slot_map Container in C++

Reply-to: Allan Deutsch <allan.d@digipen.edu>

Audience: LEWG, SG14

June 12, 2017

Document Number: P0661

Abstract

This is a proposal to add a high performance associative container to the C++ STL. It proposes a non-breaking library addition that can be implemented with current language and library features included in the standard.

Table of Contents

I. Introduction

Current Associative Containers

The current options in the STL for associative containers all require the user to provide both the key and value. There are times when a user will want to store a collection of distinctly identifiable elements and not care what the key to access them is, with the condition that the keys still be unique.

The most common approach to this is using a hash map and a key generation algorithm or hashing function. These methods have varying execution times but often require a lot when used for many elements, which is compounded by the poor cache utilization of the node-based std::unordered_map.

Another approach to solving this is using vector, since the element index is a unique identifier. Issues can arise from this when a user wishes to remove an element not at the end of the container, but must modify the location and index of other element(s) to do so.

Other Containers Considerations

One common challenge in game development is that the application is on a very tight schedule. The entire simulation must be updated many times per second. Frames with inconsistent durations create a jarring experience for the end user. For this reason, it is desirable to have consistent low costs, as opposed to amortized solutions offering better best cases and worse worst cases. While these solutions may provide better overall performance, the inconsistency is undesirable in games.

Slot-Based Containers

A consistent high performance associative container is something that would greatly benefit games and other real-time applications. This proposal is for a slot-based associative container which is capable of reusing indices (slots) without requiring that elements be shifted in order to achieve O(1) insert and erase operations. There are several ways to implement this, and these variations are discussed in the "Design Decisions" section. One approach is described in the C++Now 2017 lightning talk on slot maps, which is suggested as a quick introduction to slot maps.

III. Motivation and Scope

Why is it important?

In games, there are many objects which need to be identified uniquely. An example of this is an entity-component-system architecture. Each game object is a composition of various types of components such as a rigid body and transform data. These components are composed together in an entity to add functionality without creating complex inheritance hierarchies. It is common for each type of component to be iterated over by a system which performs transformations based on the data stored in them such as updating positions or rendering them to the display. Data locality is very important for performance, resulting in various techniques to keep these components in contiguous memory.

This data also needs to be uniquely identifiable somehow so it can be associated with the correct entity, ideally in memory which can be reused safely to prevent the overhead of frequent allocations. It also needs to avoid situations in which a valid reference to a slot exists, but the slot is cleared and filled with a new element. Since that reference is no longer valid it shouldn't be usable.

What kinds of problems does it address?

This container addresses several problems present in other containers; inconsistent costs for common operations, unstable element location, poor cache utilization, and high overhead for find (in the case of hashing). It also solves the ABA problem of an element being deleted and replaced with an identical element by using a generation counter for each slot.

The container has consistent O(1) cost for the following operations: inserting or emplacing an element when a reallocation is not required, releasing an existing element, and accessing an element using a key, pointer, or iterator. It also provides the user lightweight unique keys for all their elements, something not currently available in any other standard containers.

What is the intended user community?

This container is meant for use in application domains where performance is critical – the target audience is the same as that of SG14. People who have expressed interest in this proposal so far typically come from one of the following industries:

- video games
- high frequency trading
- scientific computing
- kernel/OS

What level of programmer is it intended to support?

As a container, I believe it should be accessible to anyone who understands how associative containers work. The ideal interface and behavior will be as close to those of other containers in the STL as possible.

Existing Uses

This container is very popular and well known among AAA game studios, however there is not yet a standardized name or implementation. The primary differences between implementations are fixed vs dynamic size and whether they use a layer of indirection. The indirection layer allows the physical memory location of elements to move, allowing the implementation to keep them packed together in memory. It is also common to use hash map variants optimized for lookup times.

List of Existing Implementations

- http://bitsquid.blogspot.com/2011/09/managing-decoupling-part-4-id-lookup.html - packed array
- http://blog.molecular-matters.com/2013/07/24/adventures-in-data-oriented-design-part-3c-external-references/ - unnamed
- http://seanmiddleditch.com/data-structures-for-game-developers-the-slot-map/ - slot map
- https://gist.github.com/Masstronaut/e96b62bf2b3dffc61af6 - slot array

Note that this list is not exhaustive, and that more implementations exist within the codebases of companies involved in the development in games. A similar technique is used in the object pool allocation strategy.

IV. Impact On the Standard

Dependencies

The container is a pure extension, but would require specializations for many of the standard algorithms. It can be implemented without any modifications to existing language or library features.

V. Design Decisions

Fixed- vs. Dynamic-size

There are two prevailing options for how to implement the dynamic variation. One is using dynamically sized array storage, which can store elements densely at the cost of a worst-case O(N) insertion when reallocation is required. The other option is to implement it as a dynamic array of fixed-size arrays. It guarantees O(1) insert and emplace operations even when an allocation is required, but the elements will be stored across multiple disjointed blocks of memory, not a single contiguous one.

Alternatively a fixed-size implementation can guarantee O(1) for all cases of insert, remove, and find, with the downside of not being able to expand if necessary.

The suggestion for this decision is to use a vector-like approach which offers the best cache utilization for iterating over the entire container, and supports varying sizes. Dynamic sizing is suggested both for similarity to other containers and the flexibility it offers to users.

Indirection and Compacting vs. Stable Indices and Raw Pointers

This decision is most relevant in a fixed-size or deque-like variant which could potentially guarantee stable addresses for elements by storing the generation counter alongside the elements, with the index part of keys referencing the element index and foregoing the indirection layer completely. Instead of dense storage the elements would be stored sparsely, staying in the same place in memory for their entire lifetime. This has costs during element iteration, but has better lookup performance.

Alternatively, it could have a layer of indirection between keys and data. This allows keys to reference a stable slot which in turn references an actual container element. While the lookup overhead is higher, it allows the container to guarantee elements are densely contiguous by moving the element in the back of the container into the position of an element being removed. This approach offers better cache coherency and with an array-like backing could provide a .data() member function. These upsides come at the cost of the extra indirection on key lookup, unstable indices for elements, and a move/assign/copy operation being used on erase.

An approach using indirection and compacting is suggested. This configuration offers a good mixture of iteration and lookup performance.

Copy, Assign, and Move

For a slot map implemented with stable indices, an optimization could be performed on top of copy/assign/move operations by positioning all the elements densely at the front of the lvalue container, which has the downside of invalidating all keys to the rvalue. That is not an acceptable side effect in most usage scenarios so it is suggested that this not be allowed by requiring keys still work on the resulting container instance, much like the keys in unordered_* containers.

Container vs. Adapter

Slot map could be added as a standalone container, following the specific design decisions suggested in this proposal or those deemed best by the committee. It could also be added as a container adapter, which allows the users to make those design decisions for themselves by specifying the underlying container.

One issue raised about current standard containers is that while they serve their general purpose effectively, many of the guarantees they make force certain implementation details or design decisions with poor performance. Rather than attempting to tackle those directly to appeal to developers working on truly performance critical systems, an adapter approach offloads those decisions to the developer. If they have a compatible container implementation with optimizations and guarantees best suited to their use case, they can use that as the underlying storage type.

For these reasons, it is suggested that slot map be added as a container adapter. To address the usability issues this causes, a template parameter defaulted to std::vector is suggested, as that best resembles the suggested designs in other sections.

Reference Keys

The reference keys are composed of 2 parts:

- An element index – this is the index in the underlying array at which the referenced element is stored
- A counter – represents the generation of the element in that slot. This enables slots to be reused safely without old reference keys being able to access them.

The reference key is used as a bit field with the following layout:

- N bits – minimum number of bits which can be used to identify all indices required by the element count template parameter.
- sizeof(key_type)-N bits – counter

Allowing user-specified key types is encouraged for flexibility and consistency with other standard associative containers, with the limiting characteristic being that the key type must cause std::is_integral to be true. Additionally, a non-type template parameter should exist for users to specify the max element count supported by the container. By doing this, the users can configure the key type and how many of its bits are used for indexing and counting. It also means that the member type alias size_type can shrink to fit the container.

Free List Implementation

The in-place free list can be implemented with only a head index, which needs size_type bytes and is able to provide O(1) allocations. Alternatively, the free list could store both a front and back index on the container, providing the ability to cycle through free slots more uniformly, reducing the chance of a generation counter overflow occurring. This is a quite valuable benefit with low overhead, so explicitly requiring it seems worthwhile.