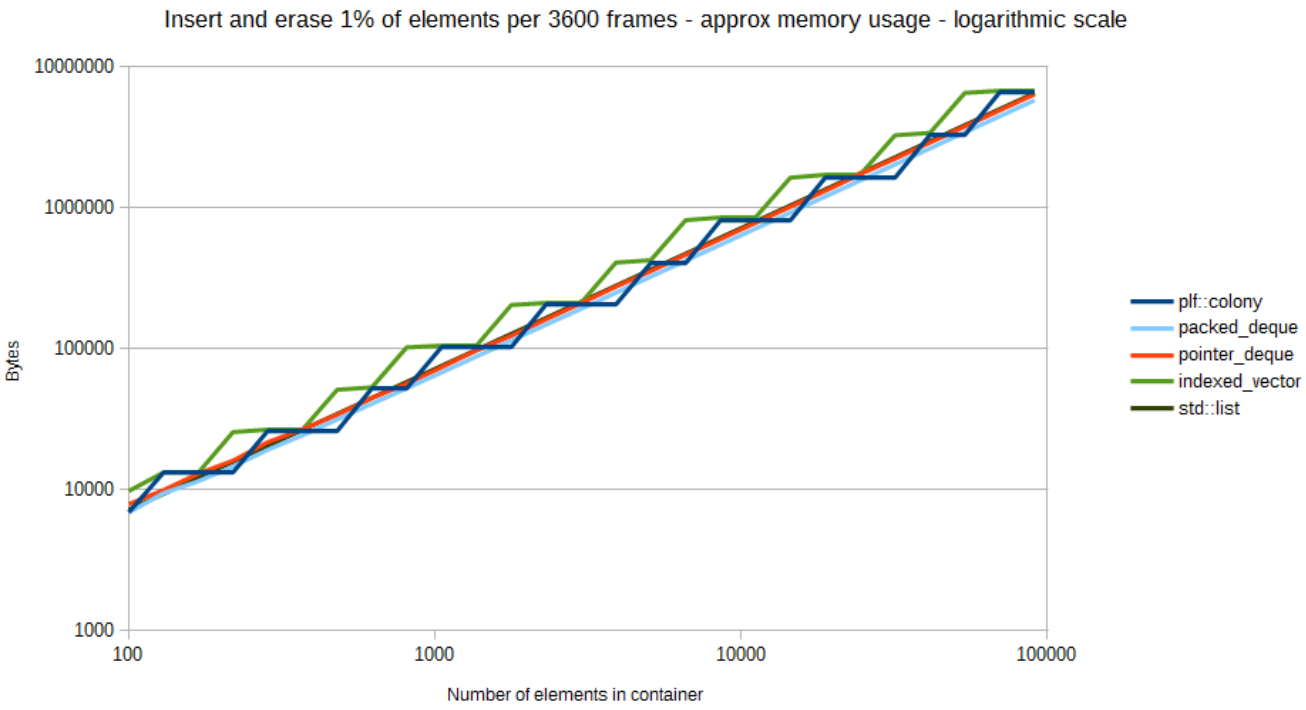


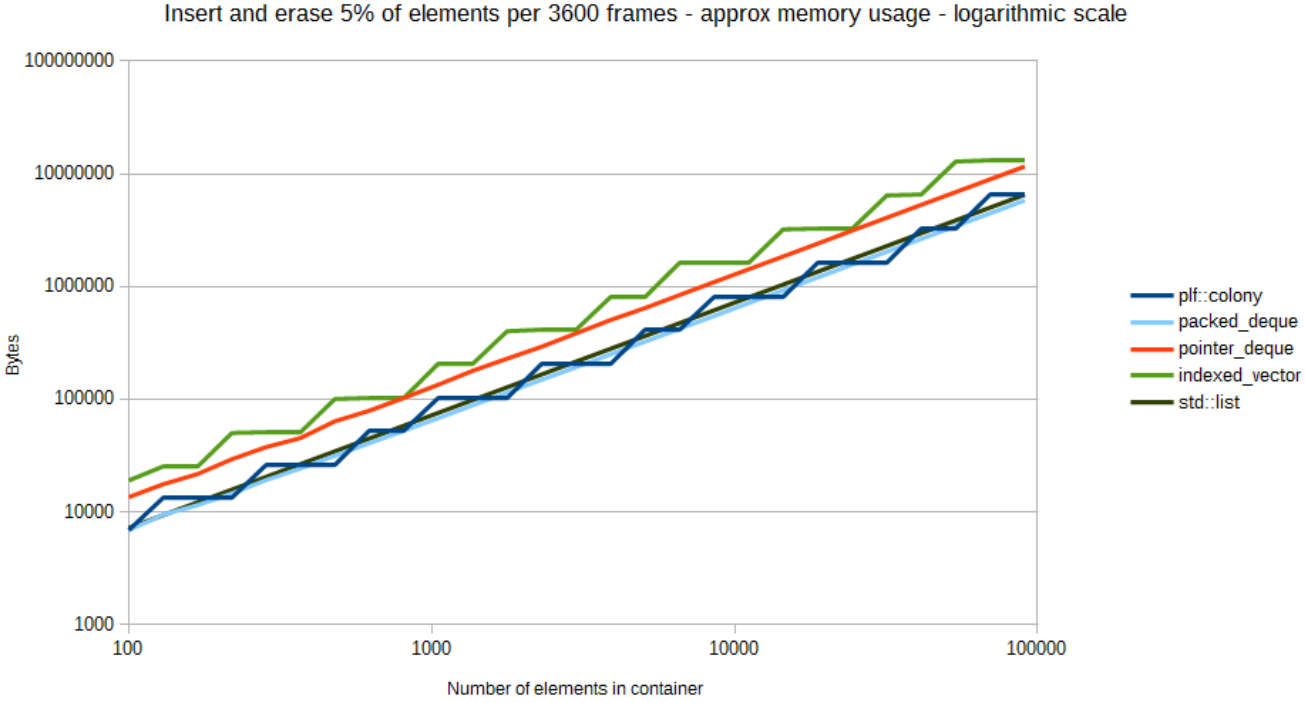
Insert and erase 5% of elements per 3600 frames, for 108000 frames - logarithmic scale

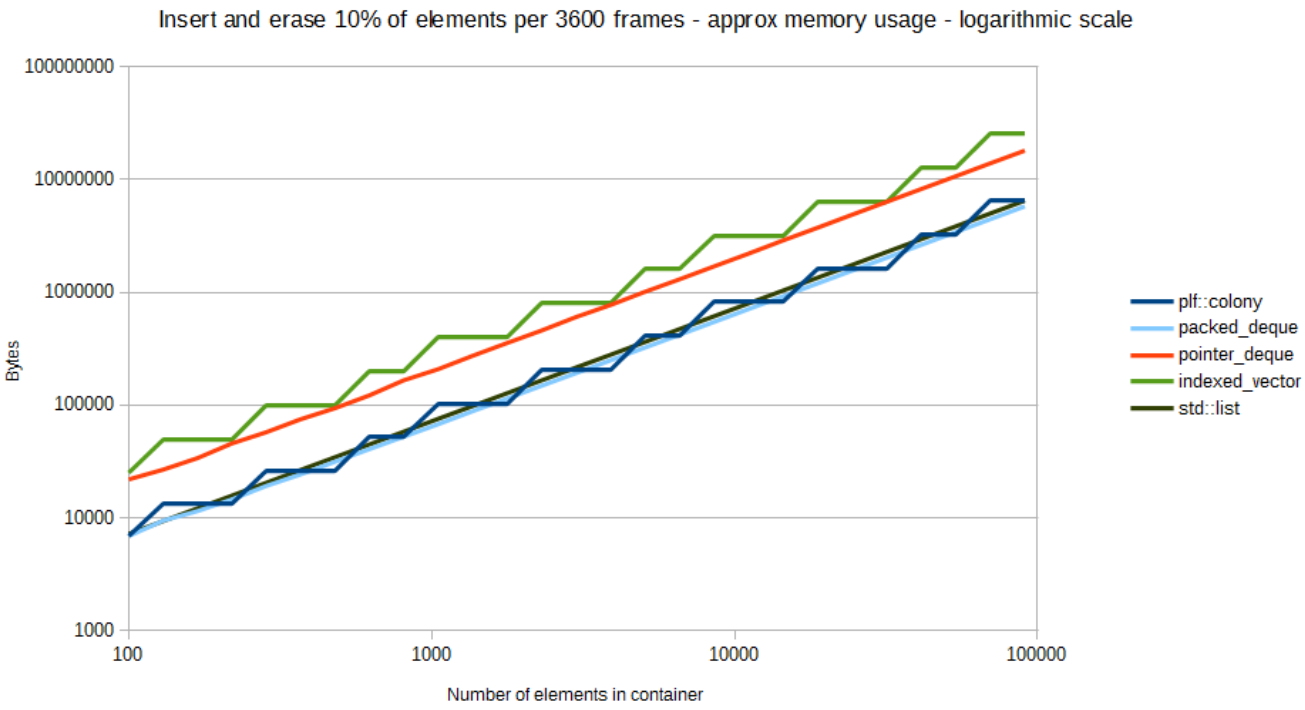




Memory results


Insert and erase 1% of elements per 3600 frames - approx memory usage - logarithmic scale
Bytes
10000000
1000000
100000
10000
1000
100
1000
10000
100000
Number of elements in container
plf::colony
packed_deque
pointer_deque
indexed_vector
std::list


Insert and erase 5% of elements per 3600 frames - approx memory usage - logarithmic scale
Bytes
100000000
10000000
1000000
100000
10000
1000
100
1000
10000
100000
Number of elements in container
plf::colony
packed_deque
pointer_deque
indexed_vector
std::list

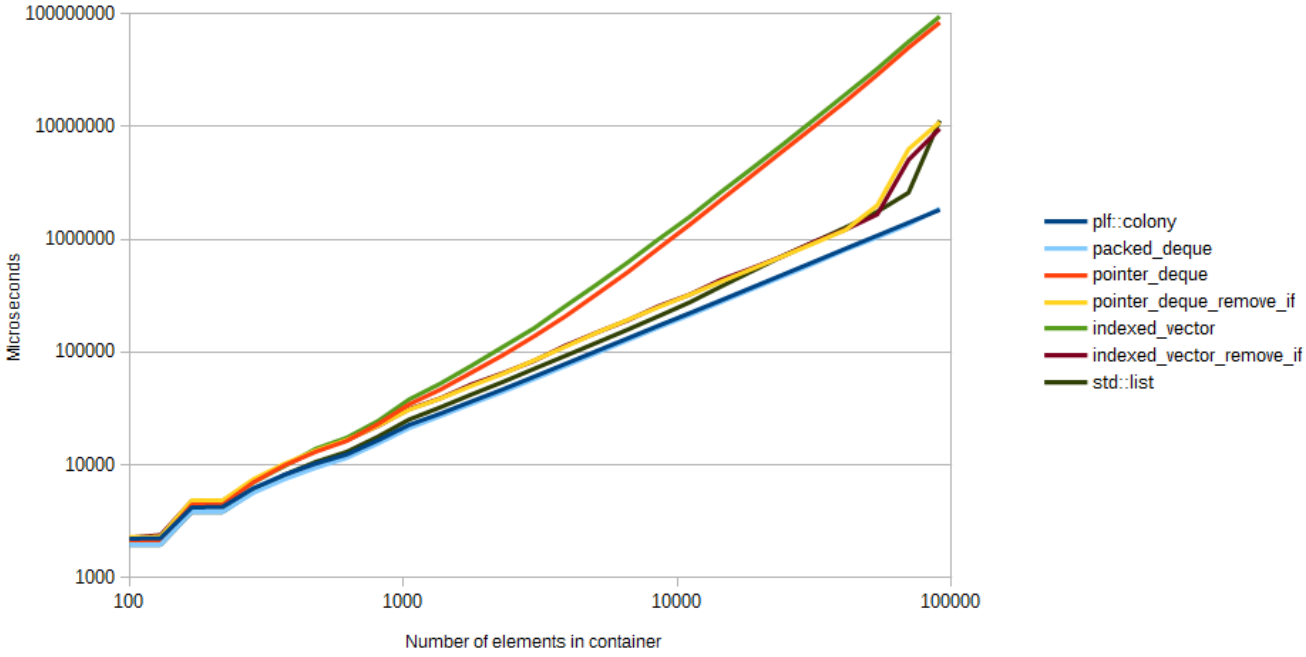


'Real-world' scenario testing - high modification

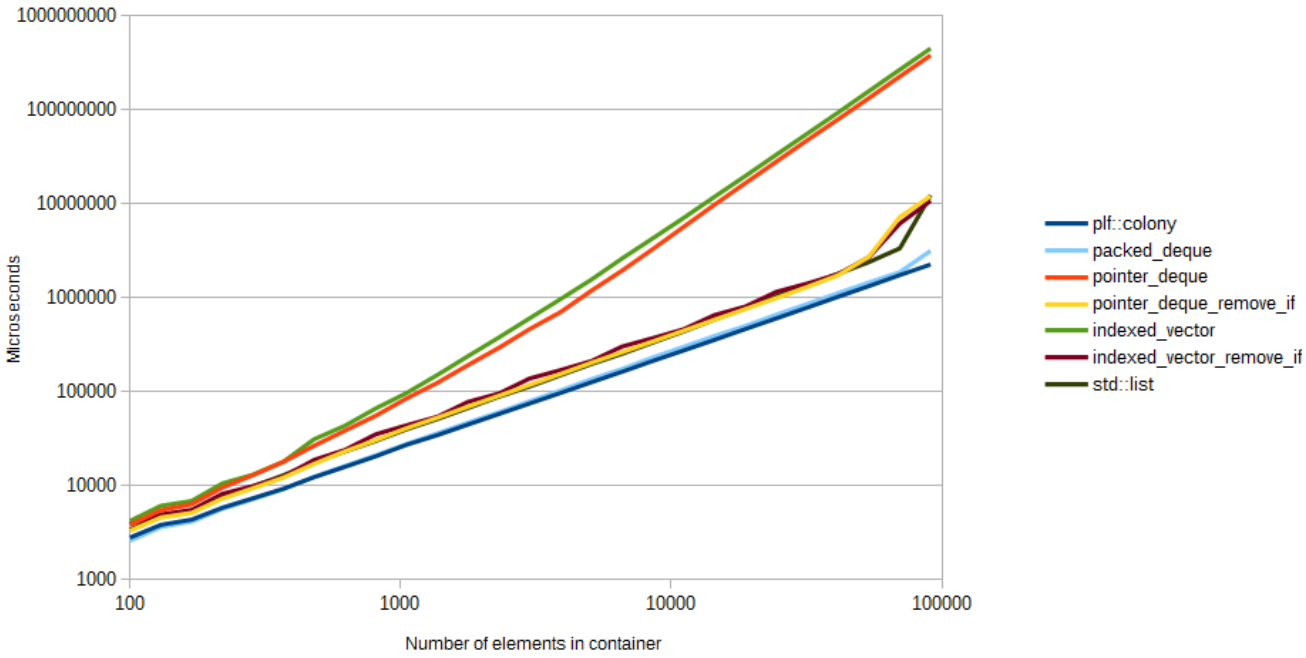
Same as the previous test but here we erase/insert 1% of all elements per-frame instead of per 3600 frames, then once again increase the percentage to 5% then 10% per-frame. This simulates the use-case of continuously-changing data, for example video game bullets, decals, quadtree/octree nodes, cellular/atomic simulation or weather simulation.

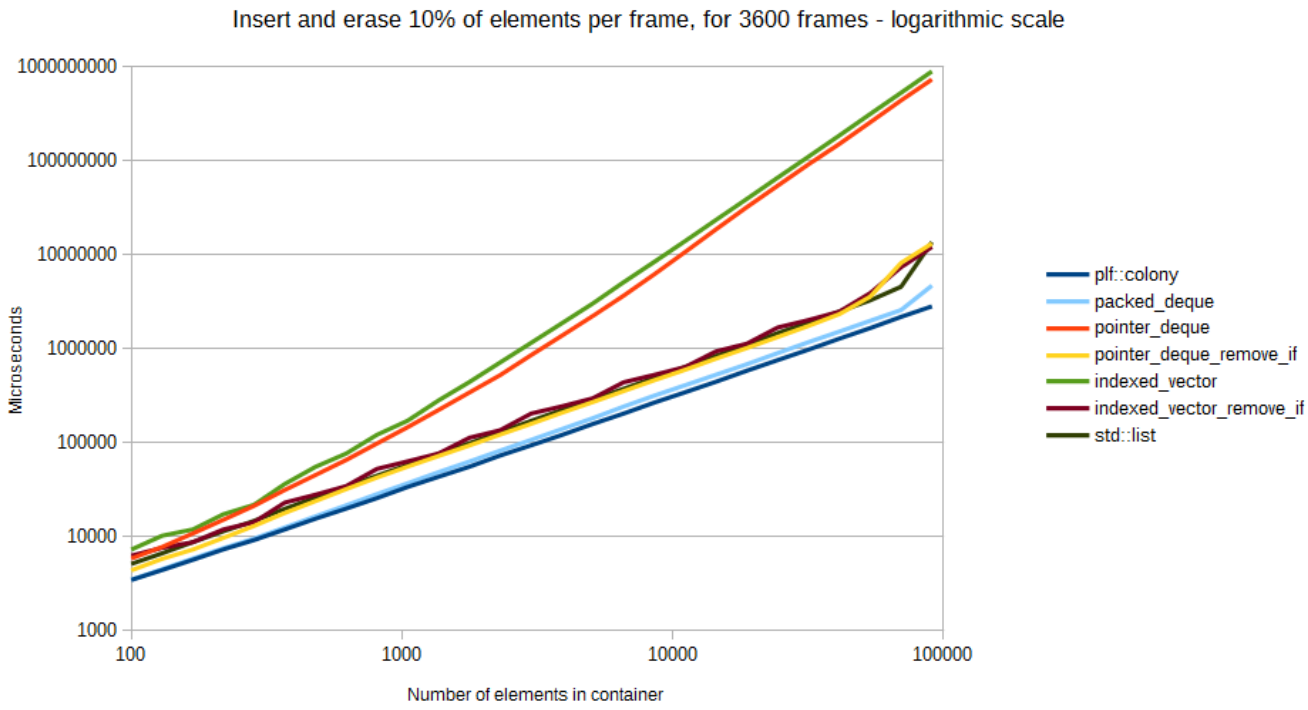
Performance results

Insert and erase 1% of elements per frame, for 3600 frames - logarithmic scale

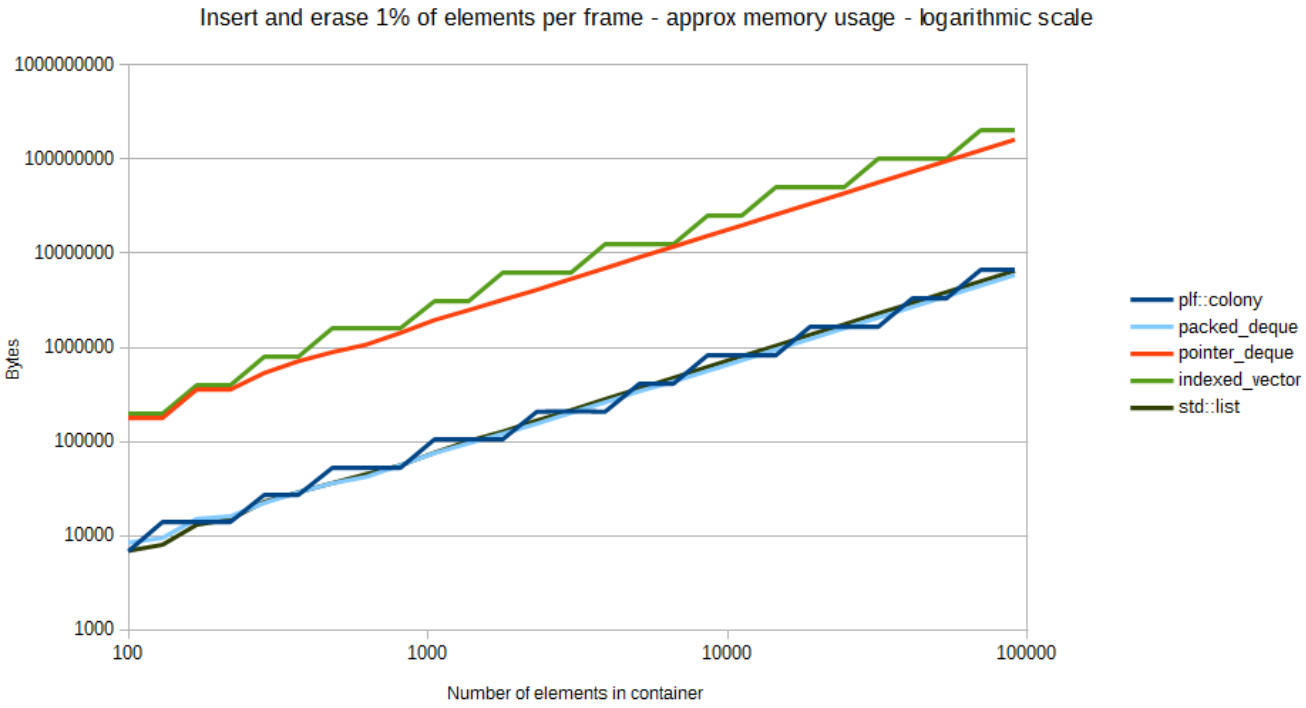


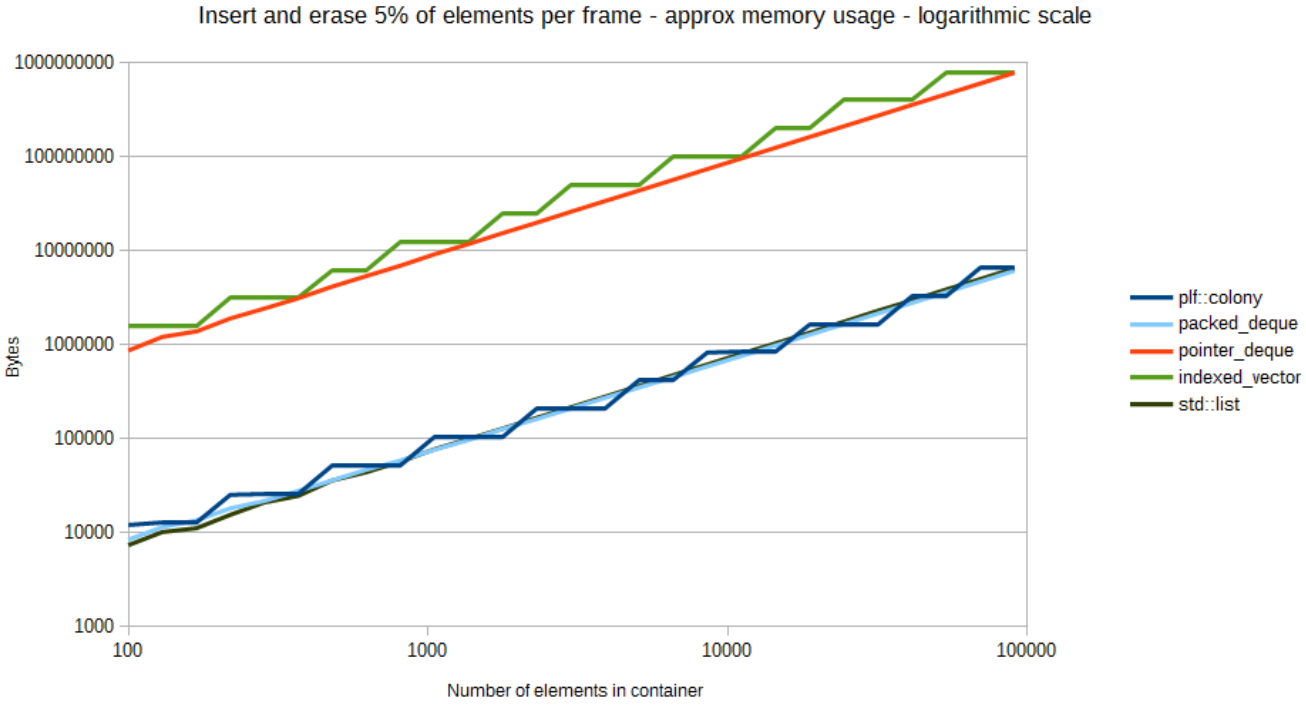
Insert and erase 5% of elements per frame, for 3600 frames - logarithmic scale

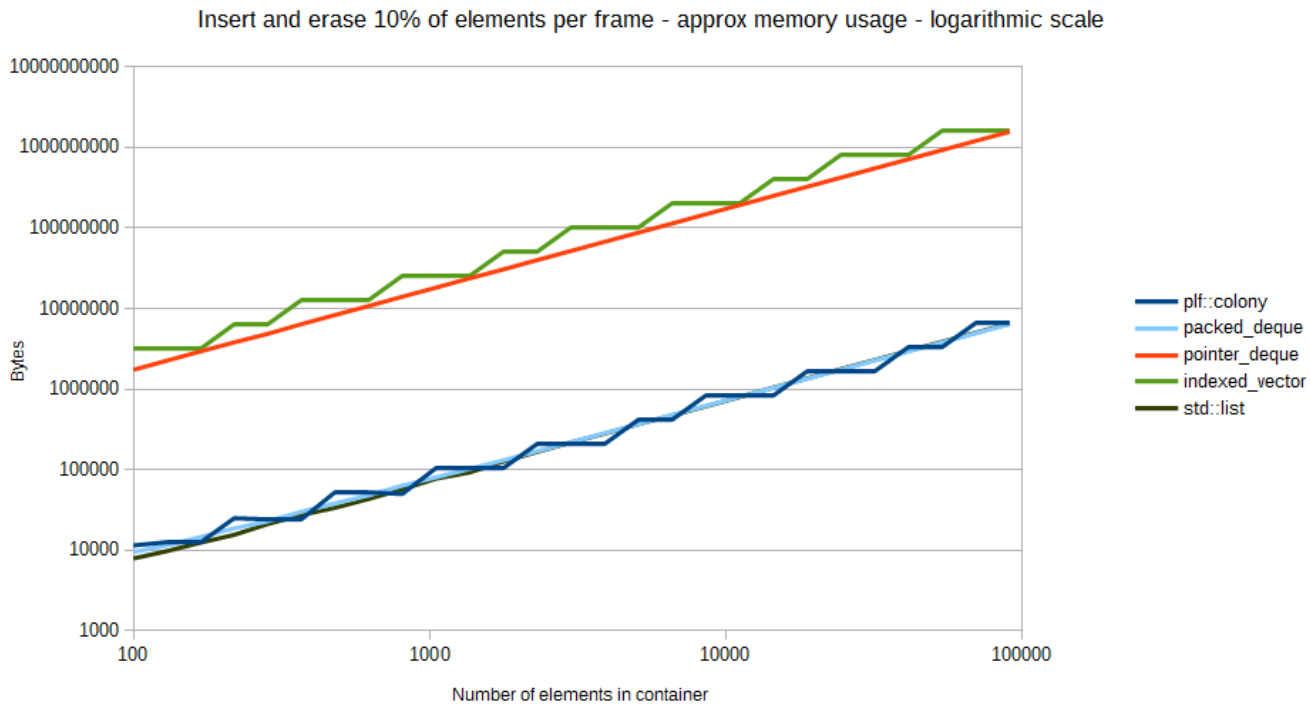




Memory results


Insert and erase 1% of elements per frame - approx memory usage - logarithmic scale
1000000000
100000000
10000000
1000000
100000
10000
1000
Bytes
100
1000
10000
100000
Number of elements in container
plf::colony
packed_deque
pointer_deque
indexed_vector
std::list


Insert and erase 5% of elements per frame - approx memory usage - logarithmic scale
1000000000
100000000
10000000
1000000
100000
10000
1000
Bytes
100
1000
10000
100000
Number of elements in container
plf::colony
packed_deque
pointer_deque
indexed_vector
std::list



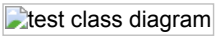
Obviously if you are doing a lot of referencing into a packed_deque from outside objects, you may encounter some speed problems due to the dereferencing system - which the next test will cover. On the other hand, if you are mainly iterating over unordered data, erasing occasionally, and only ever referring *outwards from* elements within the packed_deque, it will be an ideal solution.

'Real-world' scenario-testing: referencing with interlinked containers

In order to completely test plf::colony against a packed-array-style implementation, it is necessary to measure performance while exploiting both container's linking mechanisms - for colony this is pointers or iterators, for a packed array this is a handle, or more specifically a pointer to a pointer (or the equivalent index-based solution). Because that additional dereferencing is a performance loss and potential cache miss, any test which involves a large amount of inter-linking between elements in multiple containers should lose some small amount of performance when using a packed_deque instead of a colony. Since games typically have high levels of interlinking between elements in multiple containers (as described in the motivation section), this is relevant to performance concerns.

Consequently, this test utilizes four instances of the same container type, each containing different element types:

1. A 'collisions' container (which could represent collision rectangles within a quadtree/octree/grid/etc)
2. An 'entities' container (which could representing general game objects) and
3. Two subcomponent containers (these could be sprites, sounds, or anything else).



This is actually a very low number of inter-related containers for a game, but we're reducing the number of components in this case just to simplify the test. Elements in the 'entities' container link to elements in all three of the other containers. In turn, the elements in the collisions container link back to the corresponding elements in the entities container. The subcomponent containers do not link to anything.

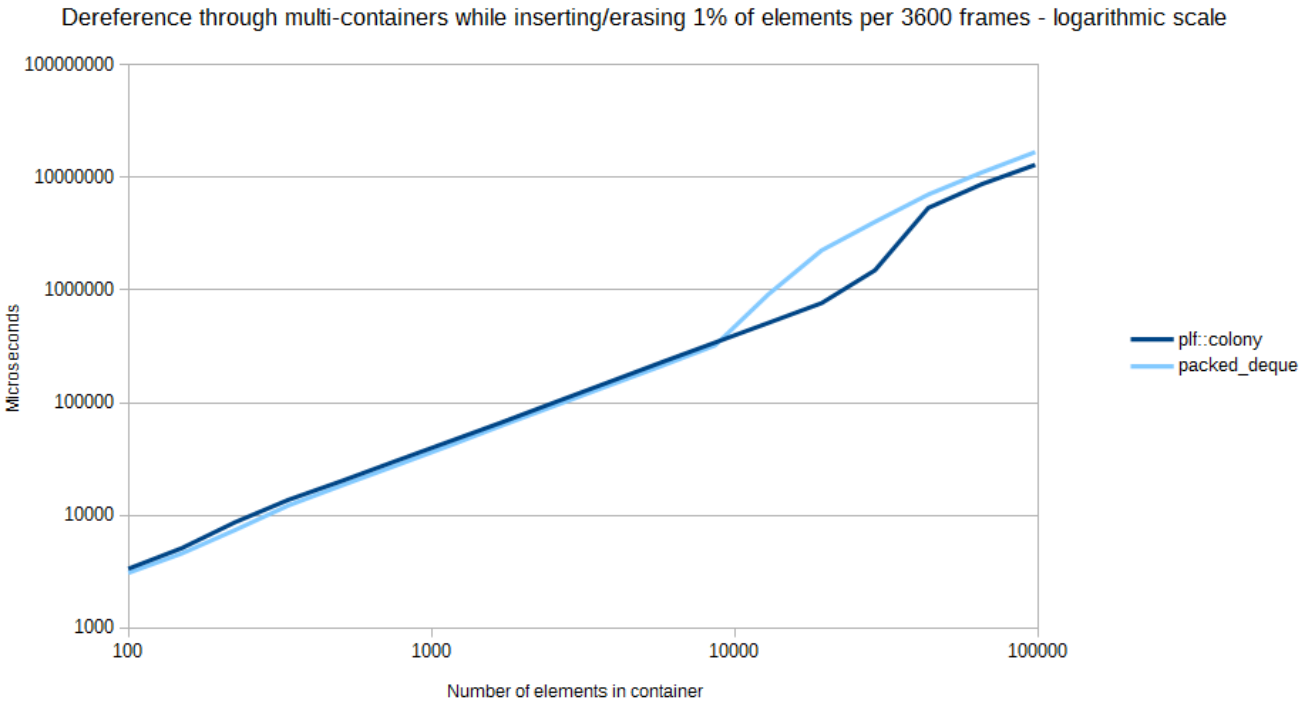
In the test, elements are first added to all four containers and interlinked (as this is a simplified test, there's a one-to-one ratio of entity elements to 'collision' and subcomponent elements). The core test process after this point is similar to the modification scenarios tested earlier: we iterate over the entity container once every frame, adding a number from both the entities and subcomponents to a total. We also erase a percentage of entities per-frame (and their corresponding subcomponents and collision blocks) - similar to the earlier tests.

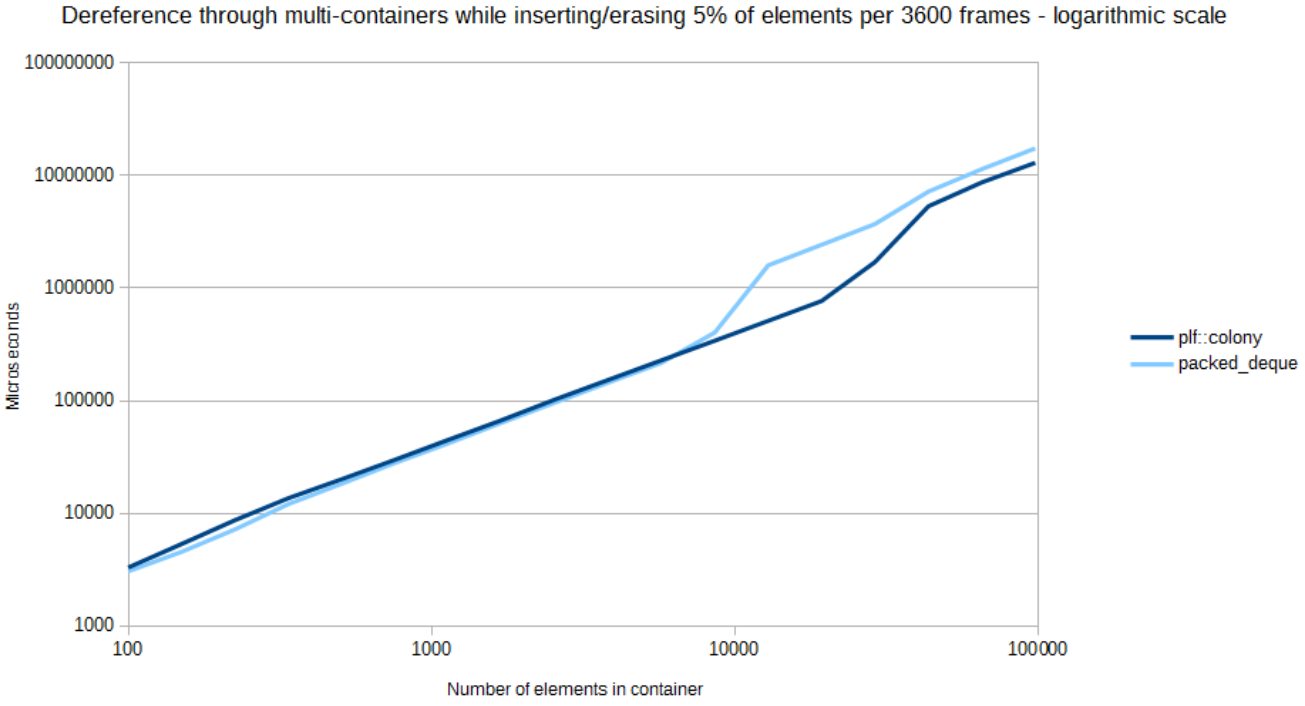
However during each frame we also iterate over the 'collisions' container and erase a percentage of these elements (and their corresponding entities and subcomponents) at random as well. This could be seen as simulating entities being removed from the game based on collisions occurring, but is mainly to test the performance effects of accessing the subcomponents via a chain of

handles versus a chain of pointers. Then, again during each frame, we re-add a certain number of entities, collision blocks and subcomponents back into the containers based on the supplied percentage value. Since both colony and packed_deque essentially re-use erased-element memory locations, this tests the efficacy of each containers mechanism for doing so (packed_deque's move-and-pop + handle free-list, versus colony's stack + skipfield).

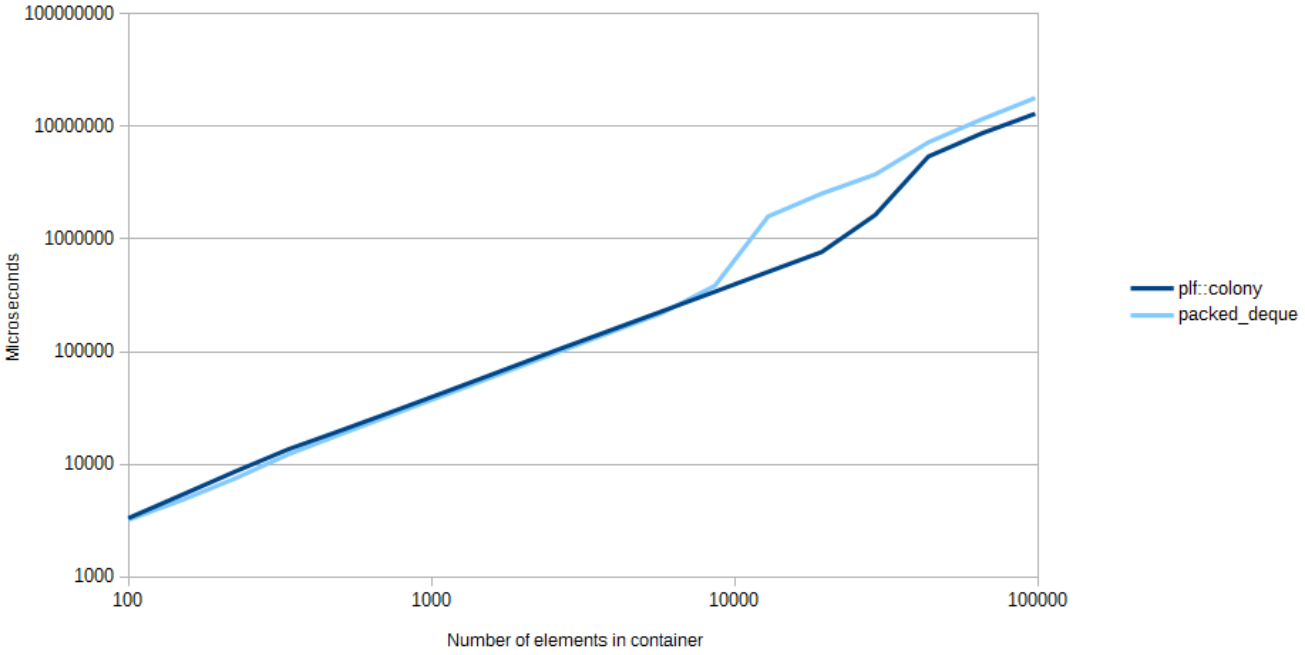
Since neither container will grow substantially in memory usage over time as a result of this process, a longer test length is not necessary like it was for the earlier modification scenario-testing with indexed_vector and pointer_deque. Testing on both plf::colony and plf::packed_deque showed that both of their test results increased linearly according to the number of simulated frames in the test (indexed_vector and pointer_deque have a more exponential growth). Similarly to the modification tests above, we will start with 1% of all elements being erased/inserted per 3600 frames, then 5% and 10%, then move up to 1% of all elements being erased/inserted per-frame, then 5% per-frame, then 10% per-frame.

Performance results

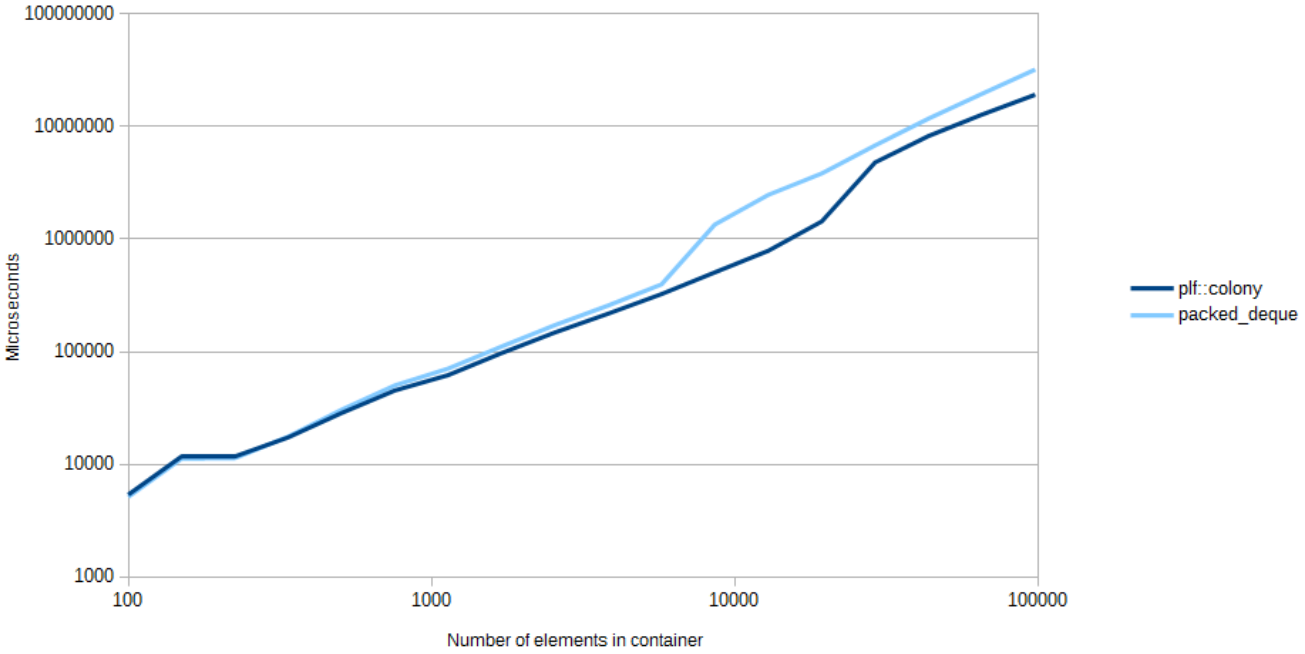


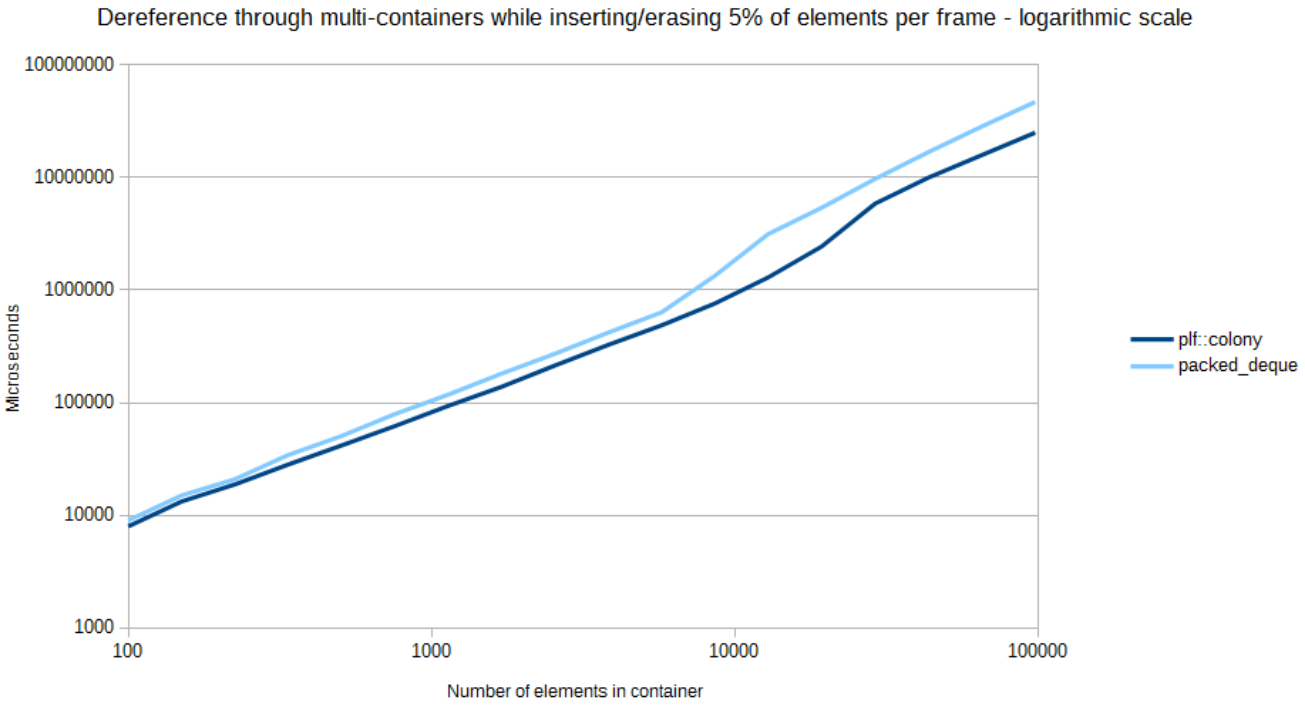


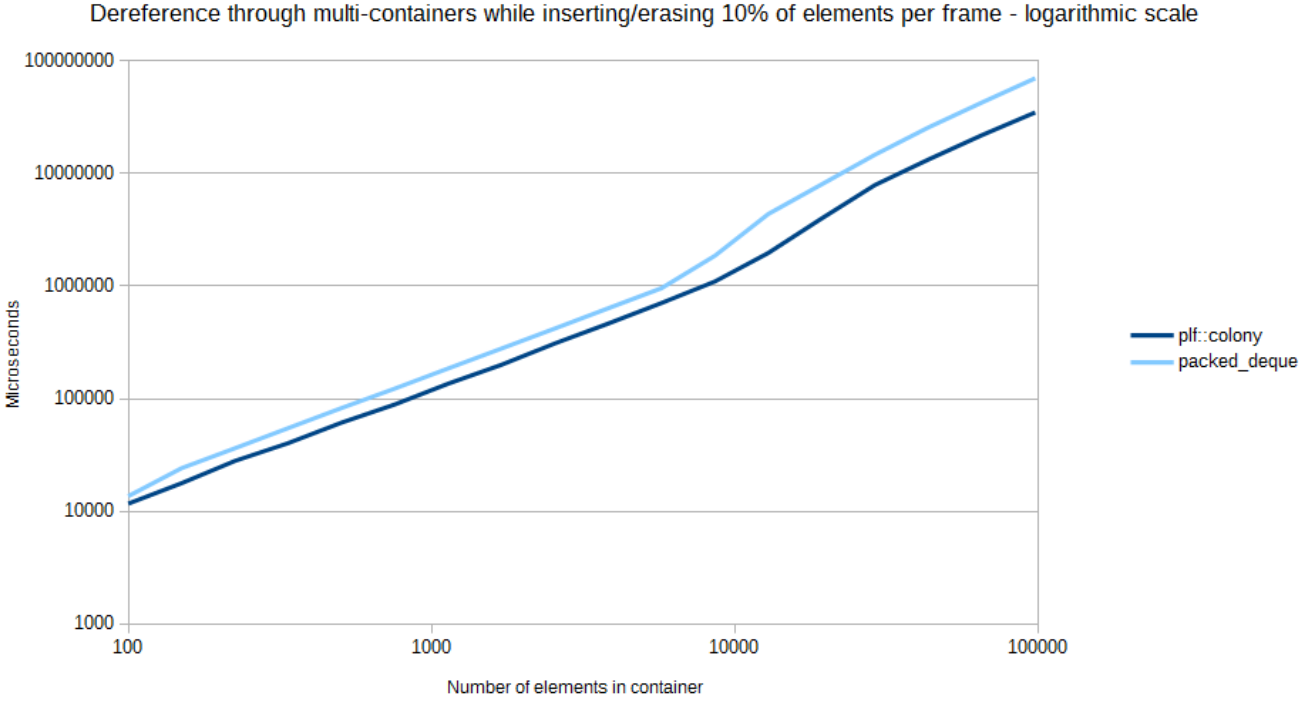
Dereference through multi-containers while inserting/erasing 10% of elements per 3600 frames - logarithmic scale



Dereference through multi-containers while inserting/erasing 1% of elements per frame - logarithmic scale



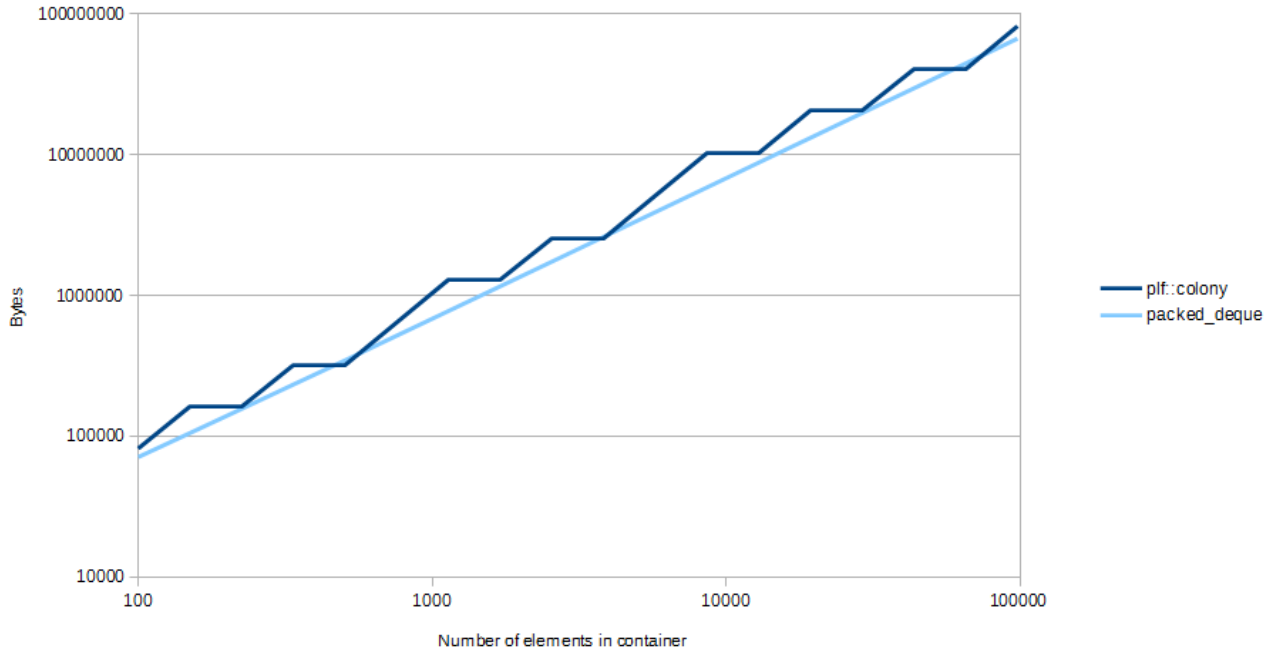




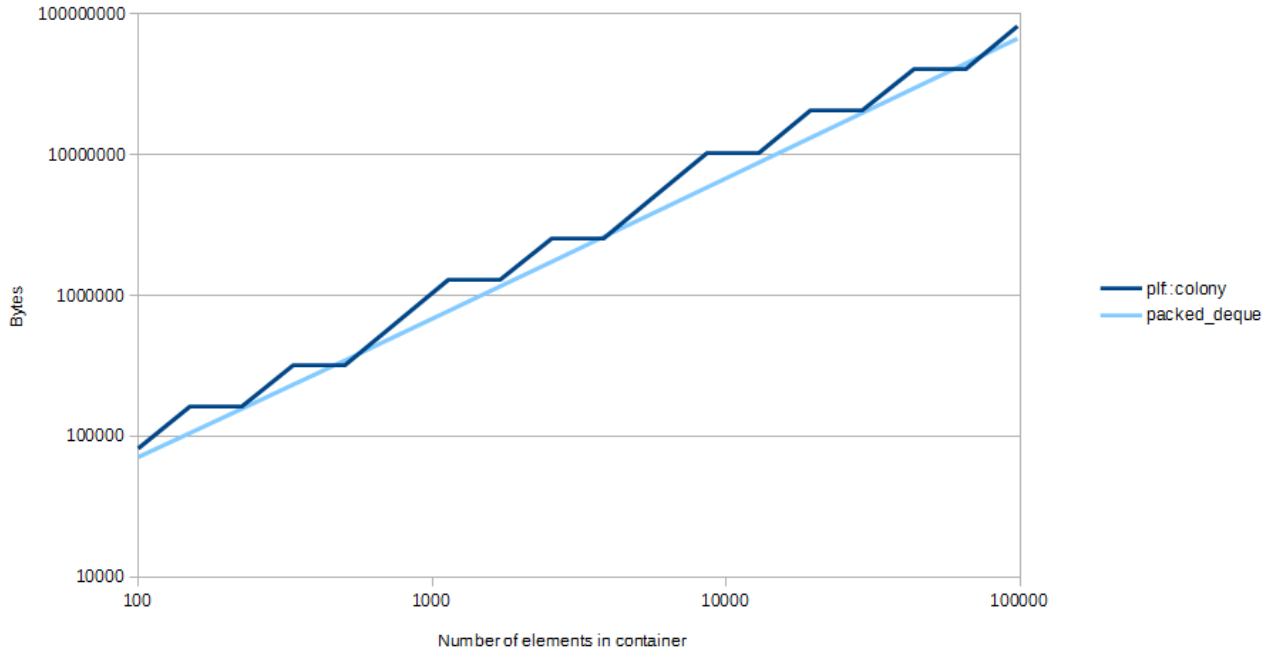
As we can see from the performance results, at low levels of modification packed_deque has a small performance advantage over colony, until about 9000 elements. After 9000 elements, colony has a larger performance advantage. And the higher the level of modification, the fewer elements there have to be in the containers before colony has an advantage. At 1% modification per frame, only 200 elements are needed, while at 5% per-frame and above, colony always has a strong advantage.

Memory results

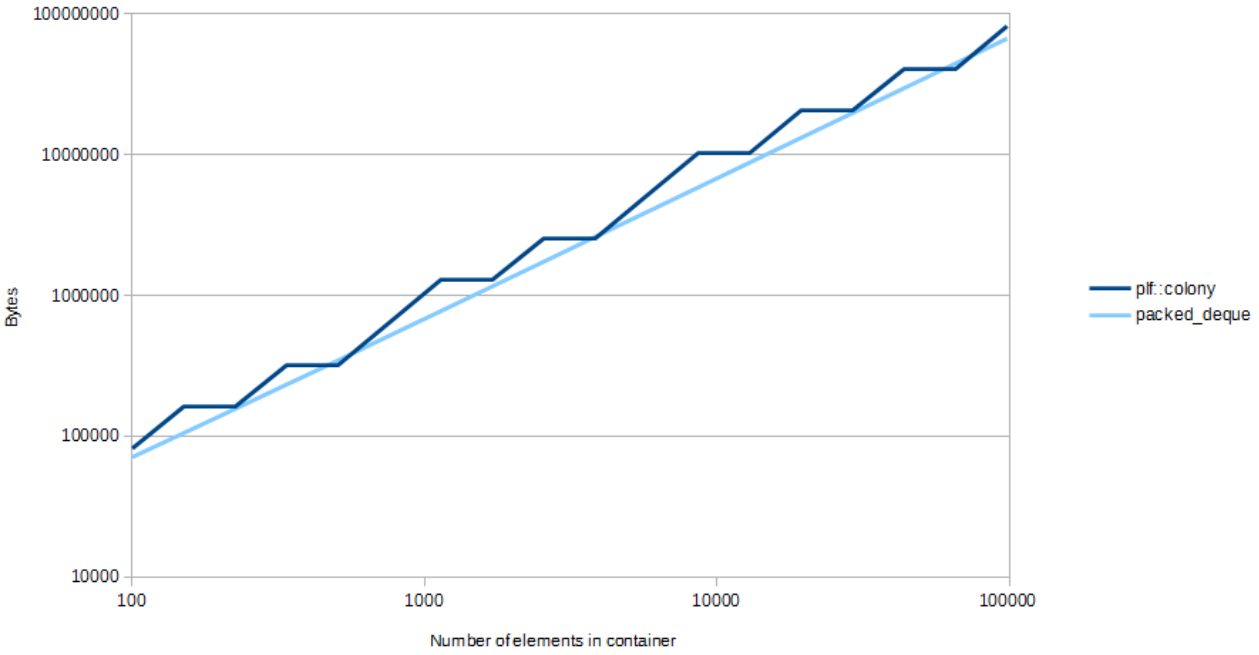
Dereference through multi-containers while inserting/erasing 1% per 3600 frames - approx memory usage - logarithmic scale



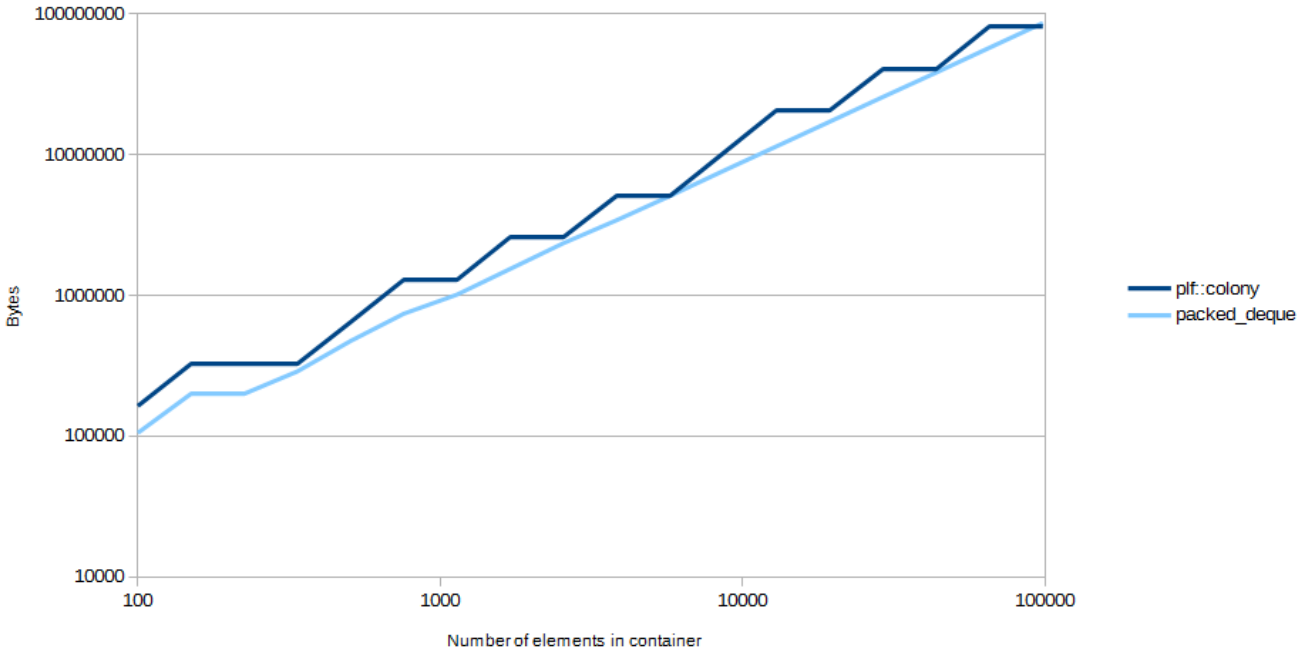
Dereference through multi-containers while inserting/erasing 5% per 3600 frames - approx memory usage - logarithmic scale

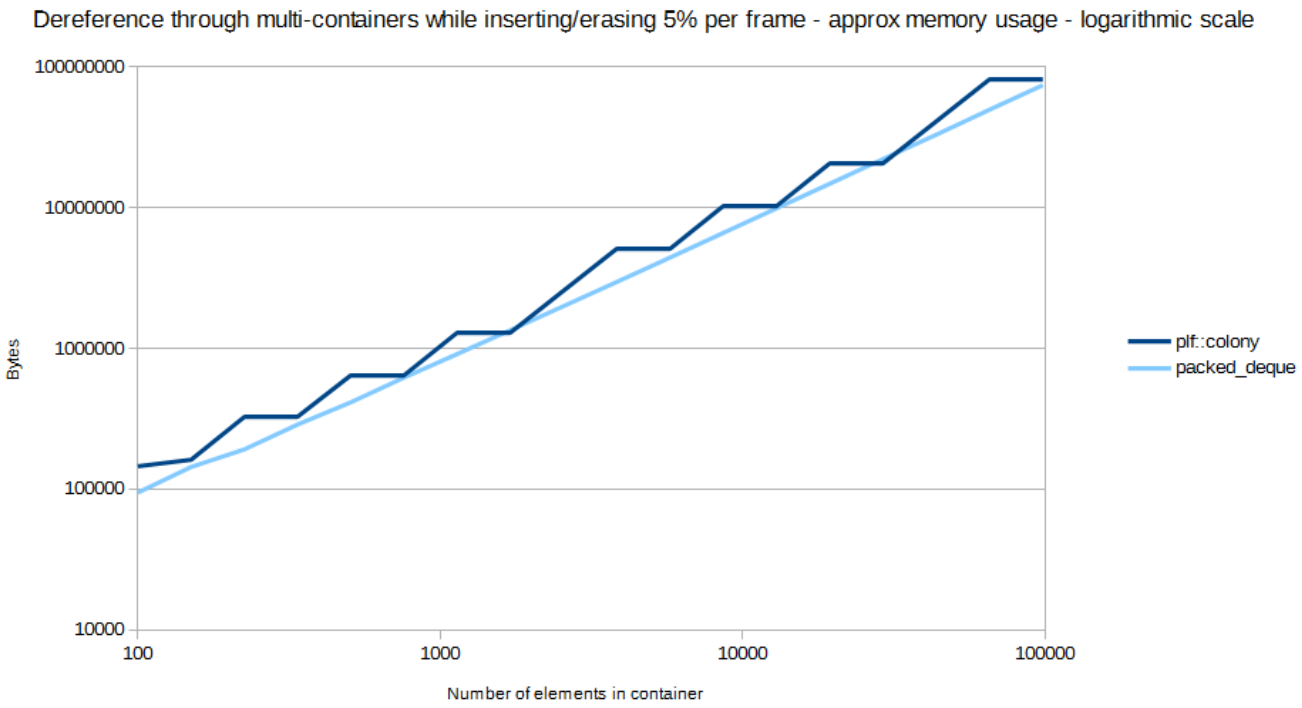


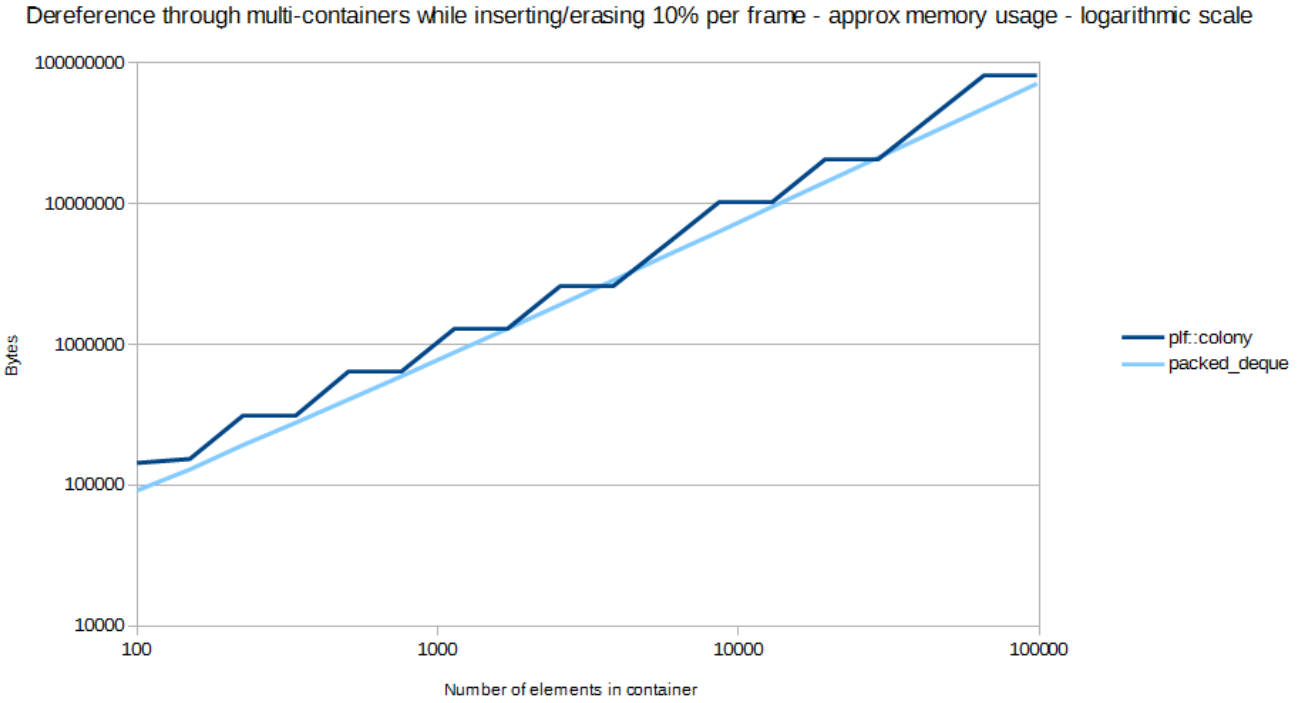
Dereference through multi-containers while inserting/erasing 10% per 3600 frames - approx memory usage - logarithmic scale



Dereference through multi-containers while inserting/erasing 1% per frame - approx memory usage - logarithmic scale







As we can see the memory results don't really change between different levels of modification - packed_deque always has an advantage over colony, though not a huge one. The memory difference can be mitigated somewhat by setting a smaller maximum group size for colony, but this will likely come at an expense of speed.

Overall Performance Conclusions

For situations where unordered, often-inserted/erased content is required, colony provides a convenient solution, while also outperforming the alternatives for the most part. Where modification rates and the number of elements are low, a packed-array-style structure like packed_deque may be your best solution both in terms of performance and memory usage. However once the numbers of elements and rates of modification begin to rise, a colony shines forth. In addition, a packed array will be affected adversely when the type is larger or non-trivially movable, due to the neccessity of moving elements from the back when erasing.

Using a boolean skipfield in combination with a vector or deque is of course a no-go, due to it's poor iteration performance once the number of erasures increases. Similarly, using an erasable iteration field as was used with indexed_vector and

pointer_deque results in both wasteful memory consumption and poor performance once the number of modifications or elements becomes too high.

In short, use a packed-array where both the rate of modification is <= 10% of all elements per 3600 iterations over data and the number of elements is <= 9000, or if external object access to container elements via the dereferencing system is uncommon. In all other areas, for unordered data use a colony.

Appendix C - Frequently Asked Questions

1. **What are some examples of situations where a colony might improve performance?**

   Some ideal situations to use a colony: cellular/atomic simulation, persistent octtree/quadtree, general game entities or destructible-objects in a video game, particle physics, anywhere where objects are being created and destroyed continuously. Also, anywhere where a vector of pointers to dynamically-allocated objects or a std::list would typically end up being used in order to preserve object references but where order is unimportant.

2. **What situations should you explicitly *not* use a colony for?**

   A colony should not be used as a stack, ie. erasing backwards from the back, and then filling, then erasing from the back, etc. In this case you should use a stack ie. plf::stack, std::vector or std::deque. The reason is that erasing backwards sequentially creates the greatest time complexity for skipfield updates, as does reinserting to the start of a sequentially-erased skipblock (which is what stack usage will entail). This effect is mitigated somewhat if an entire group is erased, in which case it is released to the OS and subsequent insertions will not be updating skipfields but simply pushing to back, but you'd still incur the skipfield update cost during erasure.

   In general you should avoid erasing sequentially during reverse iteration except where absolutely neccessary - the skipfield format is optimized for forward-iteration and forward sequential erasure.

3. **What are the time complexities for general operations?**

   Insertion: If no erasures have occured, always O(1) amortised. If erasures have occured, O(random) with the range of the random number being between from O(1) and O(std::numeric_limits<skipfield_type>::max() - 2) (65533 by default unless an alternative skipfield type is specified upon template instantiation. Average time complexity varies based on erasure pattern, but with a random erasure pattern it's closer to O(1) amortized.

   Erase: If no consecutive erasures have occured, or only consecutive erasures prior to the element being erased have occured, O(1) amortised. Otherwise O(n), where n is the number of consecutive previously-erased elements after the element being erased, which will always be between from 1 and std::numeric_limits<skipfield_type>::max() - 2 (65533 by default unless an alternative skipfield type is specified upon template instantiation).
   Assuming ignorance of consecutive erasure placement by the programmer, average time complexity will vary based on erasure patterns and will appear to be O(random). But with a random erasure pattern it will be closer to O(1) amortized.

   std::find: O(n)

   Iterator operations:
   ++ and -- : O(1) amortized
   begin()/end(): O(1)
   advance/next/prev: between O(1) and O(n), depending on current location, end location and state of colony. Average O(log N).

4. **If the time complexities of the insert/erase functions are (kind of) O(random, ranged), why are they still quick?**

   The skipfield for each group is contiguous and separate from the skipfields for other groups, and so fits into the cache easily (unless the skipfield type is changed); thus any changes to it can occur quickly - time complexity is, actually, no indicator of performance on a modern CPU.

   Colony now also uses memmove instead of iterative updates for all but one of the insert/erase operations, which again decreases performance cost. In modern implementations, memmove will typically be implemented in memory chunks, which may reduce further the time complexity. There is one situation in erase which does not use memmove, a rarer case where an element is erased and is surrounded on both sides by consecutive erased skipfield nodes. In this case it isn't actually possible to update the skipfield using memmove because the requisite numbers do not already exist in the skipfield and cannot be copied, so it is implemented as a vectorized update instead. But again

due to a low amount of branching the actual time taken for the update is quite quick, regardless of the number of nodes that need to be updated.

5. **Is it similar to a deque?**

A deque is reasonably dissimilar to a colony - being a double-ended queue, it requires quite a bit of a different internal framework. It typically uses a vector of memory blocks, whereas the colony implementation uses a linked list of memory blocks, essentially. A deque can't technically use a linked list of memory blocks because it will make some random_access iterator operations (eg. + operator) non-O(1). In addition, being a double-ended queue makes having a growth factor for memory blocks problematic because the rules for growth at each end of the queue become difficult to implement in a way which increase performance. And in fact, if you were to be remove memory blocks from within a deque, you could not utilize a growth factor because this would make the "+" iterator operator impossible to implement in O(1) time (as you would not be able to ascertain the size for each memory block without inspecting them all).

A deque and colony have no comparable performance characteristics except for insertion (for a decent deque implementation). Deque erasure performance varies wildly depending on the implementation compared to std::vector, but is generally similar to vector erasure performance. A deque also invalidates pointers to subsequent container elements when erasing elements, which a colony does not.

6. **What are the thread-safe guarantees?**

Unlike a std::vector, a colony can be read from and written to at the same time, however it cannot be iterated over and written to at the same time. If we look at a (non-concurrent implementation of) std::vector's threadsafe matrix, to see which basic operations can occur at the same time, it reads as follows (please note push_back() is the same as insertion in this regard):

std::vector	Insertion	Erasure	Iteration	Read
Insertion	No	No	No	No
Erasure	No	No	No	No
Iteration	No	No	Yes	Yes
Read	No	No	Yes	Yes

In other words, multiple reads and iterations over iterators can happen simultaneously, but the invalidation caused by insertion/push_back and erasure means those operations cannot occur at the same time as anything else.

Colony on the other hand does not invalidate pointers/iterators to non-erased elements during insertion and erasure, resulting in the following matrix:

plf::colony	Insertion	Erasure	Iteration	Read
Insertion	No	No	No	Yes
Erasure	No	No	No	Mostly
Iteration	No	No	Yes	Yes
Read	Yes	Mostly	Yes	Yes

In other words, reads may occur at the same time as insertions and erasures (provided that the element being erased is not the element being read), multiple reads and iterations may occur at the same time, but iterations may not occur at the same time as an erasure or insertion, as either of these may change the state of the skipfield which's being iterated over. Erasures will not invalidate iterators unless the iterator points to the erased element. Note that iterators pointing to end() may be invalidated by insertion.

So, colony could be considered more inherently threadsafe than a (non-concurrent implementation of) std::vector, but still has some areas which would require mutexes or atomics to navigate in a multithreaded environment.

7. **Any pitfalls to watch out for?**

    1. Because erased-element memory locations will be reused by `insert()` and `emplace()`, insertion position is essentially random unless no erasures have been made, or an equal number of erasures and insertions have been made.
    2. For architectural reasons, reserve can only reserve a number of elements up to the maximum bit-depth of the skipfield type (65535 unless an alternative type is specified in the constructor).

8. **Am I better off storing iterators or pointers to colony elements?**

   My testing so far indicates that storing pointers and then using `get_iterator_from_pointer()` when or if you need to do an erase operation on the element being pointed to, is more performant than storing iterators and performing erase directly on the iterator.

9. **Any special-case uses?**

   In the special case where many, many elements are being continually erased/inserted in realtime, you might want to experiment with limiting the size of your internal memory groups in the constructor. The form of this is as follows:
   ```
   plf::vector<object> a_vector;
   a_vector.change_group_sizes(500, 5000);
   ```
   where the first number is the minimum size of the internal memory groups and the second is the maximum size. Note these can be the same size, resulting in an unchanging group size for the lifetime of the colony (unless `change_group_sizes` is called again or operator = is called). One reason to do this is that it is slightly slower to pop an element location off the internal memory position recycling stack, than it is to insert a new element to the end of the colony (the default behaviour when there are no previously-erased elements). If there are any erased elements in the colony, the colony will recycle those memory locations, unless the entire group is empty, at which point it is freed to memory. So if a group size is large and many, many erasures occur but the group is not completely emptied, (a) the number of erased element locations in the recycling stack could get very large, resulting in a detriment to performance and (b) iteration performance will suffer. In that scenario you may want to run a benchmark limiting the minimum/maximum sizes of the groups, and tune it until you find optimal usage.

   Please note that the the fill, range and initializer-list constructors can also take group size parameters, making it possible to construct filled colonies using custom group sizes.

10. **Exception Guarantees?**

    All operations which allocate memory have strong exception guarantees and will roll back if an exception occurs, except for operator = which has a basic exception guarantee (see below). For colony, iterators are bounded by asserts in debug mode, but unbounded in release mode, so it is possible for an incorrect use of an iterator (iterating past end(), for example) to trigger an out-of-bounds memory exception. These are the only two areas where exceptions can occur.

    The reason why operator = only has a basic guarantee is they do not utilize the copy-swap idiom, as the copy-swap idiom significantly increases the chance of any exception occuring - this is because the most common way an exception tends to occur during a copy operation is due to a lack of memory space, and the copy-swap idiom doubles the memory requirement for a copy operation by constructing the copy before destructing the original data. This is doubly innappropriate in game development, which colony has been initially for, where memory constraints are often critical and the runtime lag from memory spikes can cause detrimental game performance. So in the case of colony if a non-memory-allocation-based exception occurs during copy, the = operators will have already destructed their data, so the containers will be empty and cannot roll back - hence they have a basic guarantee, not a strong guarantee.

11. **Why must groups be removed when empty?**

    Two reasons:
    1. If groups aren't removed then iterator ++ and -- operations become non-O(1) for time complexity, which makes them illegal according to the C++ standard. At the moment they are O(1) amortised, typically one update for both skipfield and element pointers, but two if a skipfield jump takes the iterator beyond the bounds of the current group and into the next group.
    If there are multiple empty groups in a row, ++ iteration essentially becomes O(random) because each group will require an op to check for emptiness and to jump to the next. Essentially you get the same scenario as you do with a boolean skipfield.
    2. Performance. Iterating over empty groups is slower than them not being present, and pushing to the back of the colony is faster than recycling memory locations. When a group is removed it's recyclable memory locations are also removed from the stack, hence subsequent insertions are also faster.

12. **Group sizes - what are they based on, how do they expand, etc**

    Group sizes start from the minimum size defined either by the default (8 elements, or larger if the type stored is small) or by the programmer (with a minimum of 3 elements). Subsequent group sizes then increase the *total capacity* of the colony by a factor of 2 (so, 1st group 8 elements, 2nd 8 elements, 3rd 16 elements, 4th 32 elements etcetera) until the maximum group size is reached. The default maximum group size is the maximum possible number that the skipfield bitdepth is capable of representing (std::numeric_limits<skipfield_type>::max()). By default the skipfield bitdepth is 16 so the maximum size of a group is 65535 elements. However the skipfield bitdepth is also a template parameter which can be set to any unsigned integer - unsigned char, unsigned int, Uint_64, etc. Unsigned short (guaranteed to be at least 16 bit, equivalent to C++11's uint_least16_t type) was found to have the best performance in real-world testing due to the balance between memory contiguousness, memory waste and the restriction on skipfield update time complexity. Initially the design also fitted the use-case of gaming better (as games tend to utilize lower numbers of elements than some other fields), as that was the primary development field at the time.