If I Had My 'Druthers: A Proposal for Improving the Containers in C++2x

Bob Steagall

C++Now 2018

If I Had My 'Druthers:
Some Thoughts on Improving the Containers in C++2x

Bob Steagall
C++Now 2018

If I Had My 'Druthers:
Some Thoughts on Improving the Containers in C++2x

Sponsored by: The American East Const Association of America®

Bob Steagall
C++Now 2018

Cognitive Dissonance

- From Wikipedia:

  In the field of psychology, *cognitive dissonance* is the **MENTAL DISCOMFORT** (psychological stress) experienced by a person who simultaneously holds two or more contradictory beliefs, ideas, or values.

- To wit:
  - I **love** the standard containers
  - I **hate** the standard containers

Goals

- Present some thoughts and ideas
- Provoke some discussion
- Perhaps win some supporters for the cause
- Escape the room relatively unharmed

A Quick History

- 1979, Alexander Stepanov begins working on generic programming (GP)
- 1983, Ada became the first to provide GP support, followed by Eiffel in 1985
- 1987, Stepanov and David Musser published an Ada library for generic list processing
- 1992, Meng Lee joins Stepanov at HP Research Labs, where team is experimenting with C and C++
- 1993, Stepanov presents the main ideas at the November meeting of the ANSI/ISO C++ Standardization Committee
- 1994, Stepanov and Lee produce a draft proposal for the March committee meeting

A Quick History

- 1994, final proposal accepted at the July committee meeting
- 1994, freely available implementation published by HP in August
- 1994-1998, much additional work, including adding the associative containers
- 1998, first ISO C++ Standard published in September
- 2003, first update to the Standard
- 2007, establishment of `tr1`, which included several new containers
- 2011, C++11 is born, which moved new containers into `std`

On the Brilliance of the STL

- Four important positive qualities
  - Speed
  - Efficiency
  - Extensibility
  - Elegance
- Separates data structures from algorithms, and tie them together with iterators
  - It is remarkable what is accomplished with only 5 iterator categories
- The underlying ideas have become embedded into our way of thinking

Making Improvements?

Guiding Principles

- Correctness
- Performance
- Conceptual integrity
- Mnemonic integrity
- Readability
- Understandability

What Could be Improved

- Container names
- Container selection
- Public APIs
- Allocators
- Various minor nits
- Underlying implementations – concrete vs. abstract
- Understandability

Requirements

- I see three categories of users that have different expectations
- **Casual users** want maximum ease of use, minimal learning curve, good experience and immediate productivity right out of the box
- **Power users** want ease of use combined with some customizability in search of higher performance
- **Expert users** want maximum customizability to solve highly specialized problems and/or highest performance
- The containers should provide levels of interface appropriate to each set of expectations

Meaningful Names – Let's Call Things What They Are

- A rose by any other name **does not** smell as sweet…
- Names are important!
- Well-chosen names…
  - Provide context
  - Denote relationships
  - Promote understanding
- `std2` may well be dead, so any design for new containers needs to pick non-conflicting names in `std`
  - Jacksonville straw poll, consensus against `std2` in LEWG

What concrete containers should be provided?

- multi-dimensional dynamic array
  - with partial specialization for 1D
- multi-dimensional fixed-size array
  - with partial specialization for 1D
- double-ended dynamic array
- singly-linked list
- doubly-linked list
- binary tree
- binary search tree
- red-black tree
- AVL tree
- radix tree (ART)
- chained hash table
- linearly probed hash table
- graph?
- matrix
- row vector / column vector

What abstract container adaptors should be provided?

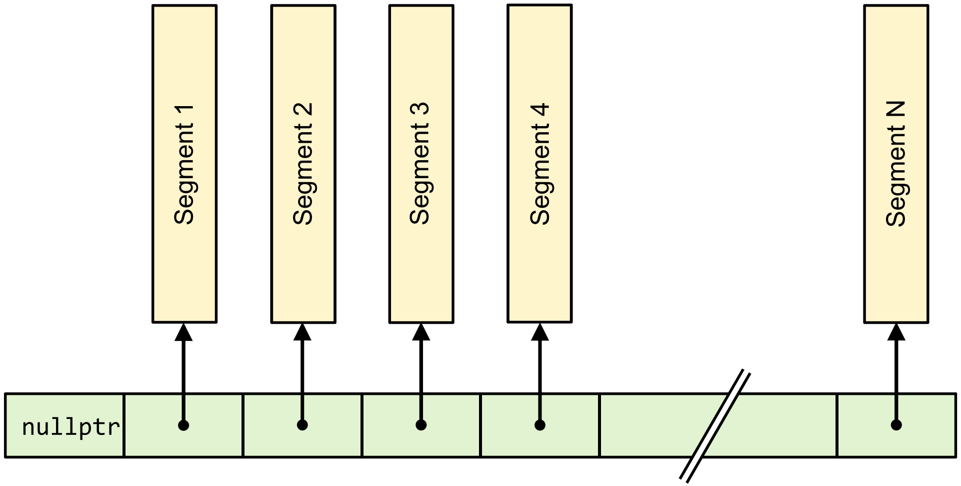
- stack
- queue
- double-ended queue
- binary heap
- Fibonacci heap
- ordered set / multiset
- ordered map / multimap
- hashed set / multiset
- hashed map / multimap
- graph?

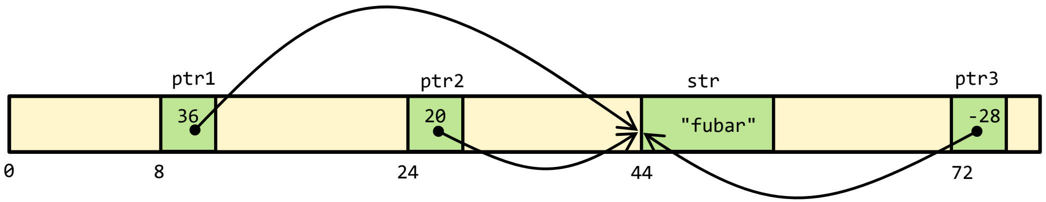
Shape – A Fundamental Container Property

- Containers have a shape
  - The nature of that shape is a fundamental property of that container’s type
  - An abstract shape is described by source code
  - In a way, the abstract shape analogous to a template
- We give a container guidance in how to form its runtime shape by providing it with template arguments
  - I think of these as policies giving the source code “marching orders”
- The container’s runtime shape may also depend on
  - The value of its elements
  - The order of insertion
  - Or both

Shape – A Fundamental Container Property

- The implementation and management of a container's runtime shape is accomplished through the addressing model, which defines
  - The bits used to represent an address
  - How an address is computed from those bits
  - How memory from a fundamental memory resource (storage model) is arranged





Addressing Model – Also a Fundamental Container Property

- The implementation and management of a container’s runtime shape is accomplished through the addressing model, which defines
  - The bits used to represent an address
  - How an address is computed from those bits
  - How memory from a fundamental memory provider is arranged
- I contend that a container’s addressing model is a fundamental property that is as critical to a container’s operation runtime as its source and its template arguments

Shape – A Fundamental Container Property

- The runtime shape depends on:
  - The container source code, which describes a sort of Platonic Ideal
  - The template arguments, which specifies how to instantiate that ideal
  - The addressing model, which provides access to memory so the instance can be built
  - The data, which is laid out in memory in accordance with the above
- Thinking in these terms,
  - The addressing model is a fundamental property of a container
  - A traditional STL allocator is not

One Possible Idiom

- Factor containers into levels
  - Level 0: an engine type that performs manipulations that depend only on the addressing model
  - Level 1: an engine type employing the corresponding level 0 engine, and which performs work depending on the allocator/heap type
    - Two variants/partial specializations: one for stateless heaps and one for stateful heaps
  - Level 2: a fully featured container similar to today's, with similar customizability
  - Level 3: a "basic" container whose type signature does not contain the allocator and uses the global heap
- Introduce a new nested namespace for levels 0, 1, 2
  - Level 3 goes in `std`

Example: Doubly-Linked List

```
namespace std::xci {

    struct global_stateless_heap
    {
        using propagate_on_container_move_assignment = true_type;
        using is_always_equal                        = true_type;

        using void_pointer = void*;

        template<class T>
        auto    allocate(size_t N) -> std::tuple<void_pointer, size_t>;
        void    deallocate(void_pointer);
    };
}
```

Example: Doubly-Linked List

```
namespace std::xci {

    struct polymorphic_heap
    {
        std::pmr::memory_resource*    mp_resource;

        using is_always_equal = false_type;
        using void_pointer    = void*;

        template<class T>
        auto    allocate(size_t N) -> std::tuple<void_pointer, size_t>;
        void    deallocate(void_pointer);
   };
}
```

Example: Doubly-Linked List

```
namespace std::xci {

    template<class T, class VP>
    struct doubly_linked_list_engine
    {
        struct list_node;
        struct data_node;

        using value_type        = T;
        using list_node_pointer = typename std::pointer_traits<VP>::template rebind<list_node>;
        using data_node_pointer = typename std::pointer_traits<VP>::template rebind<data_node>;

        struct list_node { list_node_pointer m_prev, m_next; };
        struct data_node : public list_node { value_type m_data; };

        list_node   m_sentinel;

        void    push_back(data_node_pointer);
        void    reverse();
        void    swap(doubly_linked_list_engine_base&);
    };
}
```

Example: Doubly-Linked List

```
namespace std::xci {

    template<class T, class HEAP>
    struct slh_doubly_linked_list_engine
        : public doubly_linked_list_engine<T, typename HEAP::void_pointer>
    {
        void    clear();
        void    push_back(T const& t) { push_back(make_node(t)); }

        data_node_pointer   make_node(T const& t);    //- uses HEAP::allocate<T>()
    };

}
```

Example: Doubly-Linked List

```
namespace std::xci {

    template<class T, class HEAP>
    struct sfh_doubly_linked_list_engine
        : public doubly_linked_list_engine<T, typename HEAP::void_pointer>
    {
        void    clear();
        void    push_back(T const& t) { push_back(make_node(t)); }

        data_node_pointer   make_node(T const& t);    //- uses m_heap.allocate<T>()

        HEAP    m_heap;
    };
}
```

Example: Doubly-Linked List

```
namespace std::xci {

    template<class T, class HEAP=polymorphic_heap>
    struct doubly_linked_list
    {
        using engine_type = conditional_t<HEAP::is_always_equal::value,
                                          slh_doubly_linked_list_engine<T, HEAP>,
                                          sfh_doubly_linked_list_engine<T, HEAP>>;

        engine_type     m_engine;

        void    clear()   { m_engine.clear(); }
        void    reverse() { m_engine.reverse(); }

        void    push_back(T const& t) { m_engine.push_back(t); }
    };
}
```

Example: Doubly-Linked List

```
namespace std {

    template<class T>
    struct doubly_linked_list
    {
        using engine_type = xci::slh_doubly_linked_list_engine<T, xci::global_stateless_heap>;

        engine_type     m_engine;

        void    clear()   { m_engine.clear(); }
        void    reverse() { m_engine.reverse(); }

        void    push_back(T const& t) { m_engine.push_back(t); }
    };
}
```

Example: Doubly-Linked List

```
struct my_custom_heap
{
    using is_always_equal = std::false_type;
    using void_pointer    = void*;

    template<class T>
    auto    allocate(size_t N) -> std::tuple<void_pointer, size_t>;
    void    deallocate(void_pointer);
};

void f1()
{
    doubly_linked_list<std::string>                     l1;  //- uses global stateless heap
    xci::doubly_linked_list<std::string>                l2;  //- uses polymorphic heap
    xci::doubly_linked_list<std::string, my_custom_heap>  l3;  //- uses custom heap

    l1.reverse();
    l2.clear();
    l3.reverse();
}
```

For Example

```
namespace std::xci
{
    template<class T, class HEAP=polymorphic_heap> class doubly_linked_list;

    template<class T, size_t N, class HEAP=polymorphic_heap> class dynamic_array;
    template<class T, class HEAP=polymorphic_heap>           class dynamic_array<T,1,HEAP>;

    template<class T, class CMP=less<>, class HEAP=polymorphic_heap> class red_black_tree;

}

namespace std
{
    template<class T> class doubly_linked_list;

    template<class T, size_t N> class dynamic_array;
    template<class T>           class dynamic_array<T,1>;

    template<class T, class CMP=less<>> class red_black_tree;

    template<class T, class CMP=less<>, class IMP=red_black_tree<T,CMP>> class ordered_set;
}
```

Possible Benefits

- The `std` and `std::xci` containers of a given type are different types
  - The `std::xci` version builds upon the `std` version interface without using inheritance
  - To support container customization around allocation
  - To possibly add other expert customization points
  - The layered engine approach provides for wise maximal code re-use
- Three layers of customizability for three categories of user
- Provides the right level of complexity for each category of user

Discussion / Questions

Thank You for Attending!

Talk: https://github.com/BobSteagall/CppNow2018

Blog: https://bobsteagall.com