Taming Dynamic Memory

An Introduction to Custom Allocators

Andreas Weis

BMW AG

ACCU, April 12, 2019

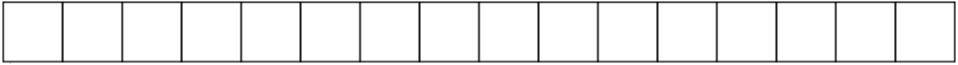
About me

- ComicSansMS
- @DerGhulbus
- Co-organizer of the Munich C++ User Group
- Currently working as a Software Architect for BMW

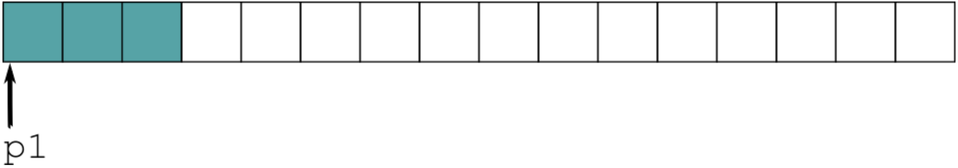
Overview

- What's wrong with global `new` and `delete`?
- Local allocators
- Alternative allocation strategies
- Allocator support in C++

General purpose allocator

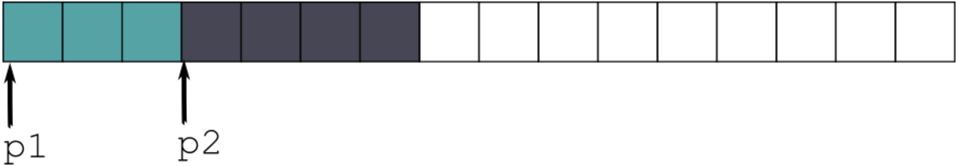


General purpose allocator



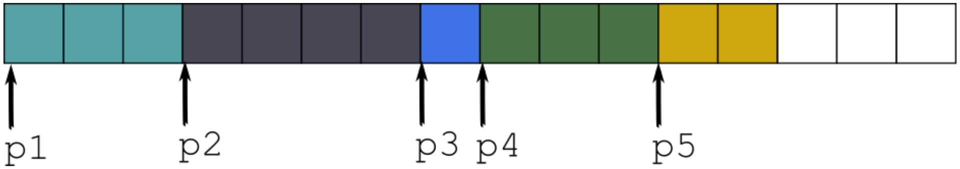

```
auto p1 = allocate(3);
```

General purpose allocator



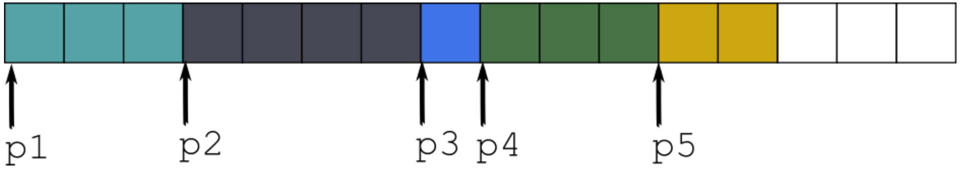

```
auto p1 = allocate(3);
auto p2 = allocate(4);
```

General purpose allocator

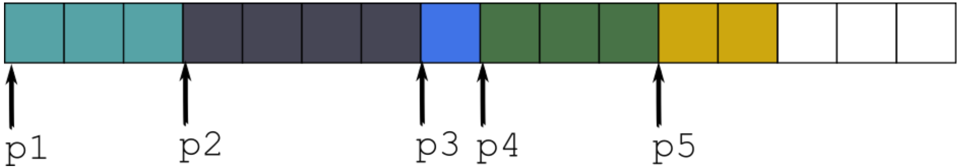



```
auto p1 = allocate(3);
auto p2 = allocate(4);
auto p3 = allocate(1);
auto p4 = allocate(3);
auto p5 = allocate(2);
```

General purpose allocator

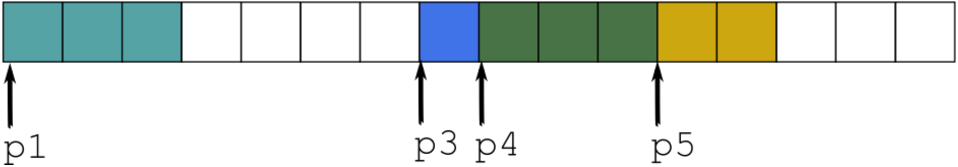


General purpose allocator



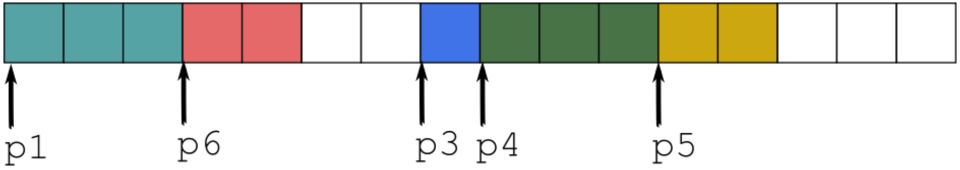

```
deallocate(p2);
```

General purpose allocator



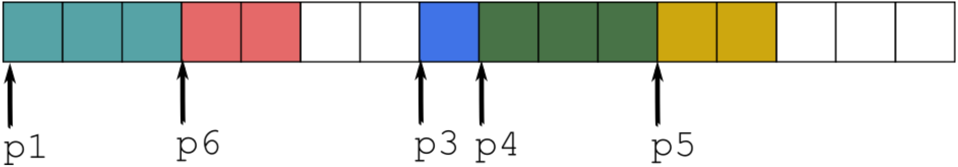

```
deallocate(p2);
```

General purpose allocator



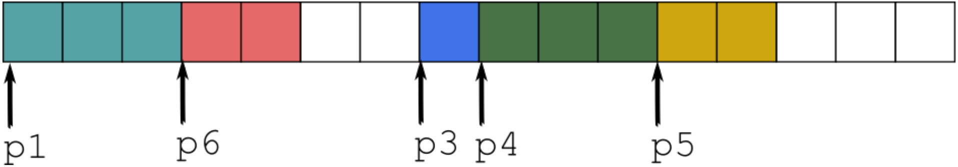

```
deallocate(p2);
auto p6 = allocate(2);
```

Fragmentation




```
auto p7 = allocate(4);
```

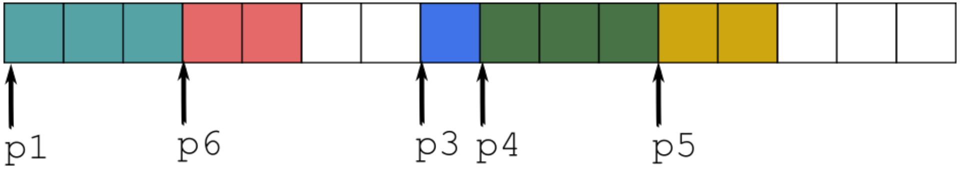
Fragmentation




```
auto p7 = allocate(4);
```

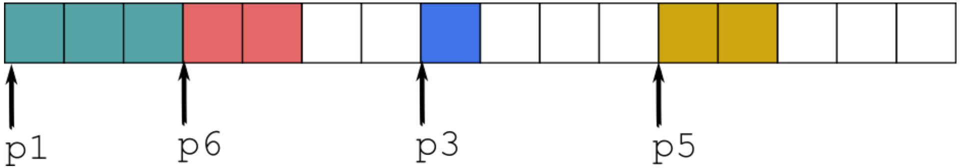
Runtime Error!

Coalescing



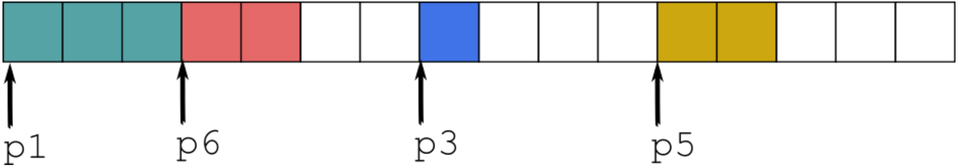
`deallocate(p4);`

Coalescing



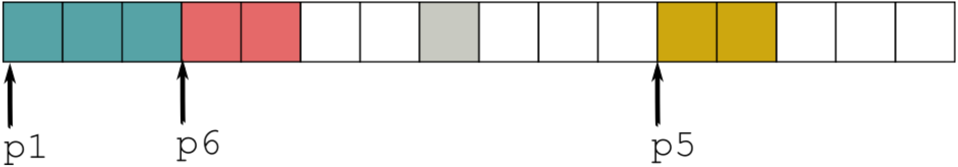

```
deallocate(p4);
```

Coalescing



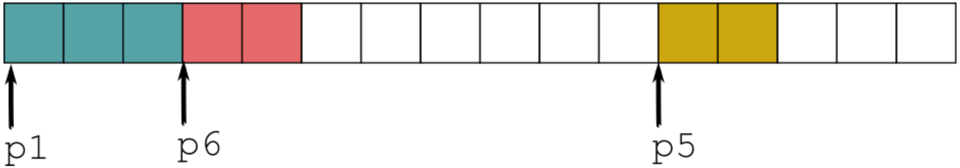

```
deallocate(p4);
deallocate(p3);
```

Coalescing




```
deallocate(p4);
deallocate(p3);
```

Coalescing




```
deallocate(p4);
deallocate(p3);
```

Problems with default allocator

- Complex runtime behavior
  - What is the maximum memory usage?
  - What is the worst-case execution time for an allocation or deallocation?
- Shared global state
  - Reasoning about allocator behavior requires global knowledge of the whole program
  - The singular resource global allocator is a potential bottleneck

It's not just about performance!

From Global to Local

```
auto p1 = allocate(42);
deallocate(p1);
```

From Global to Local

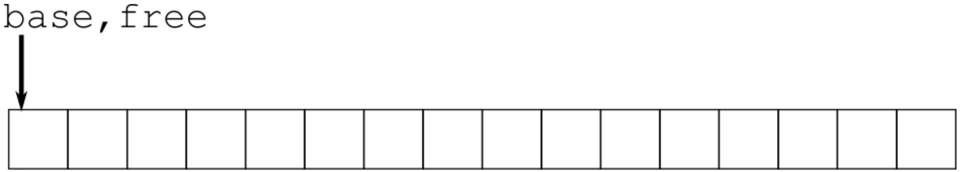
```
Allocator alloc;

auto p1 = alloc.allocate(42);
alloc.deallocate(p1);
```

Problems with default allocator

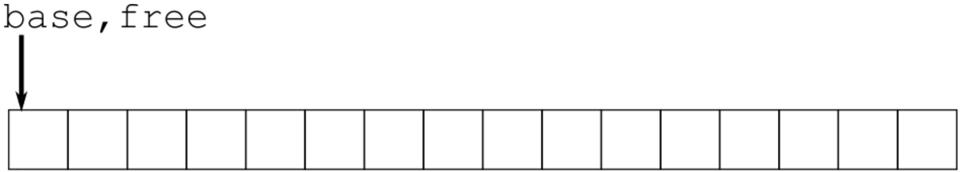
- Complex runtime behavior
  - What is the maximum memory usage?
  - What is the worst-case execution time for an allocation or deallocation?
- Shared global state ✓[1]

[1] John Lakos - Allocator-Aware Software, ACCU 2019

Monotonic Allocator

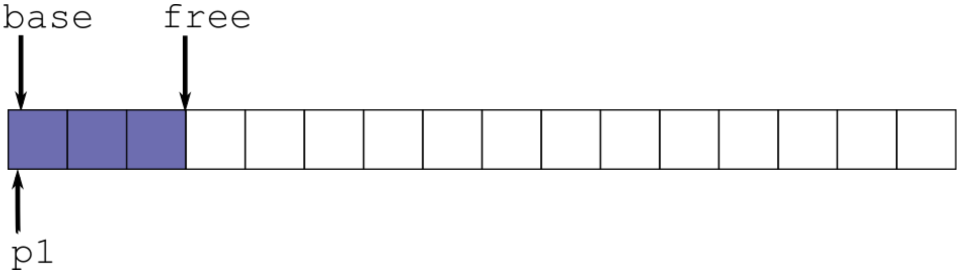


Monotonic Allocator



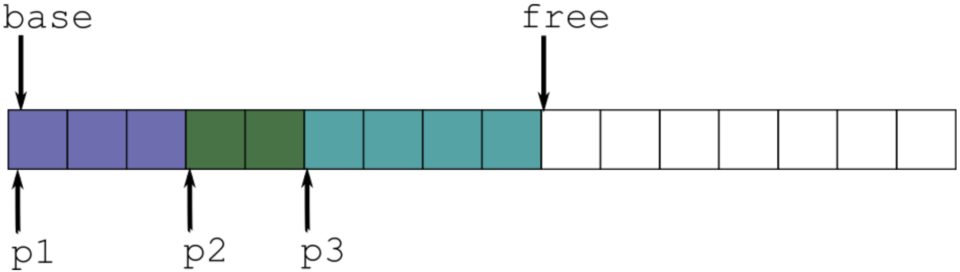

```
auto p1 = monot.allocate(3);
```

Monotonic Allocator



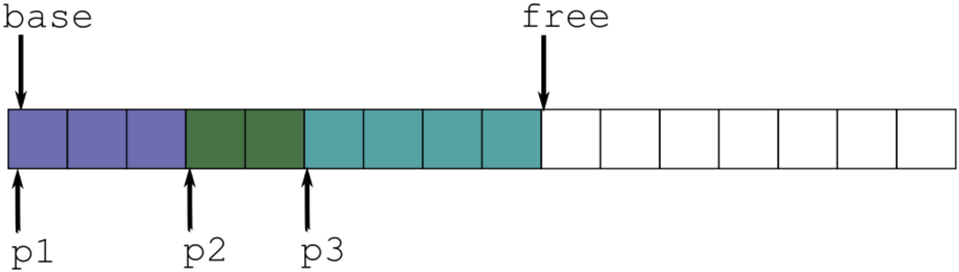

```
auto p1 = monot.allocate(3);
```

Monotonic Allocator



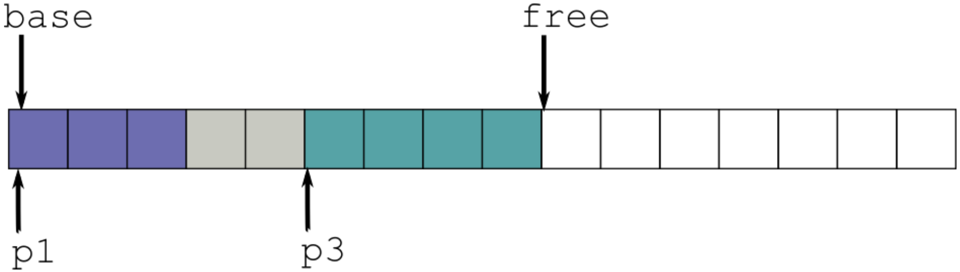

```
auto p1 = monot.allocate(3);
auto p2 = monot.allocate(2);
auto p3 = monot.allocate(4);
```

Monotonic Allocator



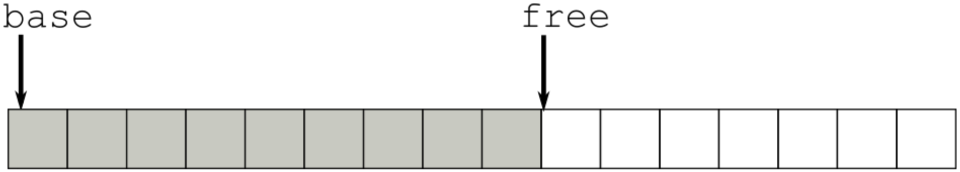

```
monot.deallocate(p2);
```

Monotonic Allocator



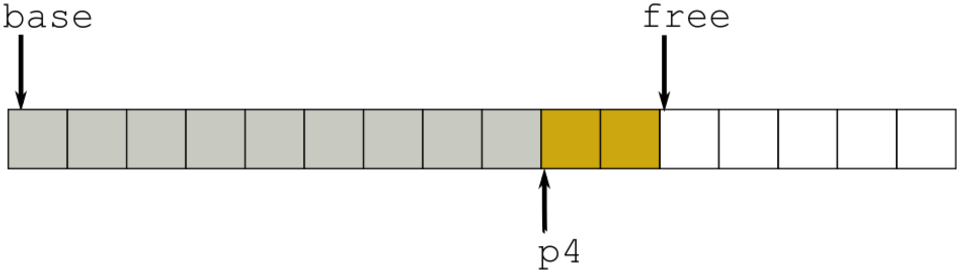

```
monot.deallocate(p2);
```

Monotonic Allocator



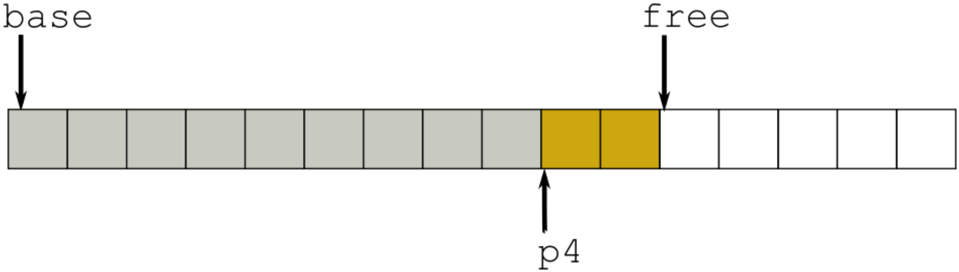

```
monot.deallocate(p2);
monot.deallocate(p1);
monot.deallocate(p3);
```

Monotonic Allocator



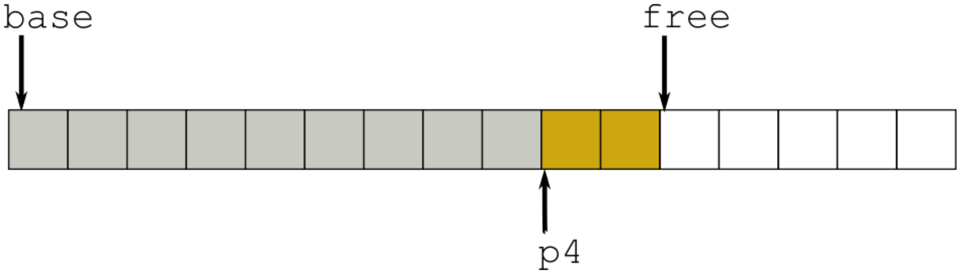

```
monot.deallocate(p2);
monot.deallocate(p1);
monot.deallocate(p3);

auto p4 = allocate(2);
```

Monotonic Allocator



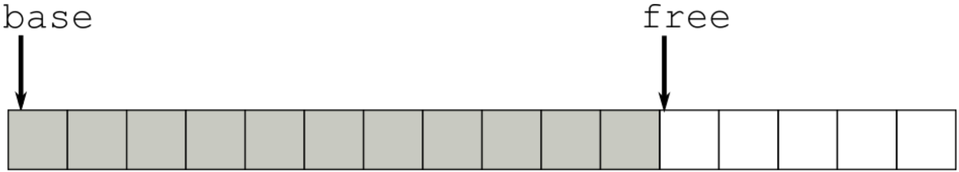

```
monot.deallocate(p2);
monot.deallocate(p1);
monot.deallocate(p3);

auto p4 = allocate(2);
```

Monotonic Allocator - Reclamation



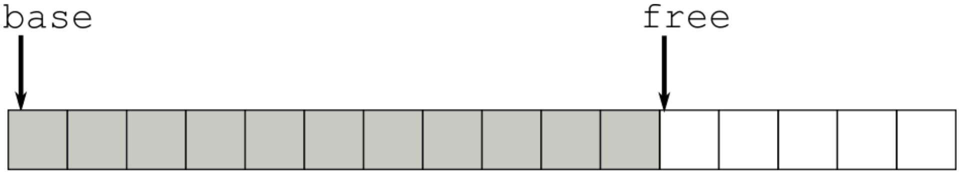

```
monot.deallocate(p4);
```

Monotonic Allocator - Reclamation



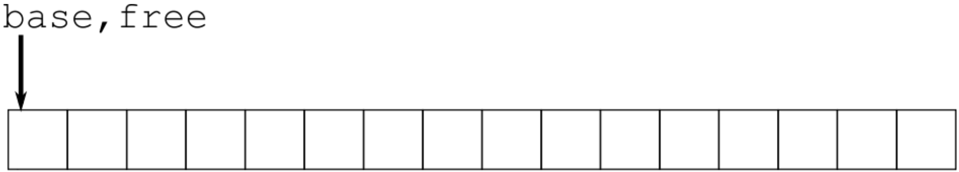

```
monot.deallocate(p4);
```

Monotonic Allocator - Reclamation




```
monot.deallocate(p4);

monot.release();
```

Monotonic Allocator - Reclamation




```
monot.deallocate(p4);

monot.release();
```

Monotonic Allocator

- Deterministic runtime cost
- Extremely efficient
- No fragmentation
- Easy to implement
- Trivial to make thread-safe

But:

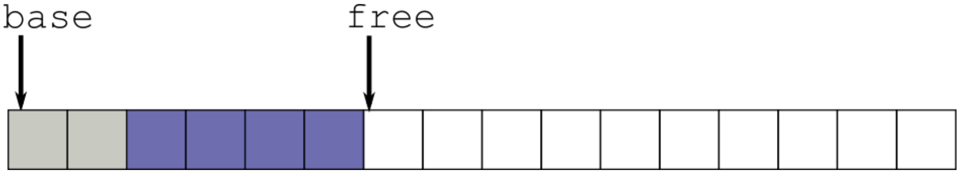
- Memory can only be reclaimed all at once

Where is this actually useful?

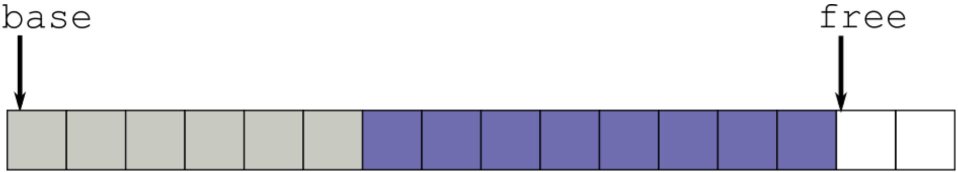
- Frames in a video game
- Handling of a single event in an event-driven system
- Cyclic execution in a real-time system
- Containers that are initialized but not changed after
- `static` state - Objects that will never be destroyed

Monotonic Allocator - `std::vector`

Monotonic Allocator - `std::vector`



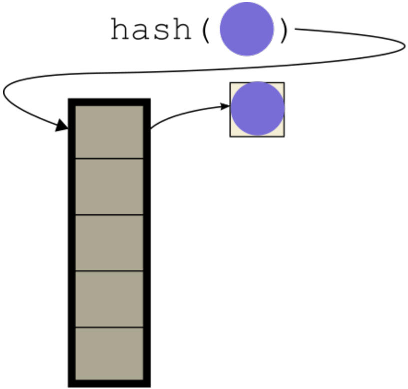
Monotonic Allocator - `std::vector`



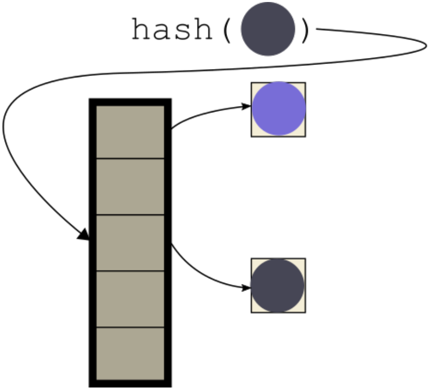
Monotonic Allocator - STL containers

- `vector` should `reserve` final size upfront
- `list` and `map` work fine, but deleted elements are not reclaimed individually
- `deque` works really well
- `unordered_map` deserves a closer look. . .

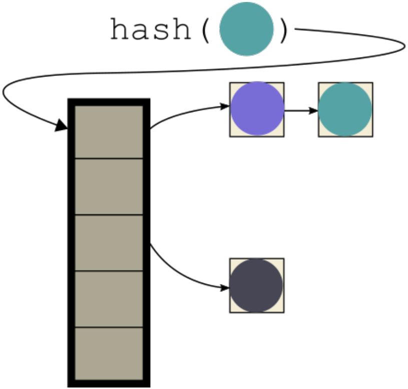
unordered_map



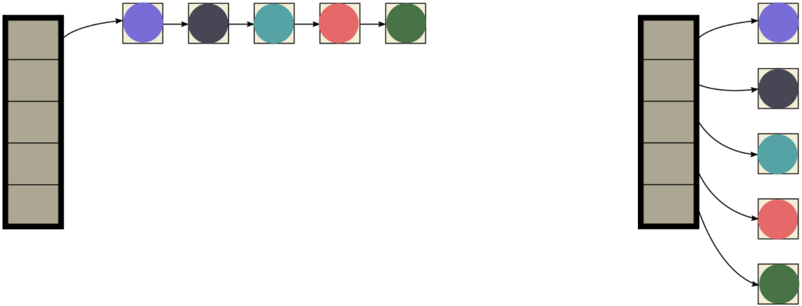
unordered_map



unordered_map

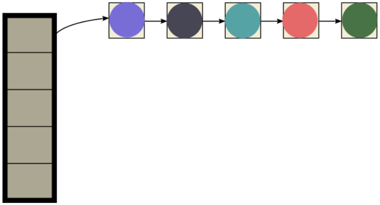


unordered_map

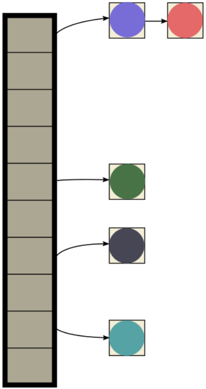


Exact layout depends on hash function and inserted values

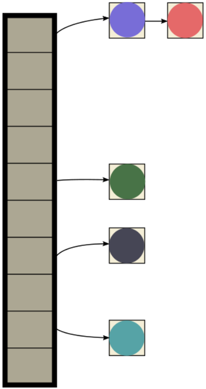
unordered_map – Rehashing



unordered_map – Rehashing

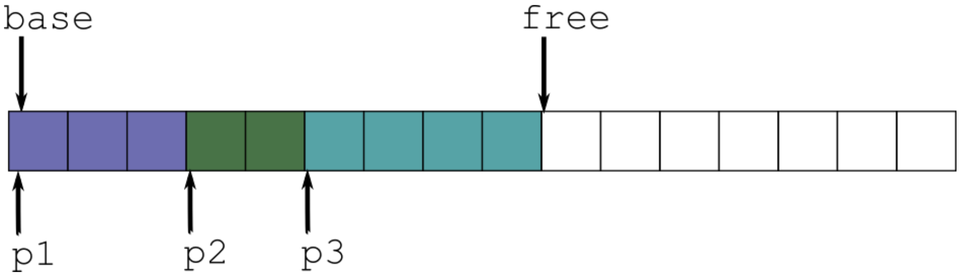


`unordered_map` – Rehashing

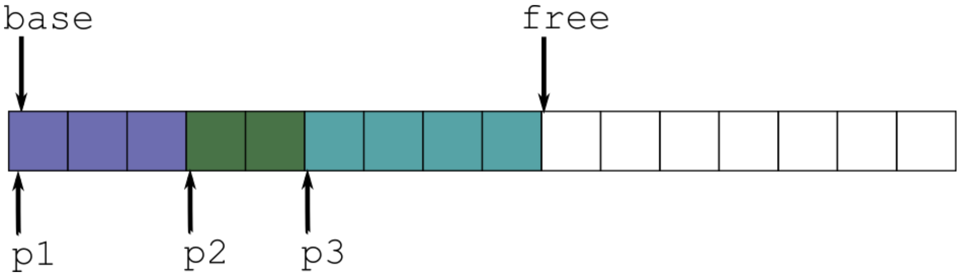


Allocation behavior is a hybrid between `vector` and `list`

Stack Allocator

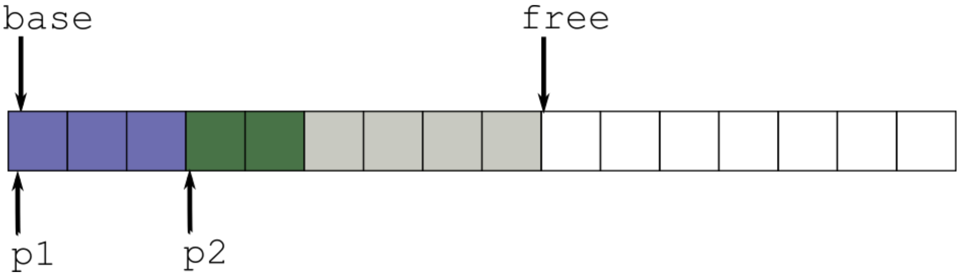


Stack Allocator



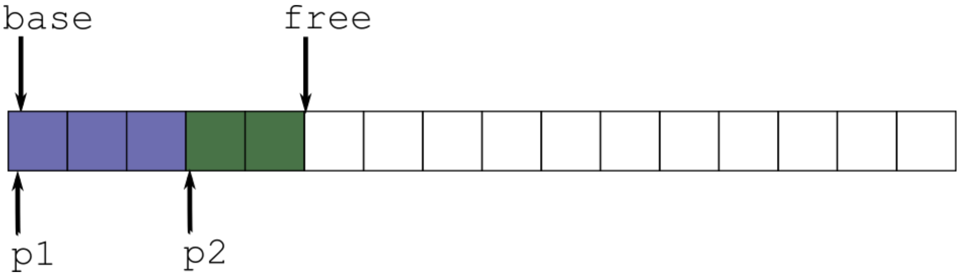

```
monot.deallocate(p3);
```

Stack Allocator




```
monot.deallocate(p3);
```

Stack Allocator

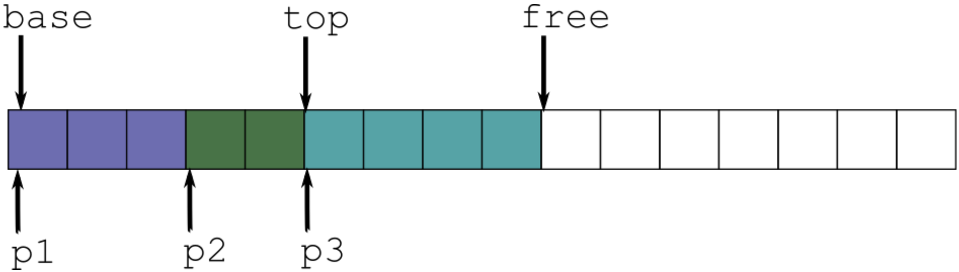



```
monot.deallocate(p3);
```

Stack Allocator

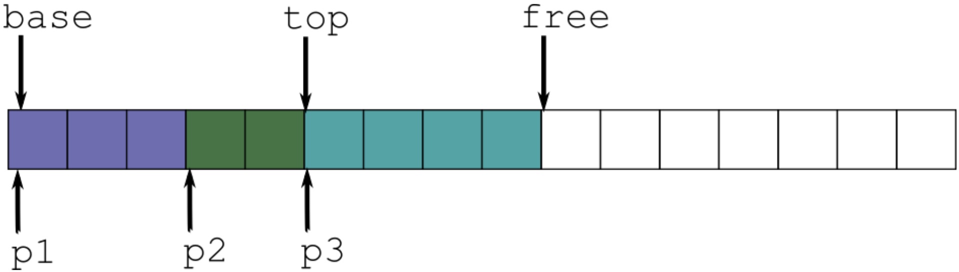
- Strict LIFO-ordering of allocations and deallocations
- No way for the implementation to check whether the deallocation order is correct!

Stack Allocator



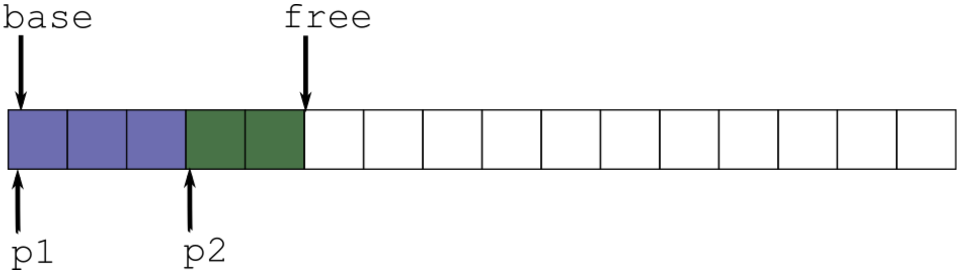

```
monot.deallocate(p3);
```

Stack Allocator

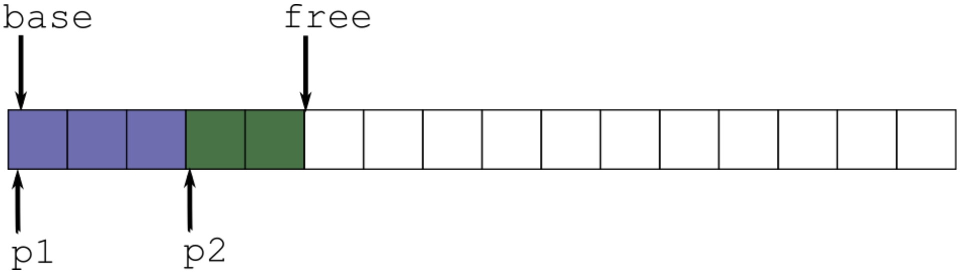



```
monot.deallocate(p3);
p3 == top  ✓
```

Stack Allocator



Stack Allocator



`top = ???`

Stack Allocator

- Strict LIFO-ordering of allocations and deallocations.
- No way for the implementation to check whether the deallocation order is correct!

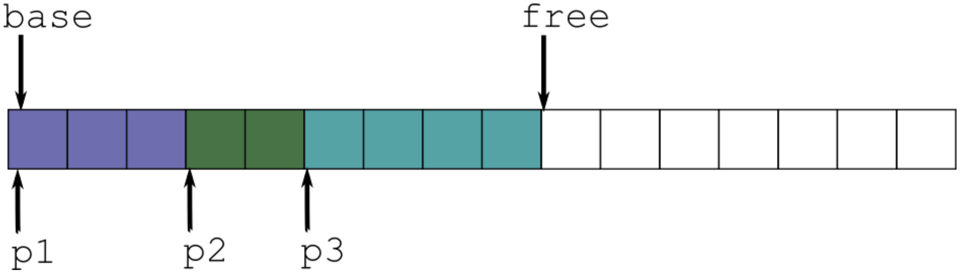
Stack Allocator

- Strict LIFO-ordering of allocations and deallocations.
- No way for the implementation to check whether the deallocation order is correct!
- Free-pointer is reset to the same pointer passed to the `deallocate` call

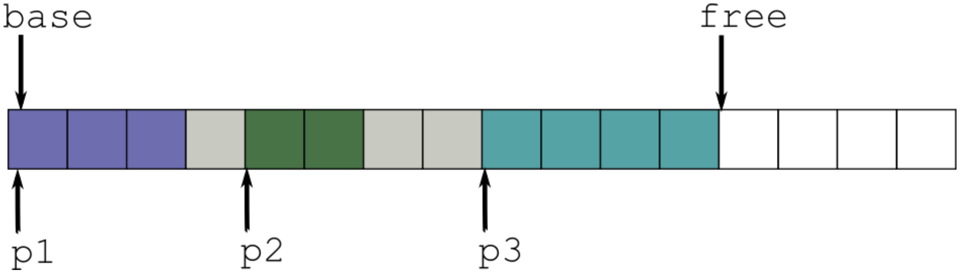
Stack Allocator

- Strict LIFO-ordering of allocations and deallocations.
- No way for the implementation to check whether the deallocation order is correct!
- Free-pointer is reset to the same pointer passed to the `deallocate` call
- Padding bytes may be lost to internal fragmentation

Padding



Padding



Alignment

```
  0xdeadbeef =

  d    e    a    d    b    e    e    f
1101 1110 1010 1101 1011 1110 1110 1111
```

Alignment

```
0xdeadbeef =

   d    e    a    d    b    e    e    f
 1101 1110 1010 1101 1011 1110 1110 1111
```

No alignment (1-byte aligned).

Alignment

```
0xdeadbeef =

  d    e    a    d    b    e    e    c
1101 1110 1010 1101 1011 1110 1110 1100
```

4-byte aligned.

Alignment

`0xdeadbeef =`

```
  d    e    a    d    b    e    e    8
1101 1110 1010 1101 1011 1110 1110 1000
```

8-byte aligned.

Alignment

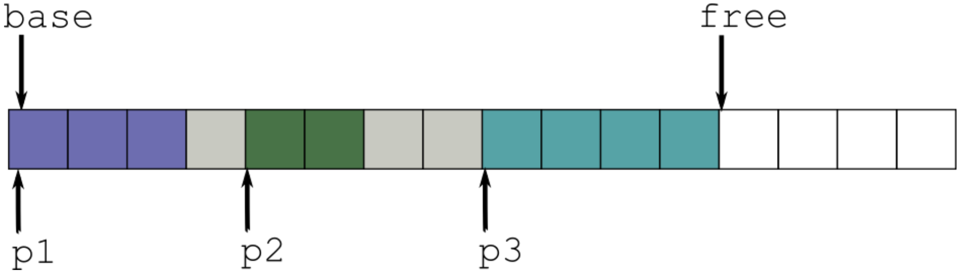
- Alignment refers to the least-significant bits of the object address being 0
- Alignment requirements are always specified in powers of 2
- Each built-in C++ type has a *natural* alignment requirement (typically `alignof(T) == sizeof(T)`)
- This is why `struct`s sometimes insert padding bytes between members

Alignment

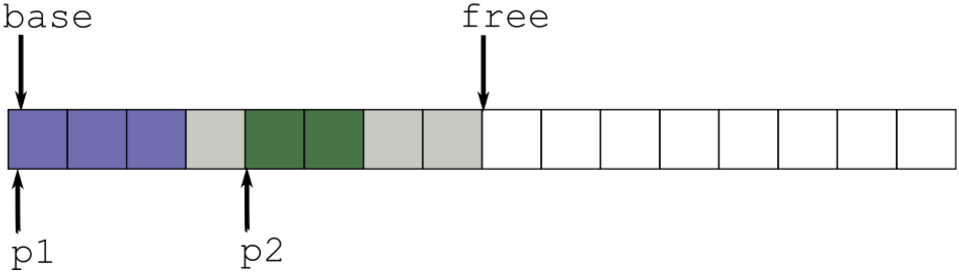
- Default allocator typically returns addresses aligned to `alignof(max_align_t)`, which is big enough for all built-in types
- Users may extend the alignment requirement for custom data types using `alignas`

```
void* allocate(std::size_t bytes,
               std::size_t alignment);
```

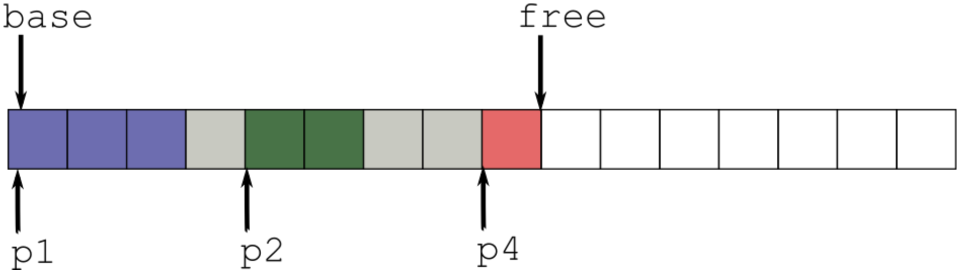
Padding



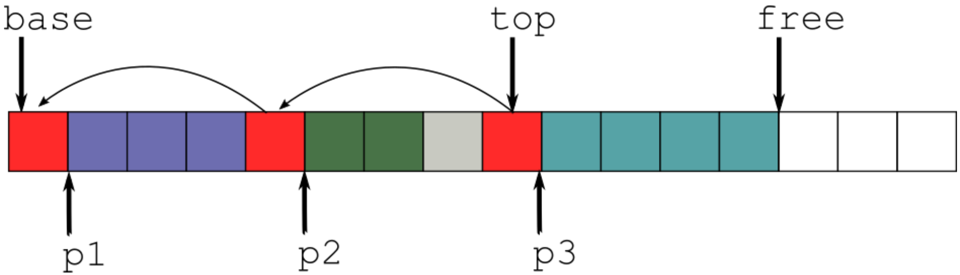
Padding



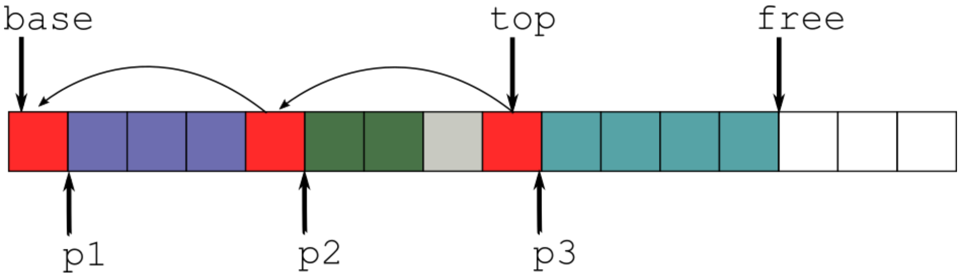
Padding



Monotonic Allocator - Extensions

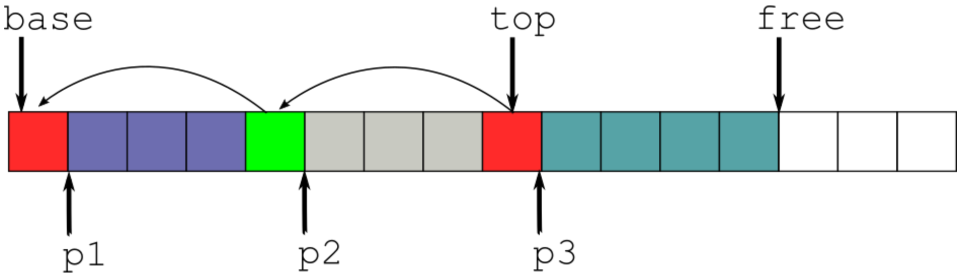


Monotonic Allocator - Extensions



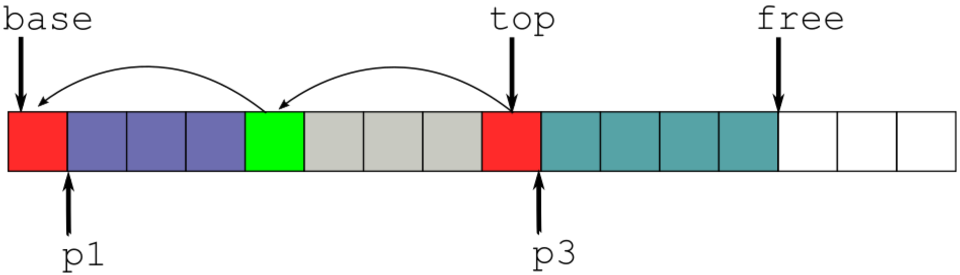

```
extpool.deallocate(p2);
```

Monotonic Allocator - Extensions



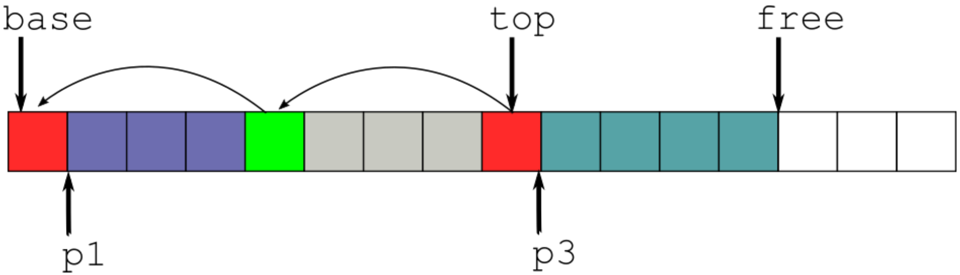
`extpool.deallocate(p2);`

Monotonic Allocator - Extensions



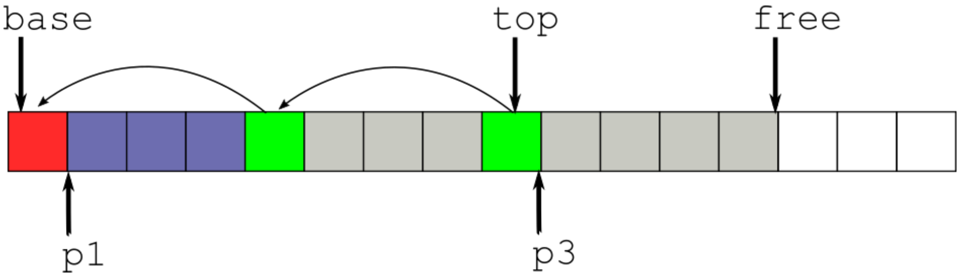

```
extpool.deallocate(p2);
```

Monotonic Allocator - Extensions



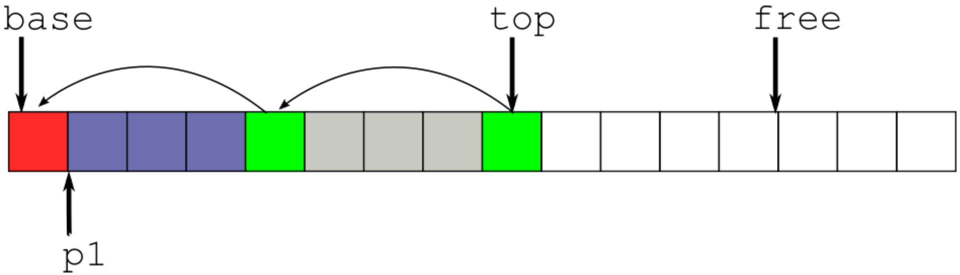

```
extpool.deallocate(p2);
extpool.deallocate(p3);
```

Monotonic Allocator - Extensions



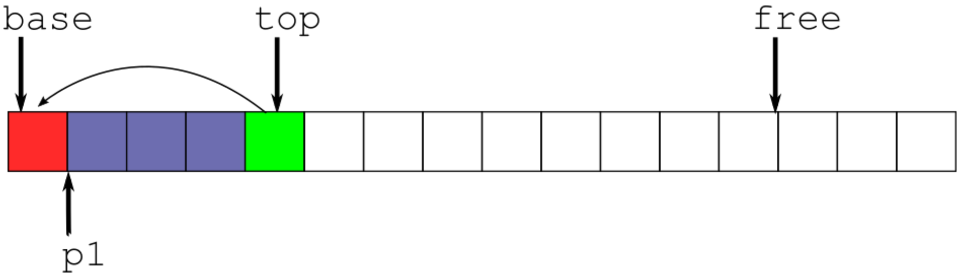

```
extpool.deallocate(p2);
extpool.deallocate(p3);
```

Monotonic Allocator - Extensions



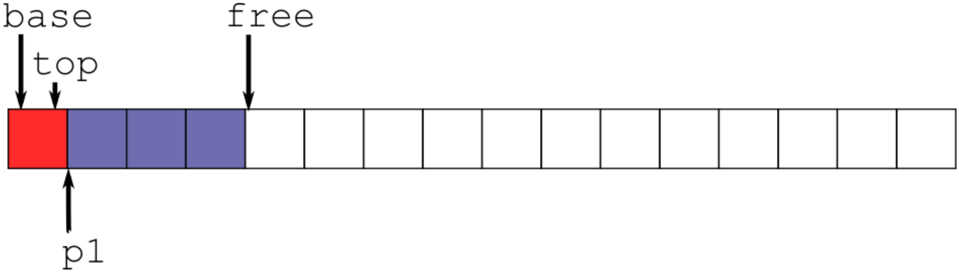

```
extpool.deallocate(p2);
extpool.deallocate(p3);
```

Monotonic Allocator - Extensions




```
extpool.deallocate(p2);
extpool.deallocate(p3);
```

Monotonic Allocator - Extensions

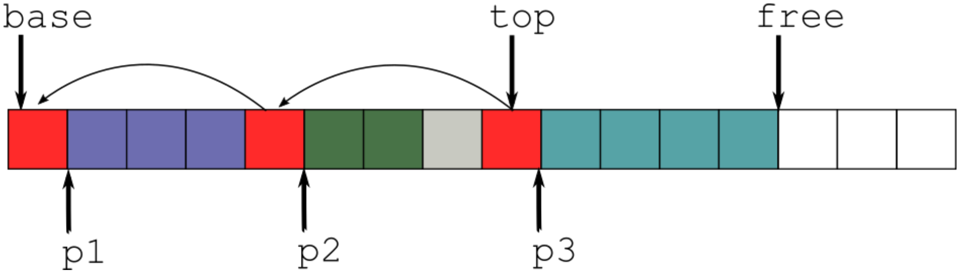



```
extpool.deallocate(p2);
extpool.deallocate(p3);
```

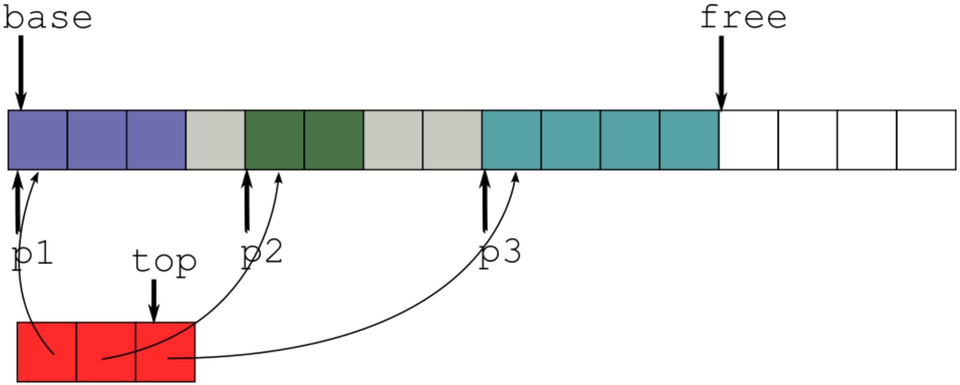
Monotonic Allocator - Extensions

- Auxiliary data structure required
- Runtime cost of deallocation now linear in number of allocations (amortized O(1))
- Auxiliary nodes have their own alignment requirements
- Where to store the auxiliary nodes?

Monotonic Allocator - Extensions



Monotonic Allocator - Extensions



Internal or External?

- External headers have better cache behavior when iterating the list
- External headers might have stricter alignment requirements than data
- Internal headers have better cache behavior when adjacent data is hot
- Internal headers require managed memory to be readable (think GPUs)
- Where does the storage for external headers come from? Same buffer? Different buffer? How big?

$\Rightarrow$ No easy answers.

The Bottom Line. . .

Even seemingly simple extensions get complicated very quickly.

Don't try to increase generality through clever extensions.
Only consider modifications if it's a perfect fit for your use case.

But what if I need to reclaim memory?

$\longrightarrow$ Pool Allocator

Pool Allocator

Pool Allocator

Pool Allocator

```
auto p1 = pool.allocate(2);
auto p2 = pool.allocate(4);
auto p3 = pool.allocate(3);
```

Pool Allocator

```cpp
auto p1 = pool.allocate(2);
auto p2 = pool.allocate(4);
auto p3 = pool.allocate(3);
```

Pool Allocator

```
auto p1 = pool.allocate(2);
auto p2 = pool.allocate(4);
auto p3 = pool.allocate(3);

pool.deallocate(p1);
```

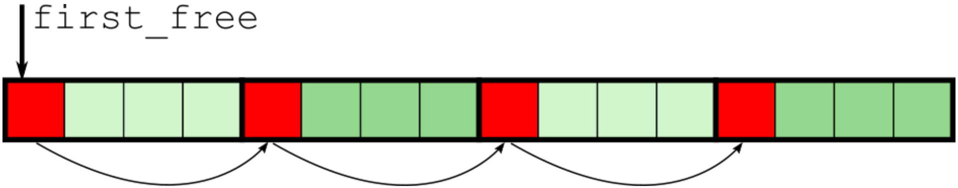
Pool Allocator

```
auto p1 = pool.allocate(2);
auto p2 = pool.allocate(4);
auto p3 = pool.allocate(3);

pool.deallocate(p1);
auto p4 = pool.allocate(1);
```

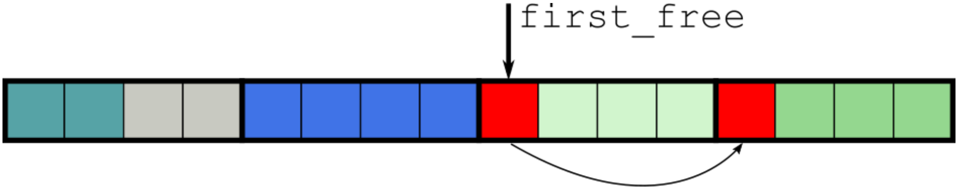
Pool Allocator

```
auto p1 = pool.allocate(2);
auto p2 = pool.allocate(4);
auto p3 = pool.allocate(3);

pool.deallocate(p1);
auto p4 = pool.allocate(1);
```

Pool Allocator - Reclamation

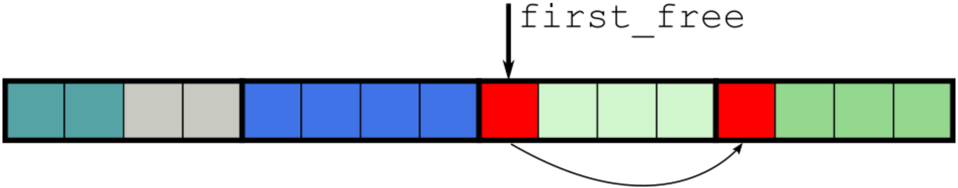


Pool Allocator - Reclamation



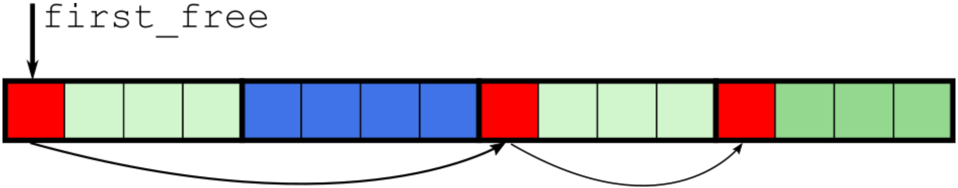

```
auto p1 = pool.allocate(2);
auto p2 = pool.allocate(4);
```

Pool Allocator - Reclamation



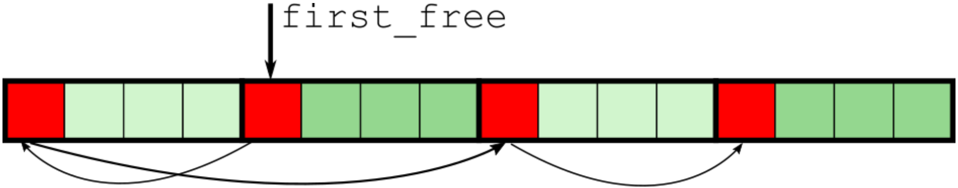

```cpp
auto p1 = pool.allocate(2);
auto p2 = pool.allocate(4);
```

Pool Allocator - Reclamation



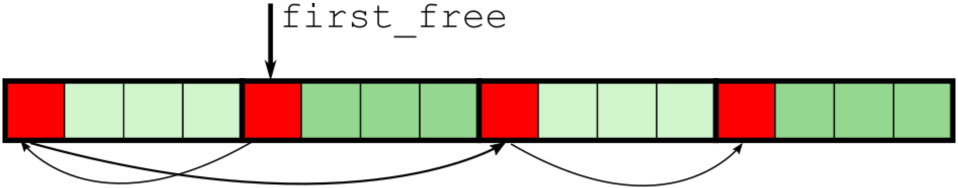

```
auto p1 = pool.allocate(2);
auto p2 = pool.allocate(4);

pool.deallocate(p1);
```

Pool Allocator - Reclamation



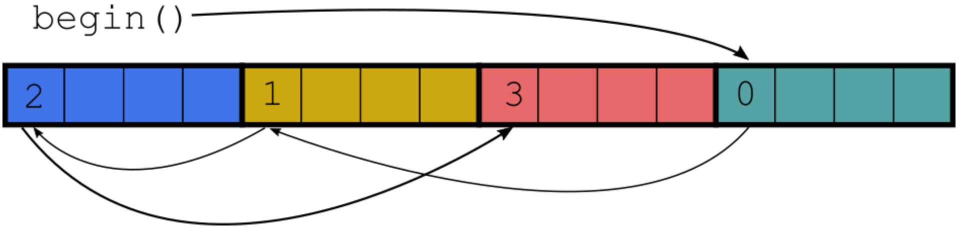

```
auto p1 = pool.allocate(2);
auto p2 = pool.allocate(4);

pool.deallocate(p1);
pool.deallocate(p2);
```

Pool Allocator - Reclamation




```
auto p1 = pool.allocate(2);
auto p2 = pool.allocate(4);

pool.deallocate(p1);
pool.deallocate(p2);
```

Pool Allocator - Diffusion



Pool Allocator

- Deterministic runtime cost
- No external fragmentation
- Easy to make thread-safe

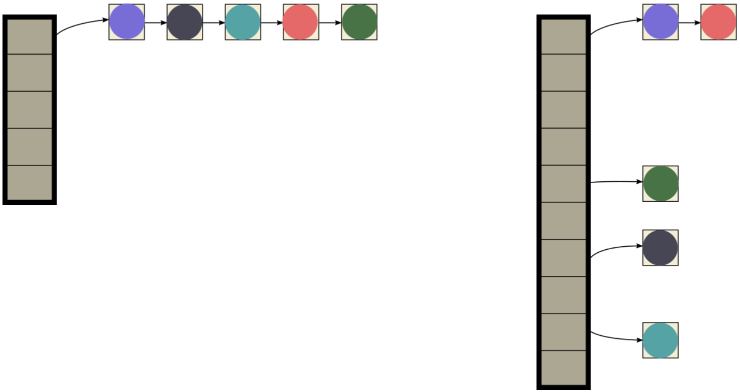
But:

- Cannot serve allocations bigger than chunk size
- High waste through internal fragmentation if sizes of objects vary a lot

Pool Allocator - STL containers

- `vector` only if chunk sizes match vector size
- `list` and `map` are a perfect fit, as the size of each node is known beforehand (though this knowledge is implementation-specific)
- Similar for `deque`
- `unordered_map` again deserves a closer look...

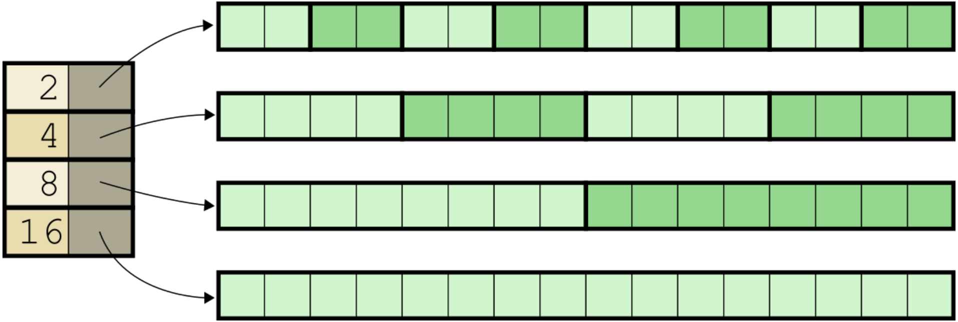
unordered_map



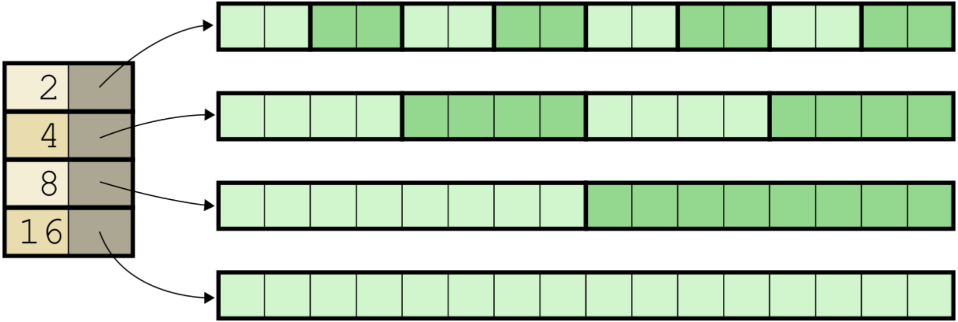
But what if I do need different sizes?

$\longrightarrow$ Multipool Allocator

Multipool Allocator

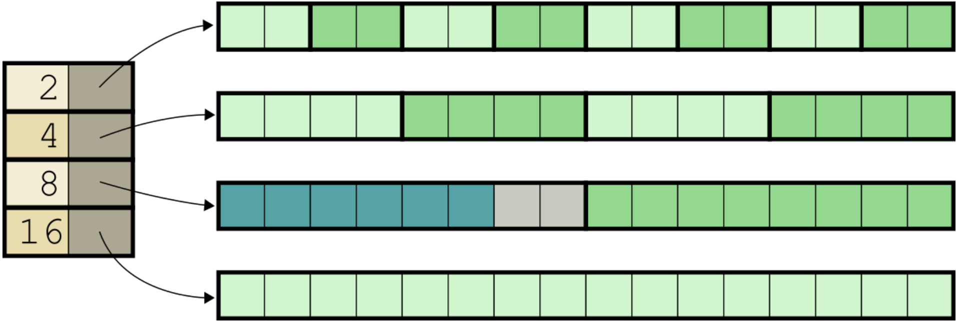


Multipool Allocator



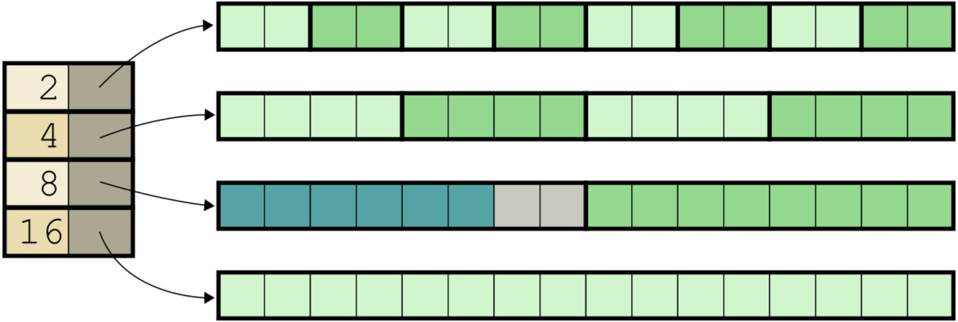

```
auto p1 = multipool.allocate(6);
```

Multipool Allocator



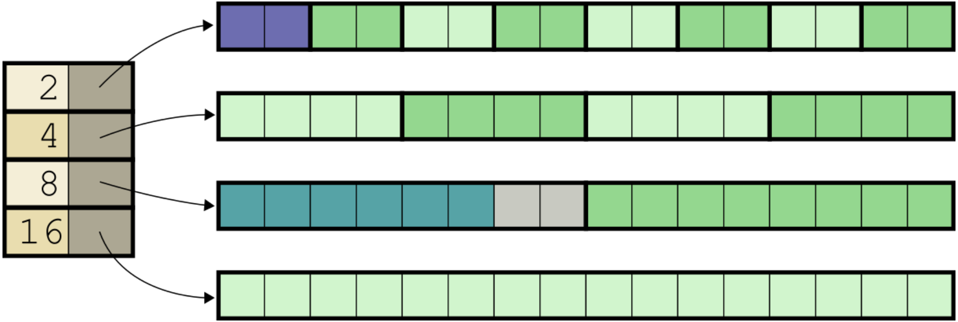

```
auto p1 = multipool.allocate(6);
```

Multipool Allocator




```
auto p1 = multipool.allocate(6);
auto p2 = multipool.allocate(2);
```

Multipool Allocator




```cpp
auto p1 = multipool.allocate(6);
auto p2 = multipool.allocate(2);
```

Multipool Allocator

- Very powerful allocator
- Runtime is still deterministic if number of pools is known beforehand
- Maximum amount of waste through internal fragmentation can be controlled precisely
- Difficult to set up: How many pools do I need? What chunk sizes? What pool sizes?
- Solid building block for a general purpose allocator

Allocator support in C++

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

v.push_back(...);
```

Allocator support in C++

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

v.push_back(...);
```

This is not the class allocating the memory.

Allocator support in C++

Historically, C++ used Allocators to abstract over different models of addressing memory.
As such, *Allocators* in C++ are "stateless".

[2] Arthur O'Dwyer - An Allocator is a Handle to the Heap

Allocator support in C++

Historically, C++ used Allocators to abstract over different models of addressing memory.
As such, *Allocators* in C++ are "stateless".

In C++ an Allocator is merely a handle to a *memory resource*.[2]

[2] Arthur O'Dwyer - An Allocator is a Handle to the Heap

Allocator support in C++

```
std::pmr::memory_resource& mr = ...;
std::vector<T, std::pmr::polymorphic_allocator> v(&mr);

v.push_back(...);
```

Allocator support in C++

```
std::pmr::memory_resource& mr = ...;
std::vector<T, std::pmr::polymorphic_allocator> v(&mr);

v.push_back(...);
```

- Enable custom allocators for the object.

Allocator support in C++

```
std::pmr::memory_resource& mr = ...;
std::vector<T, std::pmr::polymorphic_allocator> v(&mr);

v.push_back(...);
```

- Enable custom allocators for the object.
- Pass a `memory_resource` to handle allocation/deallocation.

Allocator support in C++

```
std::pmr::memory_resource& mr = ...;

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

v.push_back(...);
```

- Enable custom allocators for the object.
- Pass a `memory_resource` to handle allocation/deallocation.

C++ Memory Resources[3]

- `std::pmr::memory_resource` - Abstract base class for all resources that can be wrapped in a `std::pmr::polymorphic_allocator`
- `std::pmr::new_delete_resource()` - Global allocator
- `std::pmr::monotonic_buffer_resource` - Monotonic allocator
- `std::pmr::unsynchronized_pool_resource`/ `synchronized_pool_resource` - Multipool
- `std::pmr::null_memory_resource()` - Allocation always fails

[3] Pablo Halpern - Allocators: The Good Parts

Chaining

```
explicit monotonic_buffer_resource(
  std::pmr::memory_resource* upstream);
```

Each `memory_resource` has an upstream counterpart.

If the resource runs out of memory, it tries to allocate more memory from upstream.

Chaining

```
// fixed size buffer
std::aligned_storage_t<42> buffer;
std::pmr::monotonic_buffer_resource alloc(&buffer, 42,
    std::pmr::null_memory_resource()
);
```

Chaining

```
// fixed size buffer
std::aligned_storage_t<42> buffer;
std::pmr::monotonic_buffer_resource alloc(&buffer, 42,
    std::pmr::null_memory_resource()
);

//dynamically growing
std::aligned_storage_t<42> buffer;
std::pmr::monotonic_buffer_resource alloc(&buffer, 42,
    std::pmr::new_delete_resource()
);
```

Chaining

Possible uses of Chaining:

- Fixed-size vs. dynamic storage for allocators
- Customization of error-handling
- Combination of different allocation strategies
- Injection points for special purpose allocators for debugging and profiling

There's no universal interface for allocators

- Are size and alignment parameters passed to `deallocate`?
- Is `realloc` supported?
- How are out-of-memory errors reported?
- Is extended alignment supported?
- What is the return value for an allocation of size `0`?
- Different memory regions for internal data structures and allocated memory?

Don't underestimate the global allocator

- Competitive performance in the general case
- Security features (ASLR, secure erase of freed memory)
- Debugging & Profiling (Valgrind, Windows Debug Runtime)
- Cache Coloring

Local allocators are no free lunch!

Wrapping up

- No one-size-fits-all — Each allocator has its Achilles heel
- Global allocator is a good solution for the general case
- But you can do better with special allocators for special use cases, in terms of performance[4] as well as reliability
- C++ has good support for local allocators, but the terminology is a bit off
- Different libraries have different concepts of allocators
- No free lunch: You need to understand your use case before you can chose the right allocator

[4] John Lakos - Local (Arena) Allocators

Thanks for your attention.

ComicSansMS / @DerGhulbus