From security to performance to GPU programming: Exploring modern allocators

Sergey Zubkov

Morgan Stanley

What do they do?

```
template<class T, class Allocator> class vector;
```

1. Allocation policy
2. Deallocation policy
3. Construction policy
4. Destruction policy
5. Addressing* policy (“fancy” or “synthetic” pointers)

*must have mapping to heap memory to use standard containers

How are they used?

1. STL’s static allocators (allocator is part of the type)

```
std::vector<T, Allocator> v;
```

2. C++17’s dynamic allocators (supplied at run-time)

```
std::pmr::vector<T> v(&pool);
```

Enough theory, let’s see what this can do

Just a few of the many off-the-shelf allocators available now:

- (various libraries) secure_allocator
- std::pmr::polymorphic_allocator
- tbb::scalable_allocator

- boost::interprocess::allocator (fancy pointers inside!)
  - (with a shout-out to std::scoped_allocator_adaptor)
- boost::compute::pinned_allocator (even fancier pointers inside!)

secure_allocator

…as reinvented in Bitcoin, Botan, MongoDB, and even JsonCpp

- On allocation:
  - locks memory so that it cannot be swapped out (mlock+madvise/VirtualLock)
- On deallocation:
  - wipes memory before freeing (OPENSSL_cleanse, RtlSecureZeroMemory, etc)

secure_allocator

```
{
  std::string pwd = "correct horse battery staple";
  ...
}
p = ...; // hackers got this address
for (int n = 0; n < 28; ++n)
  if (isprint(p[n])) std::cout << p[n]; else std::cout << "�";
```

```
heap contents at 0x2534c20:
��������horse battery staple
```

secure_allocator

```
#include "support/allocators/secure.h“
{
  SecureString pwd = "correct horse battery staple";
  ...
}
p = ...; // hackers got this address
for (int n = 0; n < 28; ++n)
  if (isprint(p[n])) std::cout << p[n]; else std::cout << "�";
```

```
heap contents at 0x7f7ac0236fe0:
����������������������������
```

secure_allocator

```
#include "support/allocators/secure.h“
{
  SecureString pwd = "correct horse battery staple";
  ...
}
p = .
for (
  if
```

```
// in that header...

template <typename T>
struct secure_allocator /* we’re not going there! */;

typedef std::basic_string<char,
                          std::char_traits<char>,
                          secure_allocator<char> > SecureString;
```

```
heap contents at 0x7f7ac0236fe0:
����������������������������
```

Based on a true story…

```
struct Event { std::vector<int> data = std::vector<int>(512); };
std::list<std::shared_ptr<Event>> q;
void producer() {
  for (int n = 0; n != /* LOTS */; ++n) {
    std::lock_guard<std::mutex> lk(m);
    q.push_back(std::make_shared<Event>());
    cv.notify_one();
  }
}
void consumer() { /* might resize the event… */ }
```

What happens after running this for one year?

Based on a true story…

After this demo completes:

- 50k events x 16 produces x 4 consumers
  - `Total non-mmapped bytes (arena):        1,625,690,112`
  - `Total allocated space (uordblks):       119,632`
  - `Total free space (fordblks):            1,625,570,480`
- 100k events x 16 producers x 4 consumers
  - `Total non-mmapped bytes (arena):        3,310,215,168`
  - `Total allocated space (uordblks):       121,552`
  - `Total free space (fordblks):            3,310,093,616`

std::pmr::polymorphic_allocator

C++17 to the rescue!

```
std::pmr::synchronized_pool_resource pool;
struct Event {
    std::pmr::vector<int> data = std::pmr::vector<int>(512, &pool);
};
std::pmr::list<std::shared_ptr<Event>> q(&pool);
void producer() {
  for (int n = 0; n != /* LOTS /; ++n) {
    std::lock_guard<std::mutex> lk(m);
    q.push_back(std::allocate_shared<Event,
                        std::pmr::polymorphic_allocator<Event>>(&pool));
    cv.notify_one();
  }
}
```

std::pmr::polymorphic_allocator

After this demo completes:

- 50k events x 16 produces x 4 consumers
  - `Total non-mmapped bytes (arena):       2,908,160`
  - `Total allocated space (uordblks):      118,832`
  - `Total free space (fordblks):           2,789,328`
- 100k events x 16 producers x 4 consumers
  - `Total non-mmapped bytes (arena):       2,908,160 (1000x!)`
  - `Total allocated space (uordblks):      118,832`
  - `Total free space (fordblks):           2,789,328`

tbb::scalable_allocator

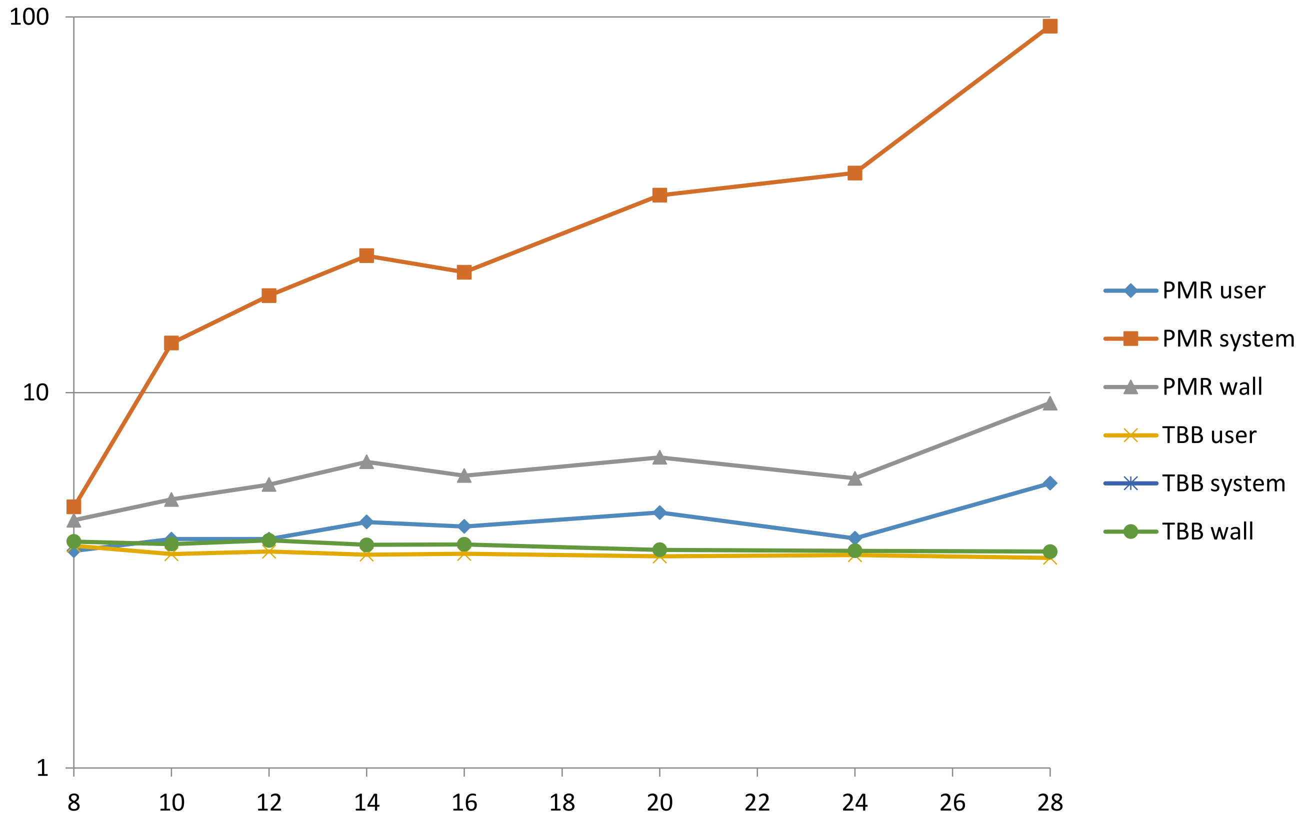
Size-segregated thread-private heaps, and many optimizations[*]

- Alexey Kukanov and Michael J. Voss "The Foundations for Scalable Multi-core Software in Intel® Threading Building Blocks", Intel Technology Journal, Volume 11, Issue 4, 2007

```
struct Event { std::vector<int, tbb::scalable_allocator<T>> data{512}; };
std::list<std::shared_ptr<Event>, tbb::scalable_allocator<T>> q;
void producer() {
  for (int n = 0; n != /* LOTS */; ++n) {
    std::lock_guard<std::mutex> lk(m);
    q.push_back(std::allocate_shared<Event,
                              tbb::scalable_allocator<Event>>({}));
    cv.notify_one();
  }
}
```

tbb::scalable_allocator

Time to transmit 160k events in the previous demo vs number of threads



boost::interprocess::allocator

How do you share a C++ container between processes?

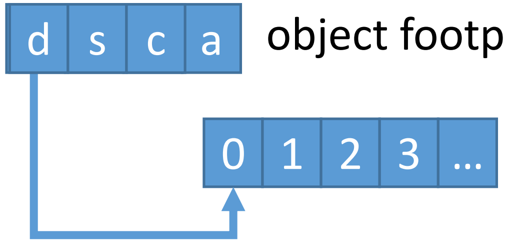
```
namespace ipc = boost::interprocess;
ipc::managed_shared_memory segment{ /* ... */ };

using vec_t = std::vector<int>;
vec_t* v = segment.construct<vec_t>("vec")(in.begin(), in.end());
```

```
                    ... in another process ...
vec_t* m = segment.find<vec_t>("vec").first;
for (int& n : *m) std::cout << n << ' '; // SEGFAULT!
```

boost::interprocess::allocator

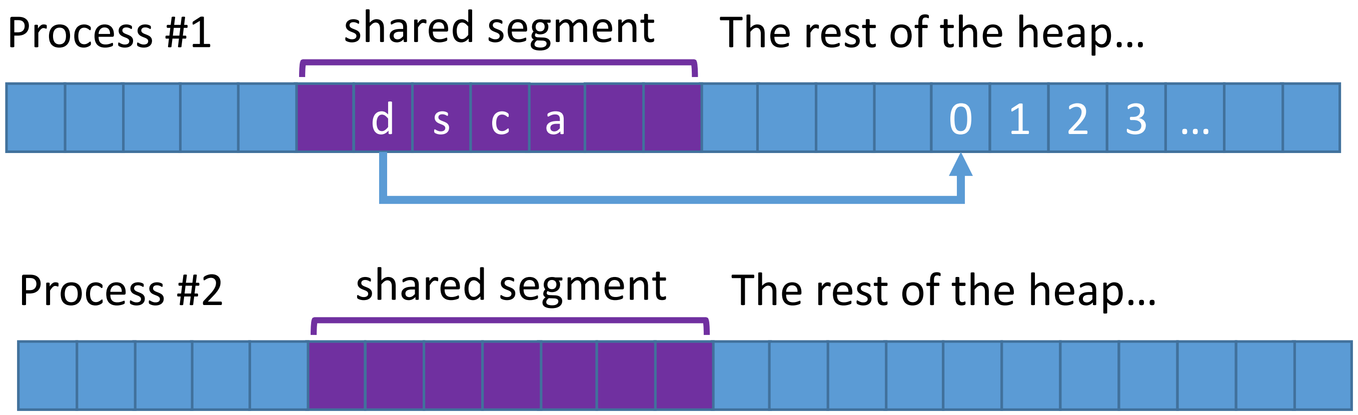
How do you share a C++ container between processes?

What’s in a `vector<int>`?



object footprint: data, size, capacity, allocator

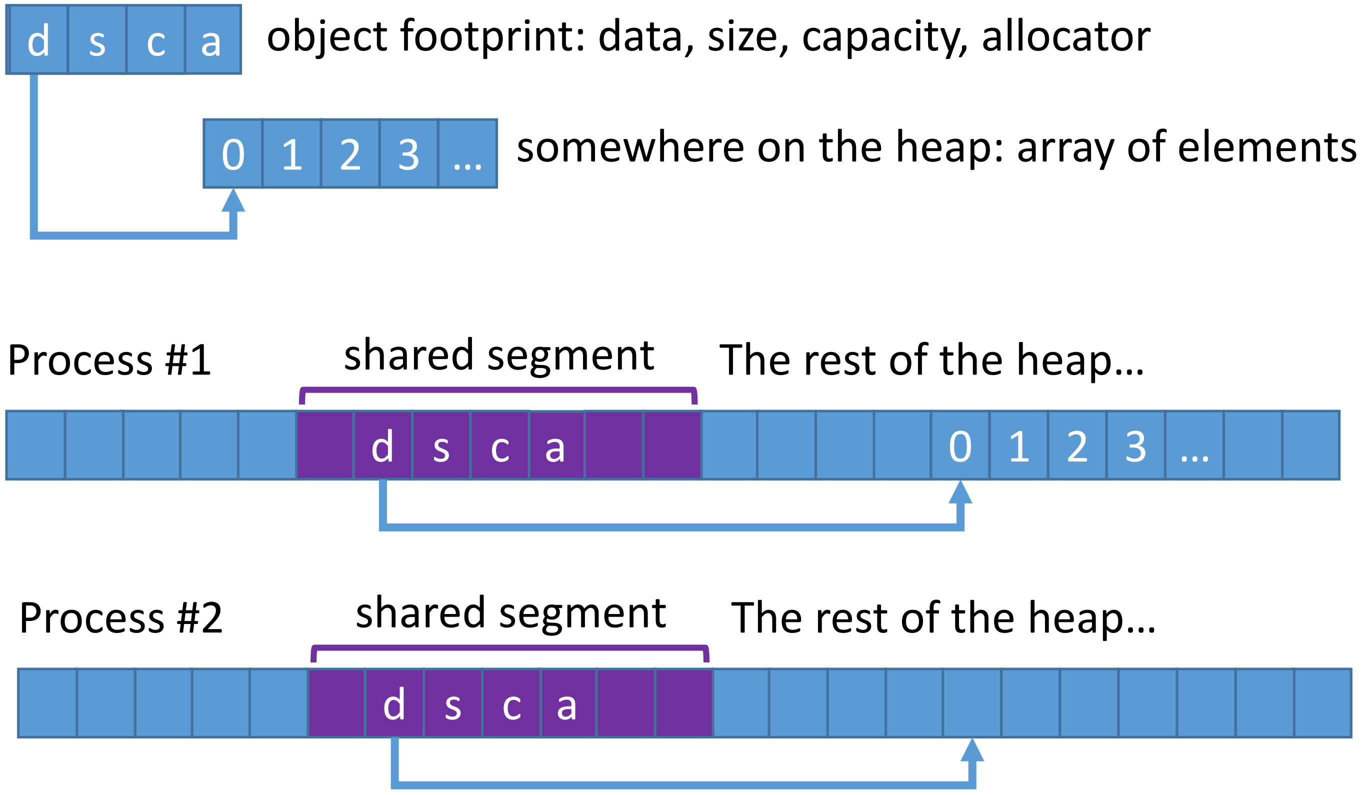
somewhere on the heap: array of elements



boost::interprocess::allocator

How do you share a C++ container between processes?

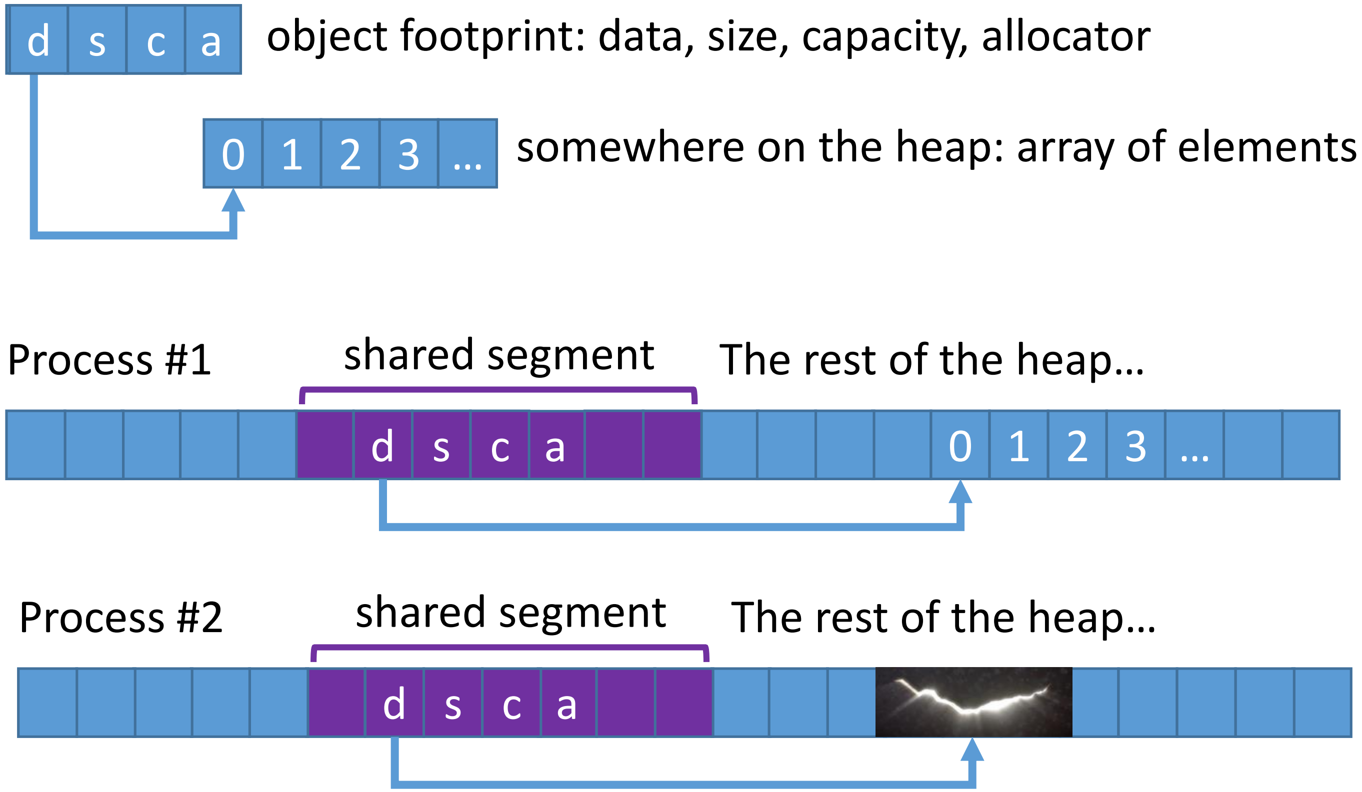
What's in a `vector<int>`?



boost::interprocess::allocator

How do you share a C++ container between processes?

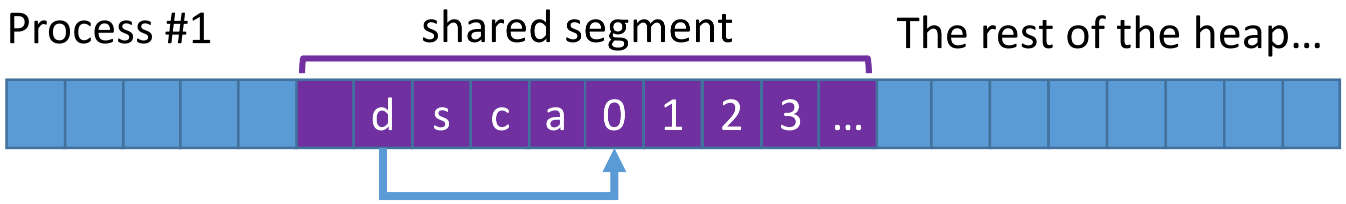
What's in a `vector<int>`?



boost::interprocess::allocator

How do you share a C++ container between processes?

```cpp
using alloc_t = ipc::allocator<int, ipc::managed_shared_memory::segment_manager>;
alloc_t a{ segment.get_segment_manager() };

using vec_t = std::vector<int, alloc_t>;
vec_t* v = segment.construct<vec_t>("vec")(in.begin(), in.end(), a);
```

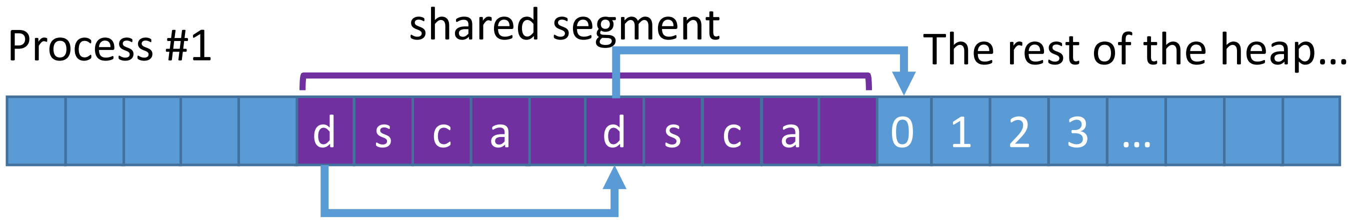



```cpp
                       ... in another process ...
vec_t* m = segment.find<vec_t>("vec").first;
for (int& n : *m) std::cout << n << ' '; // WORKS
```

boost::interprocess::allocator

What if there is a container inside that container?

```
using mat_t = std::vector<std::vector<int>, alloc_t<std::vector<int>>>;
mat_t& m = *segment.construct<mat_t>("matrix")(in.begin(), in.end(), a);
```




```
                          ... in another process ...
using mat_t = std::vector<std::vector<int>, alloc_t<std::vector<int>>>;
mat_t* m = segment.find<mat_t>("matrix").first;
for (auto& r : *m)
    for (int n : r)
        std::cout << n << ' '; // SEGFAULT!
```

boost::interprocess::allocator

What if there is a container inside that container?

```
using vec_t = std::vector<int, alloc_t<int>>;
using mat_t = std::vector<vec_t, std::scoped_allocator_adaptor<alloc_t<vec_t>>>;
mat_t& m = *segment.construct<mat_t>("matrix")(3, a);

m[0].resize(3); // allocates the row inside the same segment
m[2].assign({ 6, 7, 8 });
```

```
                                ... in another process ...
using vec_t = std::vector<int, alloc_t<int>>;
using mat_t = std::vector<vec_t, std::scoped_allocator_adaptor<alloc_t<vec_t>>>;
mat_t* m = segment.find<mat_t>("matrix").first;
for (auto& r : *m)
    for (int n : r)
        std::cout << n << ' '; // WORKS
```

Fancy pointer

Our vector's data pointer is not T*!
It is `boost::interprocess::offset_ptr<T>`

https://github.com/boostorg/interprocess/blob/develop/include/boost/interprocess/offset_ptr.hpp#L207

```
//!A smart pointer that stores the offset between between the pointer and the
//!the object it points. This allows offset allows special properties, since
//!the pointer is independent from the address of the pointee, if the
//!pointer and the pointee are still separated by the same offset. This feature
//!converts offset_ptr in a smart pointer that can be placed in shared memory and
//!memory mapped files mapped in different addresses in every process.
```

See C++ Now 2017: Bob Steagall "Testing the Limits of Allocator Awareness"
https://www.youtube.com/watch?v=fmJfKm9ano8
https://github.com/boostcon/cppnow_presentations_2017/tree/master/05-18-2017_thursday

boost::compute::pinned_allocator

- On allocation:
  - clCreateBuffer(CL_MEM_ALLOC_HOST_PTR), clRetainMemObject
- On deallocation:
  - clReleaseMemObject
- Fancy pointer
  - device_ptr<T> (holds a reference to a device buffer and an index)

```
namespace bc = boost::compute;
bc::device d = bc::system::default_device();
bc::command_queue queue(bc::context{ d }, d);

bc::vector<float, bc::pinned_allocator<float>> v({ 1.f, 2.f, 3.f, 4.f }, queue);
bc::transform(v.begin(), v.end(), v.begin(), bc::sqrt<float>(), queue);
for (float f : v) std::cout << f << ' ';
```

and many more…

- Allocators are reusable, packaged, policies that you can pick and choose.
- Hundreds of them are in use, in production.

cache_aligned_allocator	any_interprocess_allocator	mmap_allocator	tlsf_allocator
node_allocator	stack_allocator	HugeAllocator	aligned_allocator
private_node_allocator	__mt_alloc	LowAllocator	tracker_alloc
cached_node_allocator	__pool_alloc	SpecialAllocator	traceable_allocator
adaptive_pool	_ExtPtr_allocator	StackAllocator	libc_allocator_with_realloc
private_adaptive_pool	array_allocator	aligned_allocator_cpp11	throw_allocator_random
cached_adaptive_pool	bitmap_allocator	gc_allocator	casacore_allocator
pool_allocator	debug_allocator	cacheline_allocator	mallocator
fast_pool_allocator	throw_allocator	pyr_pool_compile_allocator	recycling_allocator
…	…	…	…

Allocator track on CppCon 2017

- 9/25 3:15 pm Alisdair Meredith “An allocator model for std2”
- 9/25 4:45 pm <you are here>
- 9/26 2:00 pm John Lakos “Local (‘Arena’) Memory Allocators”
- 9/28 2:00 pm Bob Steagall “How to Write a Custom Allocator”
- 9/29 9:00 am Pablo Halpern “Modern Allocators: The Good Parts”
- 9/29 1:30 pm Marshall Clow “Customizing the Standard Containers”