Customizing the Standard Containers

Marshall Clow

Qualcomm Technology, Inc.

29-Sep-2017

The Standard Containers

1. `array`
2. `vector`
3. `deque`
4. `list`
5. `forward_list`
6. `set/multiset`
7. `map/multimap`
8. `unordered_set/multiset`
9. `unordered_map/multimap`
10. `string` (kind of)
11. Container adapters (`stack`, `queue` and `priority_queue`)

Customization Methods

Each of the containers have helper classes that provide some functionality. These classes/objects have sensible defaults, but you can provide your own to bend these containers to your will.

1. Allocators
2. Comparisons
3. Hashing

Allocators

Allocator Talks this week

1. Monday: Alisdair Meredith - An allocator model for std2
2. Monday: Monday: Sergey Zubkov - From security to performance to GPU programming: exploring modern allocators
3. Tuesday: John Lakos - Local ('Arena') Memory Allocators (2 parts)
4. Thursday: Bob Steagall - How to write a custom Allocator
5. Friday: Pablo Halpern - Modern Allocators: The Good Parts

How do I use an allocator?

```
template <
    typename T,
    typename Alloc = std::allocator<T>>
class vector;
```

What does an allocator do?

1. It allocates and deallocates memory.
2. (optionally) It constructs and destructs objects.

Allocating memory
a simple implementation

```
T* allocate(std::size_t n)
{ return static_cast<T*>(
                ::operator new(n*sizeof(T)));}

void deallocate(T* p, std::size_t) noexcept
{ ::operator delete(static_cast<void*>(p));}
```

What can you do with this?

1. You can manage your own memory pool.
2. You can allocate in shared memory.
3. You can enforce alignment.

Creating objects in allocated memory
a simple implementation

```
// Create an object at p
template<class U, class... Args>
void construct(U* p, Args&&... args)
{::new ((void*)p) U(std::forward<Args>(args)...);}


// Destroy the object at p
template <class U>
void destroy(U* p)
{p->~U();}
```

What can you do with this?

1. You can register/log every object creation.
2. You can add parameters to the constructor call.
3. You can give each object a unique tag.

Using your own allocator

```
vector<int> v1;
vector<int, alloc<int>> v2;

// Assume alloc has a ctor that takes three ints
using myVector = vector<int, alloc<int>>;
myVector v3;
myVector v4{alloc<int>(1,2,3)};

alloc<int> a1{2,9,4};
myVector v5{a1};
```

Allocators and containers

1. Non-allocator aware containers - `array`
2. Contigous containers - `vector`, `string` and `deque`
3. Node based containers - `list` and `forward_list`
4. Associative containers - `set/multiset` and `map/multimap`
5. Unordered associative containers - `unordered_set/multiset` and `unordered_map/multimap`

Associative Containers

The associative containers (set/multiset and map/multimap) take three (or four, for map) template parameters.

```
template <
    typename Key,
    typename Comp = std::less<Key>,
    typename Alloc = std::allocator<Key>>
class set;

template <
    typename Key,
    typename T,
    typename Compare = std::less<Key>,
    typename Alloc =
        std::allocator<std::pair<const Key, T>>>
class map;
```

What do comparison functions look like?

They are something that you can call using the function call syntax.

They return something that can be converted to a boolean.

```
struct PersonLess {
   bool operator() (const Person& rhs,
                    const Person& lhs) const
   { return lhs.lastname < rhs.lastname; }
   };
```

Using your own comparison functions

```
set<Person> s1;  // will not compile
set<Person, PersonLess> s2;

using mySet = set<Person, PersonLess>;
mySet v3;
// This is really:
//  set<Person, PersonLess, allocator<Person>>
//     v3(PersonLess(), allocator<Person>());

mySet v4{PersonLess()}; // explicitly passing
```

Unordered Associative Containers

Unordered Associative Containers

```
template <
    typename Key,
    typename Hash = std::hash<Key>,
    typename Pred = std::equal_to<Key>,
    typename Alloc = std::allocator<Key>>
class unordered_set;

template <
    typename Key,
    typename T,
    typename Hash  = std::hash<Key>,
    typename Pred  = std::equal_to<Key>,
    typename Alloc =
        std::allocator<std::pair<const Key, T>>>
class unordered_map;
```

Unordered Associative Containers (2)

1. The unordered associative containers are implemented as hash tables.
2. They use user-provided hash functions and comparison ops to manage the tables.
3. Unlike the ordered associative containers, the comparison op implements “equality”, not “less than”.
4. The hash op and the comparison op are required to play well together.

What do you mean by “play well together”?

The relationship between the hash and the comparison fns is easy to discover if you think about how hash tables work.

Conceptually, a hash table is a collection of “buckets”, and in a lookup operation, the hash function chooses the bucket, and the equality function finds the individual item in the bucket.

So, if two elements are equal, they need to hash to the same value.

But the converse is not true; two things that are not equal may still hash to the same value.

Hash Talks this week

1. Wednesday: Matt Kulukundis - Designing a Fast, Efficient, Cache-friendly Hash Table, Step by Step
2. Thursday: Nicholas Ormrod - Fantastic Algorithms and where to find them
3. Friday: Phil Nash - The Holy Grail - A Hash Array Mapped Trie for C++

The importance of a good hash function

Matt Kulukundis' talk went into detail about this (and other things).

As a test, I built a hash table of all the prime numbers less than 100, using various hash functions.

```
    size_t idhash  (int i) { return i; }
    size_t halfhash(int i) { return i >> 1; }

    // random number chosen by dice roll
    // Ref: https://xkcd.com/221/
    size_t randhash(int i) { return 4; }
```

```
Using Set =
    std::unordered_set<int, size_t(*)(int)>;
```

Questions?

Thank you