Lightweight Object Persistence With Modern C++

Bob Steagall
CppCon 2016

Relocatable Heaps In

Lightweight Object Persistence With Modern C++

Bob Steagall
CppCon 2016

Overview

- Goals
  - Describe a way of thinking about allocator design that may be helpful
  - Outline one solution to the problem of object persistence

Overview

- Goals
  - Describe a way of thinking about allocator design that may be helpful
  - Outline one solution to the problem of object persistence
- Anti-Goals
  - !(Allocator tutorial)
  - !(Discuss improvements to standard allocators)
  - !(A complete OTS framework for object persistence)

Problem Context and Statement

- I have a set of types
  - Have container data members, possibly nested
  - Have a large number of objects (> 10 GB)
  - Have time-consuming construction / copy / traversal operations

Problem Context and Statement

- I have a set of types
  - Have container data members, possibly nested
  - Have a large number of objects (> 10 GB)
  - Have time-consuming construction / copy / traversal operations
- I want to
  - Save to persistent storage
  - Transmit somewhere else

Problem Context and Statement

- I have a set of types
  - Have container data members, possibly nested
  - Have a large number of objects (> 10 GB)
  - Have time-consuming construction / copy / traversal operations
- I want to
  - Save to persistent storage
  - Transmit somewhere else
- How can I accomplish these feats?

The Obvious Solution - Serialization

- Step 1: Iterate over and **serialize** source objects into some intermediate format
  - *JSON / YAML / XML / protocol buffers / proprietary*
  - Purpose: save **important** object state

The Obvious Solution - Serialization

- Step 1: Iterate over and **serialize** source objects into some intermediate format
  - *JSON / YAML / XML / protocol buffers / proprietary*
  - Purpose: save **important** object state

- Step 2: **De-serialize** the intermediate format into destination objects
  - Purpose: recover **important** state
  - Result: each destination object is semantically identical to its corresponding source object

The Obvious Solution - Serialization

- Step 1: Iterate over and **serialize** source objects into some intermediate format
  - *JSON / YAML / XML / protocol buffers / proprietary*
  - Purpose: save **important** object state

- Step 2: **De-serialize** the intermediate format into destination objects
  - Purpose: recover **important** state
  - Result: each destination object is semantically identical to its corresponding source object

- Traversal-based serialization (TBS)

The Intermediate Format

- The intermediate format describes a schema

The Intermediate Format

- The intermediate format describes a schema

- The intermediate format can provide several forms of **independence**
  - Architectural independence
    - Byte ordering, class member layout, address space layout (e.g., x86_64 to PPC)
  - Representational independence
    - Intra-language (e.g., `list<vector<char>>` to `list<string>`)
    - Inter-language (e.g., `List<String>` to `list<string>`)
  - Positional independence
    - Important state is preserved when destination object exists at different address

Possible Traversal-Based Serialization Costs

- In C++, per-type code must be written or generated
  - Traverse source objects and render them to intermediate format
  - Parse the intermediate format and reconstruct destination objects
  - This code can become complex and fragile

Possible Traversal-Based Serialization Costs

- In C++, per-type code must be written or generated
  - Traverse source objects and render them to intermediate format
  - Parse the intermediate format and reconstruct destination objects
  - This code can become complex and fragile
- Time – entire stream must be read end-to-end

Possible Traversal-Based Serialization Costs

- In C++, per-type code must be written or generated
  - Traverse source objects and render them to intermediate format
  - Parse the intermediate format and reconstruct destination objects
  - This code can become complex and fragile
- Time – entire stream must be read end-to-end
- Space – many common intermediate formats are verbose

Possible Traversal-Based Serialization Costs

- In C++, per-type code must be written or generated
  - Traverse source objects and render them to intermediate format
  - Parse the intermediate format and reconstruct destination objects
  - This code can become complex and fragile
- Time – entire stream must be read end-to-end
- Space – many common intermediate formats are verbose
- Private implementation details might be exposed
- Encapsulation might be violated

Traversal-Based Serialization

- Point: Universal technique for implementing object persistence

- Counterpoint: Can be expensive to implement and maintain

Most of you are familiar with the virtues of a programmer. There are three, of course: laziness, impatience, and hubris.

– Larry Wall

Revised Problem Statement

- Suppose I don’t need architectural or representational independence
  - Source and destination platforms are the same
  - Class member layout is the same on the source and destination platforms
  - I can use the same object code on the source and destination platforms

Revised Problem Statement

- Suppose I don’t need architectural or representational independence
  - Source and destination platforms are the same
  - Class member layout is the same on the source and destination platforms
  - I can use the same object code on the source and destination platforms

- Implement object persistence
  - That does not require per-type serialization/de-serialization code
  - That allows me to persist standard containers and strings
  - That uses fast binary I/O, like `write()`/`read()` or `send()`/`recv()`

One Idea – Relocatable Heaps

- A heap is relocatable **if**
  - It can be serialized and de-serialized with simple binary I/O

and, after de-serialization at a different address,

  - The heap continues to function correctly, **and**
  - The heap’s contents continue to function correctly

One Idea – Relocatable Heaps

- A heap is relocatable **if**
  - It can be serialized and de-serialized with simple binary I/O

  and, after de-serialization at a different address,
  - The heap continues to function correctly, **and**
  - The heap's contents continue to function correctly

- Every object in a relocatable heap must be of a **relocatable type**

Relocatable Type Requirements

- A type is relocatable **if**
  - It is serializable by writing raw bytes (`write()` / `memcpy()`), **and**
  - It is de-serializable by reading raw bytes (`read()` / `memcpy()`), **and**
  - A destination object of that type is semantically identical to its corresponding source object, regardless of that object's address in the destination process

Relocatable Type Requirements

- A type is relocatable **if**
  - It is serializable by writing raw bytes (`write()` / `memcpy()`), **and**
  - It is de-serializable by reading raw bytes (`read()` / `memcpy()`), **and**
  - A destination object of that type is semantically identical to its corresponding source object, regardless that object's address in the destination process
- These types are relocatable
  - Integer types
  - Floating point types
  - A POD type that ultimately contains only integer and floating-point types

Relocatable Type Requirements

- These types are not relocatable:
  - Ordinary pointers to data
    - *Referenced data may exist at a different address*
  - Pointers to member functions, static member functions, or free functions
    - *Referenced object code will likely exist a different address*
  - Types with virtual functions
    - *vtables will likely exist a different address*
  - Types, or values of relocatable types, that express process dependence
    - *File descriptors, Windows HANDLEs, etc.*
    - *By definition, process-dependent “handles” are meaningless outside their own process*

Relocatable Heaps in Practice

- Design
  - Provide methods to initialize, serialize, and de-serialize the heap
  - Provide methods to store and access a **master object** residing in the heap

Relocatable Heaps in Practice

- Design
  - Provide methods to initialize, serialize, and de-serialize the heap
  - Provide methods to store and access a **master object** residing in the heap
- Source side
  - Ensure that relocatable type requirements are observed by all contents
  - Allocate everything to be persistent from the heap
  - Serialize the heap

Relocatable Heaps in Practice

- Design
  - Provide methods to initialize, serialize, and de-serialize the heap
  - Provide methods to store and access a **master object** residing in the heap
- Source side
  - Ensure that relocatable type requirements are observed by all contents
  - Allocate everything to be persistent from the heap
  - Serialize the heap
- Destination side
  - De-serialize the heap
  - Obtain access to the heap's contents through the master object

Eschew obfuscation!

Thinking (Slightly) Differently About Memory Allocation

- Structural Management
  - Addressing Model
  - Storage Model
  - Pointer Interface
  - Allocation Strategy

Thinking (Slightly) Differently About Memory Allocation

- Structural Management
  - Addressing Model
  - Storage Model
  - Pointer Interface
  - Allocation Strategy

- Concurrency Management
  - Thread Safety
  - Transaction Safety

Concept – Addressing Model

- Policy type that implements primitive addressing operations
  - Analogous to `void*`
  - Convertible to `void*`

Concept – Addressing Model

- Policy type that implements primitive addressing operations
  - Analogous to `void*`
  - Convertible to `void*`
- Internally, the addressing model defines
  - The bits used to represent an address
  - How an address is computed
  - How memory is arranged

Concept – Addressing Model

- Policy type that implements primitive addressing operations
  - Analogous to `void*`
  - Convertible to `void*`
- Internally, the addressing model defines
  - The bits used to represent an address
  - How an address is computed
  - How memory is arranged
- Representations
  - Ordinary pointer `void*` (aka natural pointer)
  - Synthetic void pointer (aka fancy pointer, pointer-like type) or other UDT

Concept - Storage Model

- Policy type that manages segments
  - Interacts with an external source of memory to borrow and return segments
  - Provides an interface to segments in terms of the addressing model
  - Lowest-level allocation

Concept - Storage Model

- Policy type that manages segments
  - Interacts with an external source of memory to borrow and return segments
  - Provides an interface to segments in terms of the addressing model
  - Lowest-level allocation
- **Segment**: a region of memory that has been provided to the storage model by some external source
  - `brk()` / `sbrk()` — Unix private memory
  - `VirtualAlloc()` / `HeapAlloc()` — Windows private memory
  - `shmget()` / `shmat()` — System V shared memory
  - `shm_open()` / `mmap()` — POSIX shared memory
  - `CreateFileMapping()` / `MapViewOfFile()` — Windows shared memory

Concept - Pointer Interface

- Policy type that wraps the addressing model to emulate a pointer to data
  - Analogous to `T*`
  - Provides (enough) pointer syntax
  - Is convertible "in the right direction" to ordinary pointers
  - Is convertible "in the right direction" to other pointer interface types

Concept - Pointer Interface

- Policy type that wraps the addressing model to emulate a pointer to data
  - Analogous to `T*`
  - Provides (enough) pointer syntax
  - Is convertible "in the right direction" to ordinary pointers
  - Is convertible "in the right direction" to other pointer interface types

- Representations
  - Ordinary pointer `T*` (aka, natural pointer)
  - Synthetic pointer UDT (aka, fancy pointer, pointer-like type)

Concept - Allocation Strategy

- Policy type that manages the process of allocating memory for clients
  - Requests segment allocation/deallocation from the storage model
  - Interacts with segments in terms of the addressing model
  - Divides segments into chunks
  - Provides chunks to the client in terms of the pointer interface

Concept - Allocation Strategy

- Policy type that manages the process of allocating memory for clients
  - Requests segment allocation/deallocation from the storage model
  - Interacts with segments in terms of the addressing model
  - Divides segments into chunks
  - Provides chunks to the client in terms of the pointer interface

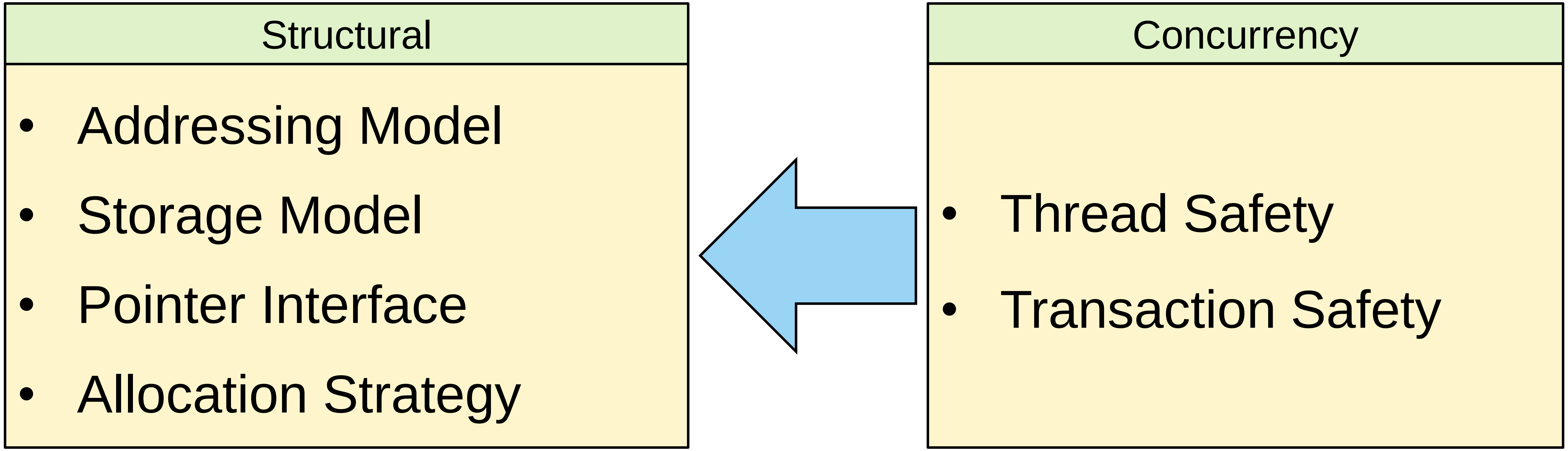
- **Chunk:** A region of memory carved out of a segment to be used by an allocator's client

Concepts – Thread Safety and Transaction Safety

- Thread safety – correct operation with multiple threads/processes
- Transaction safety – supporting allocate/commit/rollback semantics

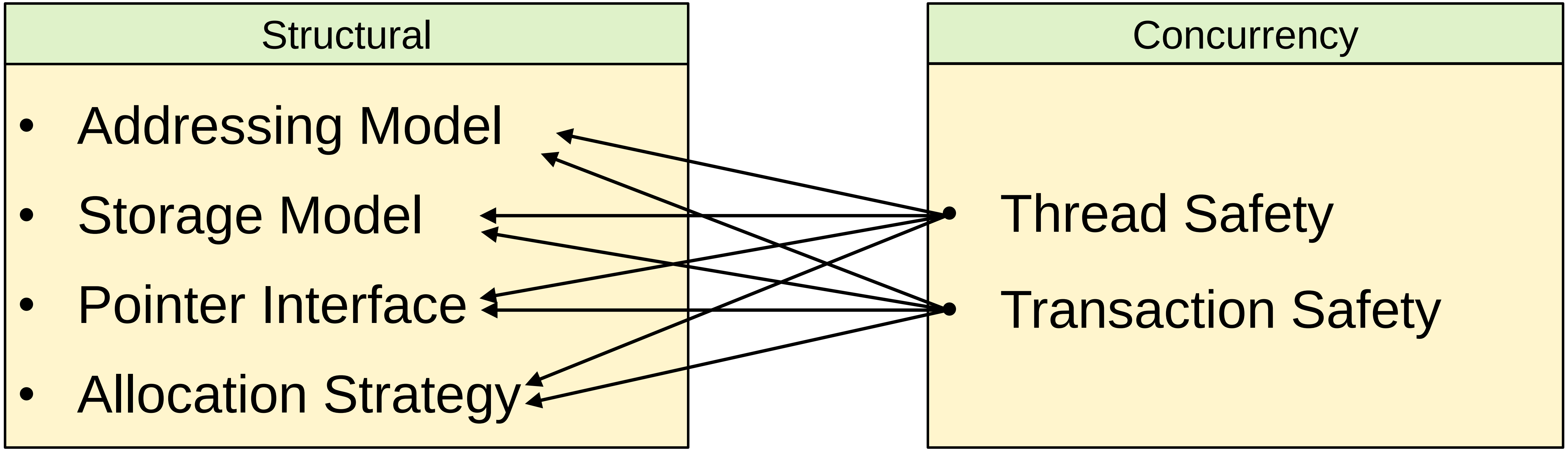
Concepts – Thread Safety and Transaction Safety

- Thread safety – correct operation with multiple threads/processes
- Transaction safety – supporting allocate/commit/rollback semantics



Concepts – Thread Safety and Transaction Safety

- Thread safety – correct operation with multiple threads/processes
- Transaction safety – supporting allocate/commit/rollback semantics



How Is `std::allocator<T>` Characterized By This Framework?

- Addressing Model: **`void*`**
- Storage Model: **`::operator new()`**
- Pointer Interface: **`T*`**
- Allocation Strategy: **`::operator new()`**
- Thread Safety: **`::operator new()`**
- Transaction Safety: *none*

Other Allocators

- dlmalloc
- jemalloc
- tcmalloc
- Hoard
- VMem

- Addressing Model: `void*`
- Storage Model: ***`abc`***
- Pointer Interface: `T*`
- Allocation Strategy: ***`uvw`***
- Thread Safety: ***`xyz`***
- Transaction Safety: none

Allocators Before C++11

- 14882:2003 / 20.1.5.4

  Implementations of containers described in this International Standard are permited to assume that their Allocator template parameter meets the following two additional requirements beyond those in Table 32.

  — All instances of a given allocator type are required to be interchangeable and always compare equal to each other.

  — The typedef members pointer, const_pointer, size_type, and difference_type are required to be T*, T const*, size_t, and ptrdiff_t, respectively.

- 14882:2003 / 20.1.5.5

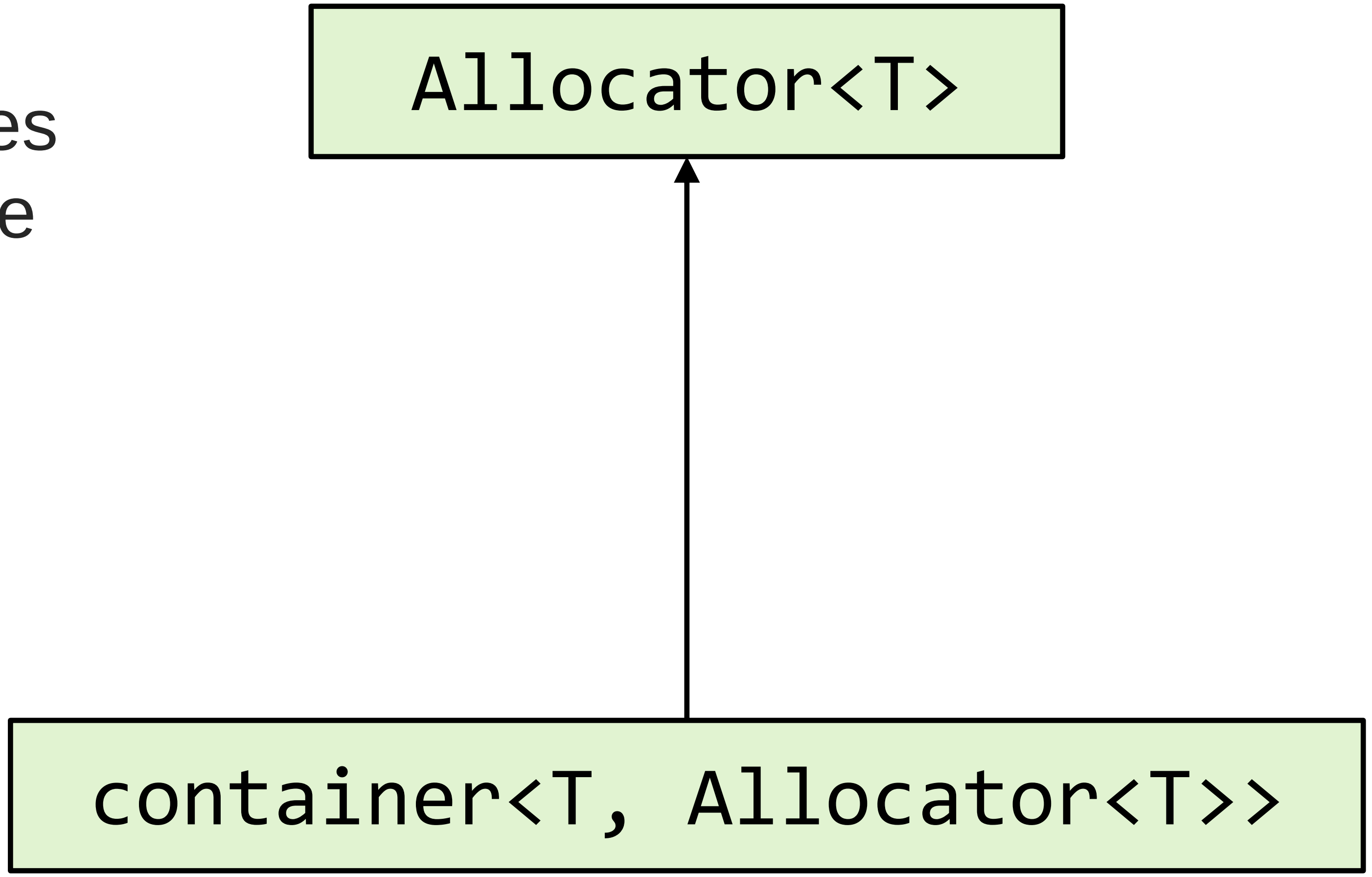
  Implementors are encouraged to supply libraries that can accept allocators that encapsulate more general memory models and that support non-equal instances. In such implementations, any requirements imposed on allocators by containers beyond those requirements that appear in Table 32, and the semantics of containers and algorithms when allocator instances compare non-equal, are implementation-defined.

Allocators Before C++11

Containers obtain their allocation services and part of their view of memory from the allocator template argument

But, can assume that:

```
using pointer = T*
using const_pointer = T const*
```

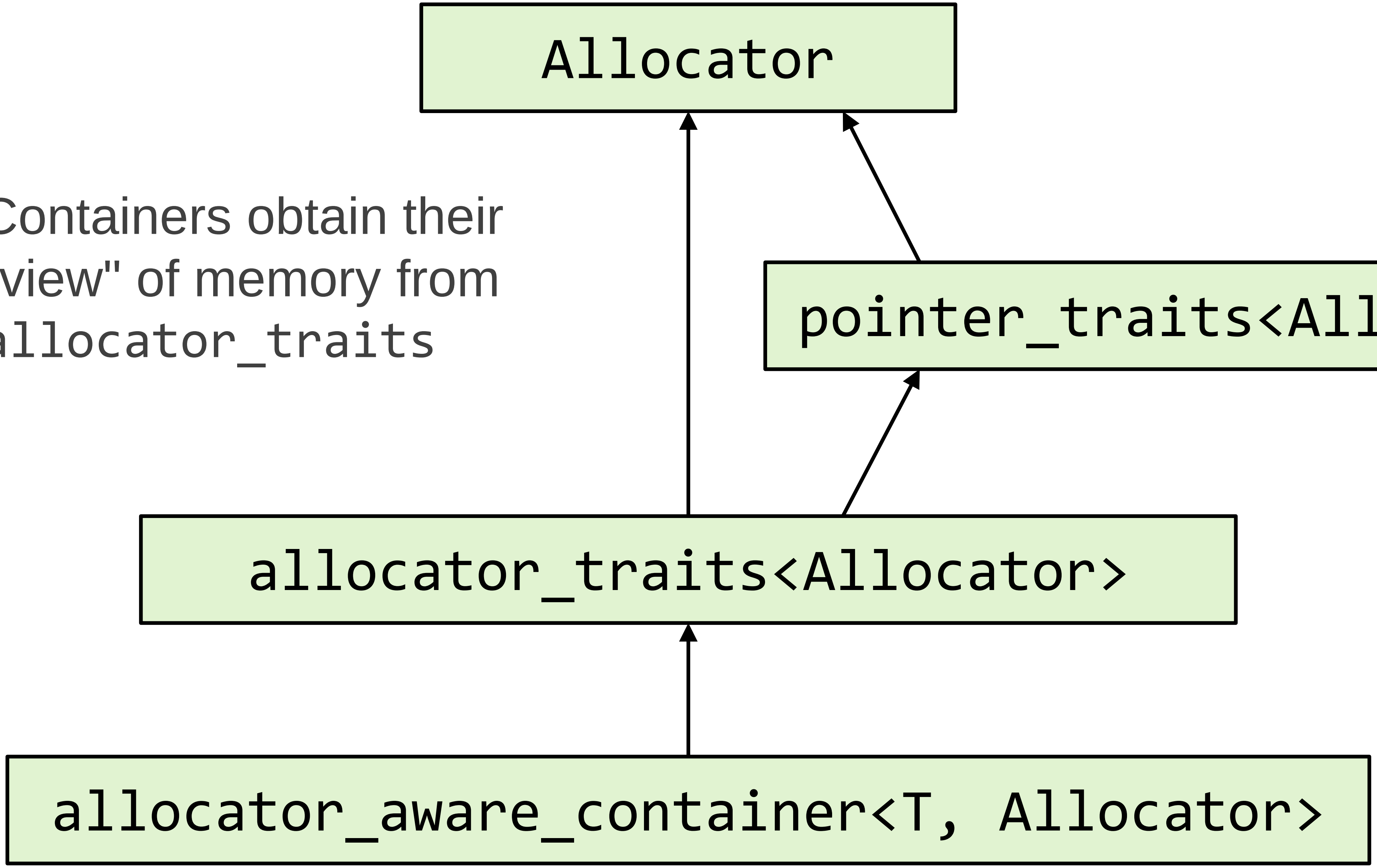


Allocators After C++11

- Paragraphs 20.1.5.4 / 5 deleted!
- New requirements to improve allocators (C++14)
  - *nullablepointer.requirements* (17.6.3.3)
    - Pointer-like type that supports null values
  - *allocator.requirements* (17.6.3.5)
    - Defines allocator and relationship to allocator traits
  - *pointer.traits* (20.7.3)
    - Describes a uniform interface to pointer-like types
  - *allocator.traits* (20.7.8)
    - Describes uniform interface to allocator types
  - *container.requirements.general* (23.2.1, Table 99)
    - Defines allocator-aware container requirements

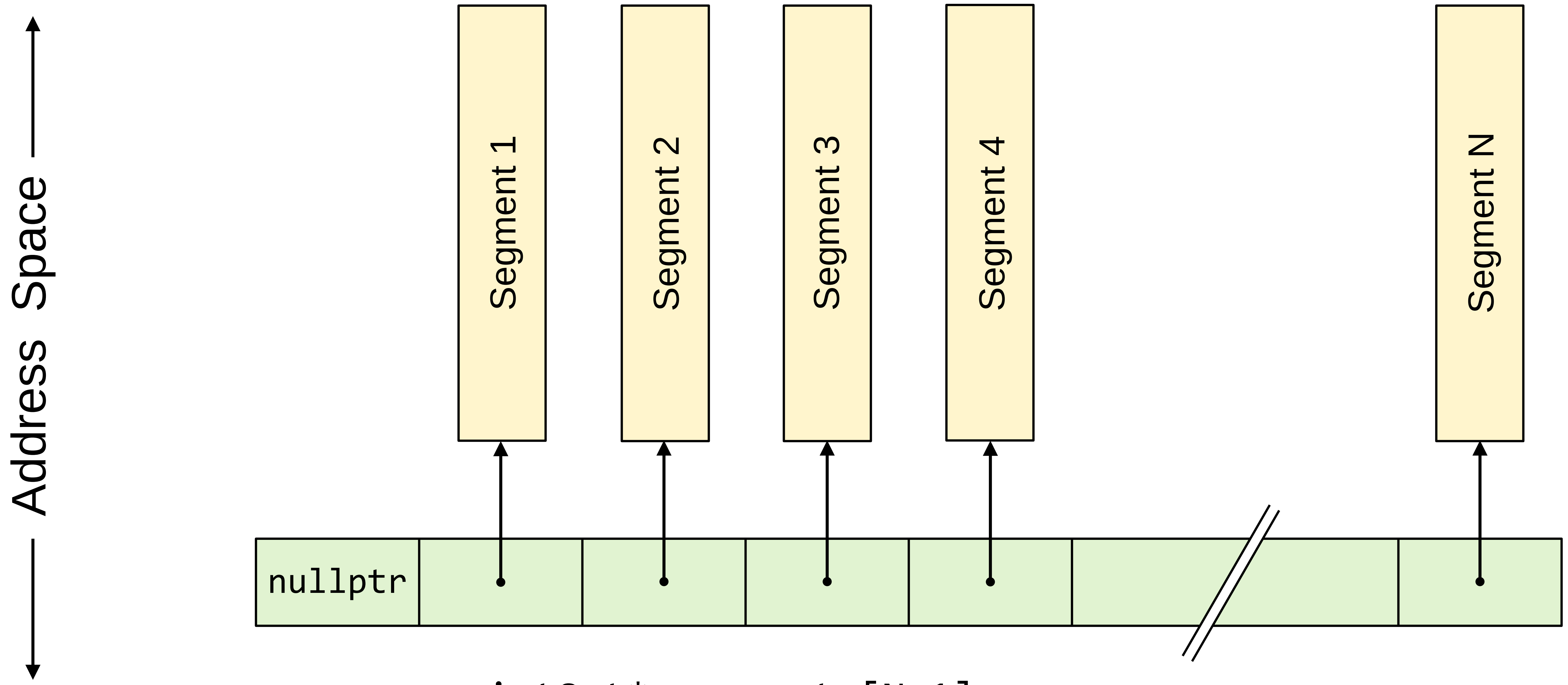
Allocators After C++11

Containers obtain their "view" of memory from `allocator_traits`



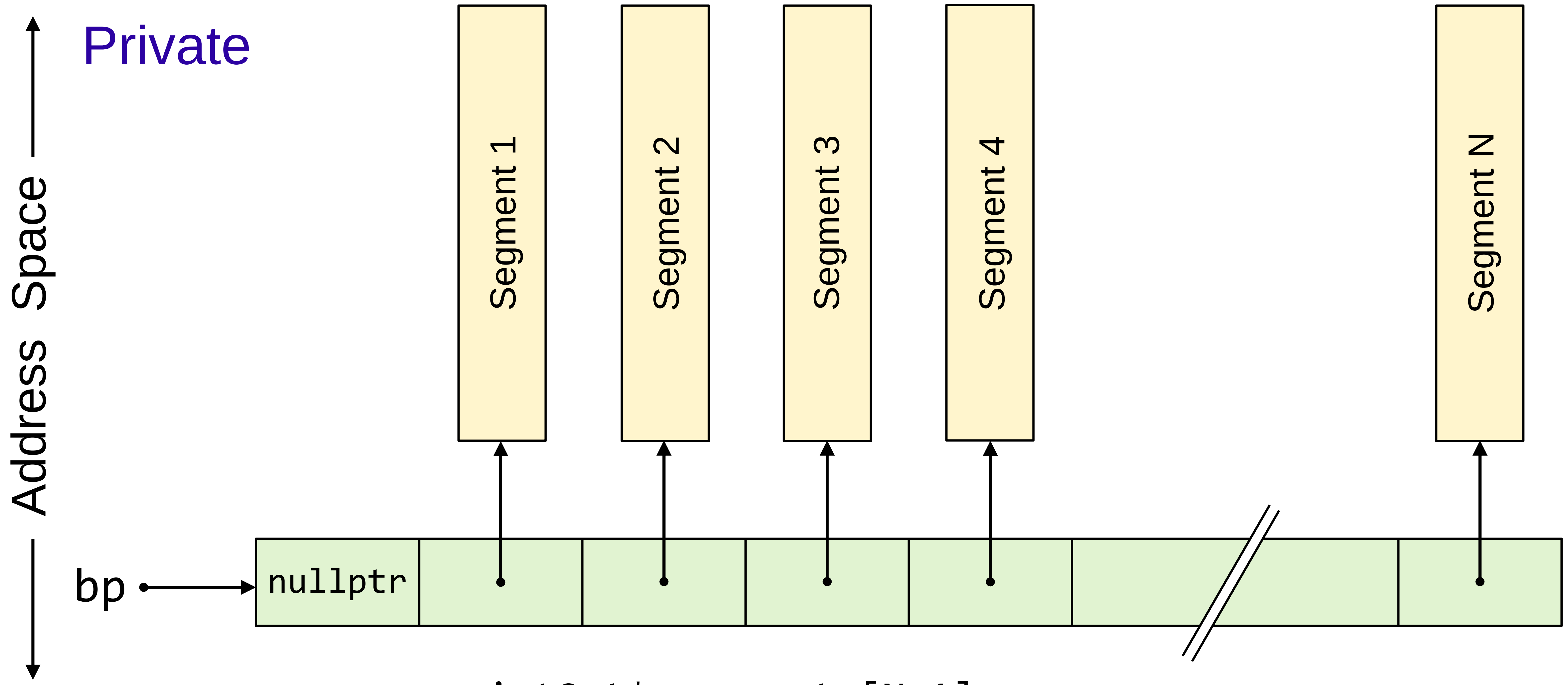
No more speed I'm almost there…

Addressing Model



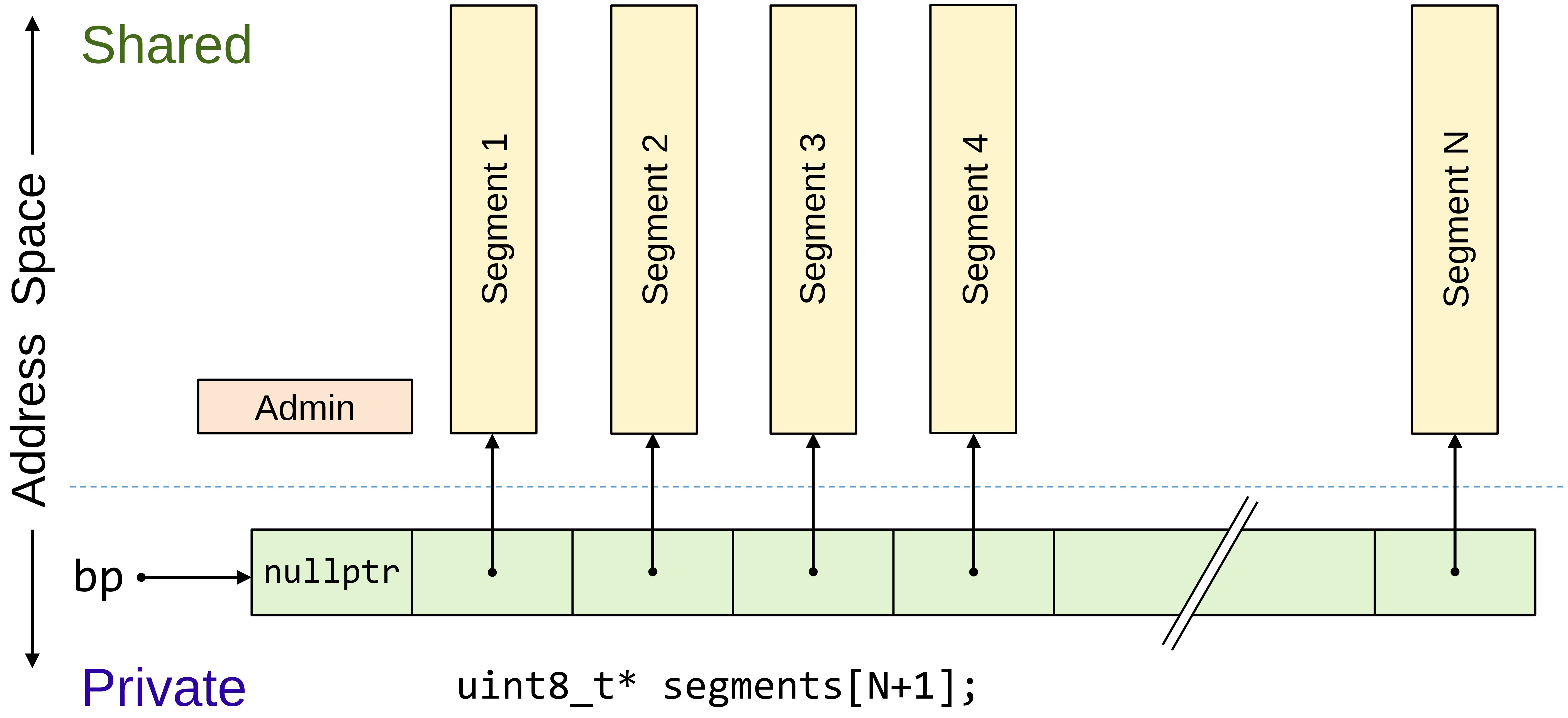

```
uint8_t* segments[N+1];
```

Example Addressing and Storage Models – Private Segments




```
uint8_t* segments[N+1];
```

Example Addressing and Storage Models – Shared Segments




```
uint8_t* segments[N+1];
```

Example Addressing Model

```
template<typename SM>
class segmented_addressing_model
{
  public:
    using size_type       = std::size_t;
    using difference_type = std::ptrdiff_t;

    ~segmented_addressing_model() = default;

    segmented_addressing_model() noexcept = default;
    segmented_addressing_model(segmented_addressing_model&&) noexcept = default;
    segmented_addressing_model(segmented_addressing_model const&) noexcept = default;
    segmented_addressing_model(std::nullptr_t) noexcept;

    segmented_addressing_model& operator =(segmented_addressing_model&&) noexcept = default;
    segmented_addressing_model& operator =(segmented_addressing_model const&) noexcept = default;
    segmented_addressing_model& operator =(std::nullptr_t) noexcept;

    ...
};
```

Example Addressing Model

```
template<typename SM>
class segmented_addressing_model
{
    ...
    void*       address() const noexcept;
    size_type   offset() const noexcept;
    size_type   segment() const noexcept;

    bool        equals(std::nullptr_t) const noexcept;
    bool        equals(void const* p) const noexcept;
    bool        equals(segmented_addressing_model const& other) const noexcept;

    //- less_than() and greater_than() go here
    ...

    void        assign_from(void const* p);
    void        decrement(difference_type dec) noexcept;
    void        increment(difference_type inc) noexcept;

    ...
};
```

Example Addressing Model

```
template<typename SM>
class segmented_addressing_model
{
    ...
  private:
    friend  SM;
    segmented_addressing_model(size_type segment, size_type offset) noexcept;

    enum : uint64_t { offset_mask = 0xFFFFFFFFFFFFFFFF >> 16 };

    struct addr_bits
    {
        uint16_t    m_word1;
        uint16_t    m_word2;
        uint16_t    m_word3;
        uint16_t    m_segment;
    };

    ...
};
```

Example Addressing Model

```
template<typename SM>
class segmented_addressing_model
{
    ...

    struct addr_bits
    {
        uint16_t    m_word1;
        uint16_t    m_word2;
        uint16_t    m_word3;
        uint16_t    m_segment;
    };

    union
    {
        uint64_t    m_addr;
        addr_bits   m_bits;
    };
};
```

Example Addressing Model

```
template<typename SM>
class segmented_addressing_model
{
    ...

    struct addr_bits
    {
        uint16_t    m_word1;
        uint16_t    m_word2;
        uint16_t    m_word3;
        uint16_t    m_segment;
    };

    union
    {
        uint64_t    m_addr;
        addr_bits   m_bits;
    };
};
```

Hello, DOS!

Example Storage Model

```
class segmented_private_storage_model
{
  public:
    using difference_type  = std::ptrdiff_t;
    using size_type        = std::size_t;
    using addressing_model = segmented_addressing_model<segmented_private_storage_model>;

    enum : size_type
    {
        max_segments = 256,
        max_size     = 1u << 22
    };

    static  void    allocate_segment(size_type segment, size_type size = max_size);
    static  void    deallocate_segment(size_type segment);
    static  void    swap_buffers();

    static  uint8_t*            segment_address(size_type segment) noexcept;
    static  addressing_model    segment_pointer(size_type segment, size_type offset=0) noexcept;
    static  size_type           segment_size(size_type segment) noexcept;

    ...
};
```

Example Storage Model

```
class segmented_private_storage_model
{
  public:
    ...

    static  constexpr   size_type   first_segment();
    static  constexpr   size_type   max_segment_count();
    static  constexpr   size_type   max_segment_size();

  private:
    friend class segmented_addressing_model<segmented_private_storage_model>;

    static  uint8_t*            sm_segment_addr[max_segments + 2];
    static  addressing_model    sm_segment_data[max_segments + 2];
    static  size_type           sm_segment_size[max_segments + 2];
    static  uint8_t*            sm_shadow_addr[max_segments + 2];
};
```

Example Addressing Model + Storage Model

```
    enum : uint64_t { offset_mask = 0x0000FFFFFFFFFFFF };
    struct addr_bits
    {
        uint16_t    m_word1;
        uint16_t    m_word2;
        uint16_t    m_word3;
        uint16_t    m_segment;
    };
    union
    {
        uint64_t    m_addr;
        addr_bits   m_bits;
    };
    ...
    static  uint8_t*    sm_segment_addr[max_segments + 2];

template<typename SM> inline void*
segmented_addressing_model<SM>::address() const noexcept
{
    return SM::sm_segment_addr[m_bits.m_segment] + (m_addr & offset_mask);
}
```

Example Addressing Model + Storage Model

```
    enum : uint64_t { offset_mask = 0x0000FFFFFFFFFFFF };
    struct addr_bits
    {
        uint16_t    m_word1;
        uint16_t    m_word2;
        uint16_t    m_word3;
        uint16_t    m_segment;
    };
    union
    {
        uint64_t    m_addr;
        addr_bits   m_bits;
    };
    ...
    static  uint8_t*    sm_segment_addr[max_segments + 2];

template<typename SM> inline void*
segmented_addressing_model<SM>::address() const noexcept
{
    return SM::sm_segment_addr[m_bits.m_segment] + (m_addr & offset_mask);
}
```

Example Addressing Model + Storage Model

```
    enum : uint64_t { offset_mask = 0x0000FFFFFFFFFFFF };
    struct addr_bits
    {
        uint16_t    m_word1;
        uint16_t    m_word2;
        uint16_t    m_word3;
        uint16_t    m_segment;
    };
    union
    {
        uint64_t    m_addr;
        addr_bits   m_bits;
    };
    ...
    static  uint8_t*    sm_segment_addr[max_segments + 2];

template<typename SM> inline void*
segmented_addressing_model<SM>::address() const noexcept
{
    return SM::sm_segment_addr[m_bits.m_segment] + (m_addr & offset_mask);
}
```

Example Addressing Model + Storage Model

```
    enum : uint64_t { offset_mask = 0x0000FFFFFFFFFFFF };
    struct addr_bits
    {
        uint16_t    m_word1;
        uint16_t    m_word2;
        uint16_t    m_word3;
        uint16_t    m_segment;
    };
    union
    {
        uint64_t    m_addr;
        addr_bits   m_bits;
    };
    ...
    static  uint8_t*    sm_segment_addr[max_segments + 2];

template<typename SM> inline void*
segmented_addressing_model<SM>::address() const noexcept
{
    return SM::sm_segment_addr[m_bits.m_segment] + (m_addr & offset_mask);
}
```

Example Pointer Interface

```
template<class T, class AM>
class synthetic_pointer
{
  public:
    [ Canonical Member Functions ]

    [ Other Constructors ]

    [ Other Assignment Operators ]

    [ Conversion Operators ]

    [ Dereferencing and Pointer Arithmetic ]

    [ Helpers to Support Library Requirements ]

    [ Helpers to Support Comparison Operators ]

  private
    [ Data Members ]
};
```

Example Pointer Interface – Traits for SFINAE

```
struct synthetic_pointer_traits
{
    template<class From, class To>
    using implicitly_convertible =
        typename std::enable_if<std::is_convertible<From*, To*>::value, bool>::type;

    template<class From, class To>
    using explicit_conversion_required =
        typename std::enable_if<!std::is_convertible<From*, To*>::value, bool>::type;

    template<class T1, class T2>
    using implicitly_comparable =
        typename std::enable_if<std::is_convertible<T1*, T2 const*>::value ||
                                std::is_convertible<T2*, T1 const*>::value, bool>::type;
};
```

Example Pointer Interface – Nested Aliases

```
template<class T, class AM>
class synthetic_pointer
{
  public:
    template<class U>
    using rebind = synthetic_pointer<U, AM>;

    using difference_type   = typename AM::difference_type;
    using size_type         = typename AM::size_type;
    using element_type      = T;
    using value_type        = T;
    using reference         = T&;
    using pointer           = synthetic_pointer;

    using iterator_category = std::random_access_iterator_tag;

    ...
};
```

Example Pointer Interface – Canonical Member Functions

```
template<class T, class AM>
class synthetic_pointer
{
    ...

    ~synthetic_pointer() noexcept = default;

    synthetic_pointer() noexcept = default;
    synthetic_pointer(synthetic_pointer&&) noexcept = default;
    synthetic_pointer(synthetic_pointer const&) noexcept = default;

    synthetic_pointer&  operator =(synthetic_pointer&&) noexcept = default;
    synthetic_pointer&  operator =(synthetic_pointer const&) noexcept = default;

    ...
};
```

Example Pointer Interface – Other Constructors

```
template<class T, class AM>
class synthetic_pointer
{
    ...

    synthetic_pointer(AM am);
    synthetic_pointer(std::nullptr_t);

    template<class U, synthetic_pointer_traits::implicitly_convertible<U, T> = true>
    synthetic_pointer(U* p);

    template<class U, synthetic_pointer_traits::implicitly_convertible<U, T> = true>
    synthetic_pointer(synthetic_pointer<U, AM> const& p);

    ...
};
```

Example Pointer Interface – Other Assignment Operators

```
template<class T, class AM>
class synthetic_pointer
{
    ...

    synthetic_pointer&   operator =(std::nullptr_t);

    template<class U, synthetic_pointer_traits::implicitly_convertible<U, T> = true>
    synthetic_pointer&   operator =(U* p);

    template<class U, synthetic_pointer_traits::implicitly_convertible<U, T> = true>
    synthetic_pointer&   operator =(synthetic_pointer<U, AM> const& p);

    ...
};
```

Example Pointer Interface – Conversion Operators

```
template<class T, class AM>
class synthetic_pointer
{
    ...

    explicit    operator bool() const;

    template<class U, synthetic_pointer_traits::implicitly_convertible<T, U> = true>
    operator U* () const;

    template<class U, synthetic_pointer_traits::explicit_conversion_required<T, U> = true>
    explicit    operator U* () const;

    template<class U, synthetic_pointer_traits::explicit_conversion_required<T, U> = true>
    explicit    operator synthetic_pointer<U, AM>() const;

    ...
};
```

Example Pointer Interface

```
template<class T, class AM>
class synthetic_pointer
{
    ...
    T*  operator ->() const;
    T&  operator  *() const;
    T&  operator [](size_type n) const;

    difference_type      operator -(const synthetic_pointer& p) const;
    synthetic_pointer    operator -(difference_type n) const;
    synthetic_pointer    operator +(difference_type n) const;

    synthetic_pointer&   operator ++();
    synthetic_pointer    operator ++(int);
    synthetic_pointer&   operator --();
    synthetic_pointer    operator --(int);
    synthetic_pointer&   operator +=(difference_type n);
    synthetic_pointer&   operator -=(difference_type n);

    ...
};
```

Example Pointer Interface

```
template<class T, class AM>
class synthetic_pointer
{
    ...

    static  synthetic_pointer     pointer_to(element_type& e);

    ...

    bool    equals(std::nullptr_t) const;

    template<class U, synthetic_pointer_traits::implicitly_comparable<T, U> = true>
    bool    equals(U const* p) const;

    template<class U, synthetic_pointer_traits::implicitly_comparable<T, U> = true>
    bool    equals(synthetic_pointer<U, AM> const& p) const;

    //- less_than() and greater_than() go here
    ...
};
```

Example Pointer Interface

```
template<class T, class AM>
class synthetic_pointer
{
    ...

  private:
    template<class OT, class OAM> friend class synthetic_pointer;

    AM      m_addrmodel;
};
```

Example Allocation Strategy

```
template<class SM>
class segmented_test_heap
{
  public:
    using storage_model          = SM;
    using addressing_model       = typename SM::addressing_model;
    using difference_type        = typename SM::difference_type;
    using size_type              = typename SM::size_type;
    using void_pointer           = synthetic_pointer<void, addressing_model>;
    using const_void_pointer     = synthetic_pointer<void const, addressing_model>;

    template<class T>
    using rebind_pointer         = synthetic_pointer<T, addressing_model>;

    size_type       max_size() const;

    void_pointer    allocate(size_type n);
    void            deallocate(void_pointer p);
    static  void    swap_buffers();
};
```

Example Allocator

```
template<class T, class AS>
class rhx_allocator
{
  public:
    using difference_type       = typename AS::difference_type;
    using size_type             = typename AS::size_type;
    using void_pointer          = typename AS::void_pointer;
    using const_void_pointer    = typename AS::const_void_pointer;
    using pointer               = typename AS::template rebind_pointer<T>;
    using const_pointer         = typename AS::template rebind_pointer<T const>;
    using reference             = T&;
    using const_reference       = T const&;
    using value_type            = T;

    template<class U>
    struct rebind
    {
        using other = rhx_allocator<U, AS>;
    };
    ...
```

Example Allocator

```
template<class T, class AS>
class rhx_allocator
{
    ...

    T*          address(reference t) const noexcept;
    T const*    address(const_reference t) const noexcept;
    size_type   max_size() const noexcept;

    pointer     allocate(size_type n);
    pointer     allocate(size_type n, const_void_pointer p);
    void        deallocate(pointer p);
    void        deallocate(pointer p, size_type n);

    template<class U, class... Args>  void    construct(U* p, Args&&... args);
    template<class U>                 void    destroy(U* p);

  private:
    ...
    AS  m_heap;
};
```

We'll ride the spiral to the end…

Example Program

```
//- demo.cpp
//
#include <iostream>
#include <list>
#include <map>
#include <string>

#include "segmented_addressing_model.h"
#include "segmented_private_storage_model.h"
#include "synthetic_pointer_interface.h"
#include "segmented_test_heap.h"
#include "rhx_allocator.h"

using namespace std;

using test_heap = segmented_test_heap<segmented_private_storage_model>;

template<class T> using test_allocator    = rhx_allocator<T, test_heap>;
template<class C> using test_string       = basic_string<C, char_traits<C>, test_allocator<C>>;
template<class T> using test_list         = list<T, test_allocator<T>>;
template<class K, class V> using test_map = map<K, V, less<K>, test_allocator<pair<K const, V>>>;
```

Example Program

```
//- demo.cpp
//
#include <iostream>
#include <list>
#include <map>
#include <string>

#include "segmented_addressing_model.h"
#include "segmented_private_storage_model.h"
#include "synthetic_pointer_interface.h"
#include "segmented_test_heap.h"
#include "rhx_allocator.h"

using namespace std;

using test_heap = segmented_test_heap<segmented_private_storage_model>;

template<class T> using test_allocator    = rhx_allocator<T, test_heap>;
template<class C> using test_string       = basic_string<C, char_traits<C>, test_allocator<C>>;
template<class T> using test_list         = list<T, test_allocator<T>>;
template<class K, class V> using test_map = map<K, V, less<K>, test_allocator<pair<K const, V>>>;
```

Example Program

```
//- demo.cpp
//
#include <iostream>
#include <list>
#include <map>
#include <string>

#include "segmented_addressing_model.h"
#include "segmented_private_storage_model.h"
#include "synthetic_pointer_interface.h"
#include "segmented_test_heap.h"
#include "rhx_allocator.h"

using namespace std;

using test_heap = segmented_test_heap<segmented_private_storage_model>;

template<class T> using test_allocator    = rhx_allocator<T, test_heap>;
template<class C> using test_string       = basic_string<C, char_traits<C>, test_allocator<C>>;
template<class T> using test_list         = list<T, test_allocator<T>>;
template<class K, class V> using test_map = map<K, V, less<K>, test_allocator<pair<K const, V>>>;
```

Example Program

```
//- demo.cpp
//
#include <iostream>
#include <list>
#include <map>
#include <string>

#include "segmented_addressing_model.h"
#include "segmented_private_storage_model.h"
#include "synthetic_pointer_interface.h"
#include "segmented_test_heap.h"
#include "rhx_allocator.h"

using namespace std;

using test_heap = segmented_test_heap<segmented_private_storage_model>;

template<class T> using test_allocator    = rhx_allocator<T, test_heap>;
template<class C> using test_string       = basic_string<C, char_traits<C>, test_allocator<C>>;
template<class T> using test_list         = list<T, test_allocator<T>>;
template<class K, class V> using test_map = map<K, V, less<K>, test_allocator<pair<K const, V>>>;
```

Example Program

```
//- demo.cpp
//
#include <iostream>
#include <list>
#include <map>
#include <string>

#include "segmented_addressing_model.h"
#include "segmented_private_storage_model.h"
#include "synthetic_pointer_interface.h"
#include "segmented_test_heap.h"
#include "rhx_allocator.h"

using namespace std;

using test_heap = segmented_test_heap<segmented_private_storage_model>;

template<class T> using test_allocator    = rhx_allocator<T, test_heap>;
template<class C> using test_string       = basic_string<C, char_traits<C>, test_allocator<C>>;
template<class T> using test_list         = list<T, test_allocator<T>>;
template<class K, class V> using test_map = map<K, V, less<K>, test_allocator<pair<K const, V>>>;
```

Example Program

```
//- demo.cpp
//
#include <iostream>
#include <list>
#include <map>
#include <string>

#include "segmented_addressing_model.h"
#include "segmented_private_storage_model.h"
#include "synthetic_pointer_interface.h"
#include "segmented_test_heap.h"
#include "rhx_allocator.h"

using namespace std;

using test_heap = segmented_test_heap<segmented_private_storage_model>;

template<class T> using test_allocator    = rhx_allocator<T, test_heap>;
template<class C> using test_string       = basic_string<C, char_traits<C>, test_allocator<C>>;
template<class T> using test_list         = list<T, test_allocator<T>>;
template<class K, class V> using test_map = map<K, V, less<K>, test_allocator<pair<K const, V>>>;
```

Example Program

```
void test()
{
    using   demo_map = test_map<test_string<char>, test_list<test_string<char>>>;
    auto    spmap = allocate<demo_map, test_heap>();
    auto    spkey = allocate<test_string<char>, test_heap>();
    auto    spval = allocate<test_string<char>, test_heap>();
    char    key[512], value[512];

    for (int i = 0;  i < 10;  ++i)
    {
        sprintf(key, "this is test key string %d", i);
        spkey->assign(key);

        for (int j = 1;  j <= 5;  ++j)
        {
            sprintf(value, "this is a very, very, very long test value string %d", i*100+j);
            spval->assign(value);
            (*spmap)[*spkey].push_back(*spval);
        }
    }
    ...
```

Example Program

```
void test()
{
    using   demo_map = test_map<test_string<char>, test_list<test_string<char>>>;
    auto    spmap = allocate<demo_map, test_heap>();
    auto    spkey = allocate<test_string<char>, test_heap>();
    auto    spval = allocate<test_string<char>, test_heap>();
    char    key[512], value[512];

    for (int i = 0;  i < 10;  ++i)
    {
        sprintf(key, "this is test key string %d", i);
        spkey->assign(key);

        for (int j = 1;  j <= 5;  ++j)
        {
            sprintf(value, "this is a very, very, very long test value string %d", i*100+j);
            spval->assign(value);
            (*spmap)[*spkey].push_back(*spval);
        }
    }
    ...
```

Example Program

```
void test()
{
    using   demo_map = test_map<test_string<char>, test_list<test_string<char>>>;
    auto    spmap = allocate<demo_map, test_heap>();
    auto    spkey = allocate<test_string<char>, test_heap>();
    auto    spval = allocate<test_string<char>, test_heap>();
    char    key[512], value[512];

    for (int i = 0;  i < 10;  ++i)
    {
        sprintf(key, "this is test key string %d", i);
        spkey->assign(key);

        for (int j = 1;  j <= 5;  ++j)
        {
            sprintf(value, "this is a very, very, very long test value string %d", i*100+j);
            spval->assign(value);
            (*spmap)[*spkey].push_back(*spval);
        }
    }
    ...
```

Example Program

```
void test()
{
    using   demo_map = test_map<test_string<char>, test_list<test_string<char>>>;
    auto    spmap = allocate<demo_map, test_heap>();
    auto    spkey = allocate<test_string<char>, test_heap>();
    auto    spval = allocate<test_string<char>, test_heap>();
    char    key[512], value[512];

    for (int i = 0;  i < 10;  ++i)
    {
        sprintf(key, "this is test key string %d", i);
        spkey->assign(key);

        for (int j = 1;  j <= 5;  ++j)
        {
            sprintf(value, "this is a very, very, very long test value string %d", i*100+j);
            spval->assign(value);
            (*spmap)[*spkey].push_back(*spval);
        }
    }
    ...
```

Example Program

```
void test()
{
    using   demo_map = test_map<test_string<char>, test_list<test_string<char>>>;
    auto    spmap = allocate<demo_map, test_heap>();
    auto    spkey = allocate<test_string<char>, test_heap>();
    auto    spval = allocate<test_string<char>, test_heap>();
    char    key[512], value[512];

    for (int i = 0;  i < 10;  ++i)
    {
        sprintf(key, "this is test key string %d", i);
        spkey->assign(key);

        for (int j = 1;  j <= 5;  ++j)
        {
            sprintf(value, "this is a very, very, very long test value string %d", i*100+j);
            spval->assign(value);
            (*spmap)[*spkey].push_back(*spval);
        }
    }
    ...
```

Example Program

```
void test()
{
    using   demo_map = test_map<test_string<char>, test_list<test_string<char>>>;
    auto    spmap = allocate<demo_map, test_heap>();
    auto    spkey = allocate<test_string<char>, test_heap>();
    auto    spval = allocate<test_string<char>, test_heap>();
    char    key[512], value[512];

    for (int i = 0;  i < 10;  ++i)
    {
        sprintf(key, "this is test key string %d", i);
        spkey->assign(key);

        for (int j = 1;  j <= 5;  ++j)
        {
            sprintf(value, "this is a very, very, very long test value string %d", i*100+j);
            spval->assign(value);
            (*spmap)[*spkey].push_back(*spval);
        }
    }
    ...
```

Example Program

```
    ...
    for (auto const& kvp : *spmap)                  //- Print the elements
    {
        cout << kvp.first << endl;
        for (auto const& lv : kvp.second)
        {
            cout << "    " <<  lv << endl;
        }
    }

    test_heap::swap_buffers();                     //- Swap in the shadow buffers

    for (auto const& kvp : *spmap)                 //- Print the elements, again
    {
        cout << kvp.first << endl;
        for (auto const& lv : kvp.second)
        {
            cout << "    " <<  lv << endl;
        }
    }
}
```

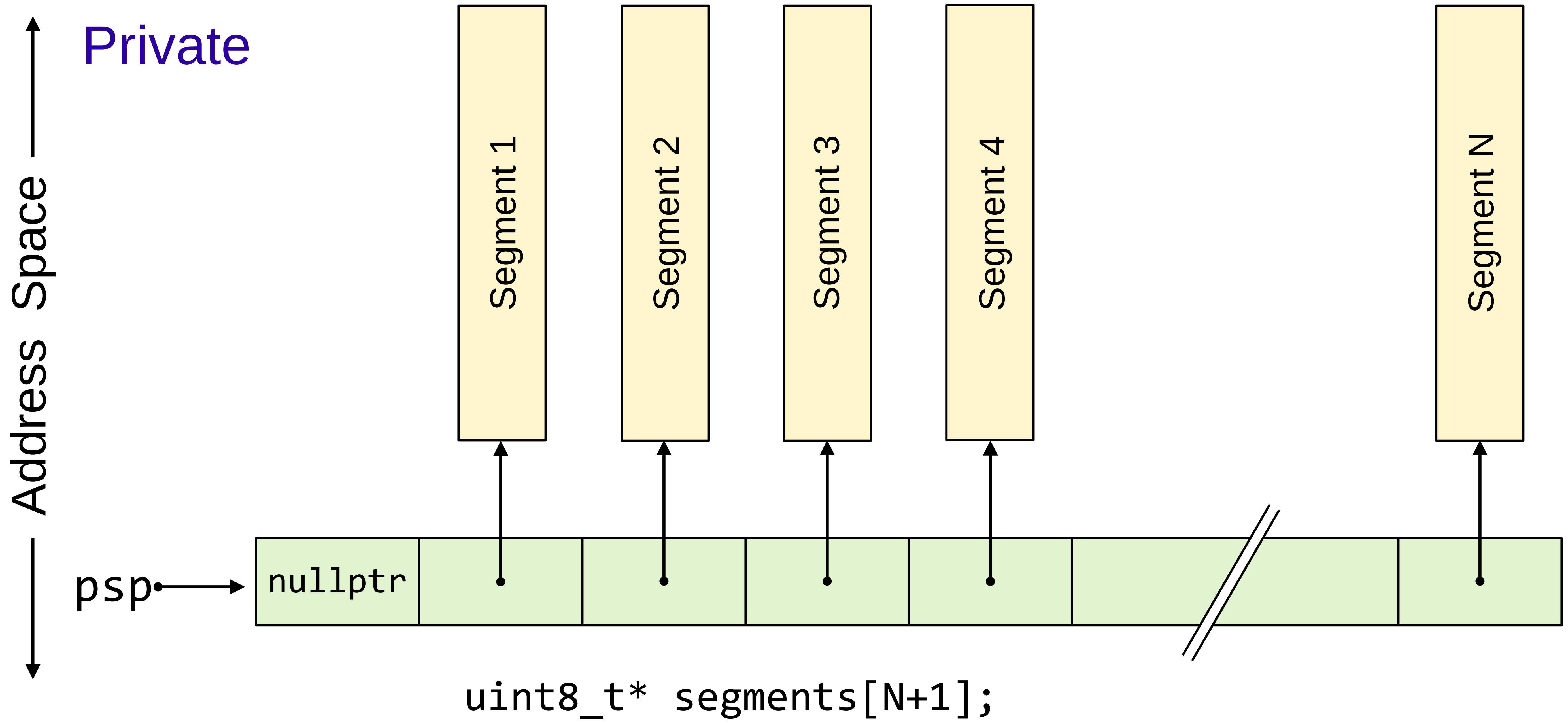
Example Program

```
    ...
    for (auto const& kvp : *spmap)              //- Print the elements
    {
        cout << kvp.first << endl;
        for (auto const& lv : kvp.second)
        {
            cout << "    " <<  lv << endl;
        }
    }

    test_heap::swap_buffers();                 //- Swap in the shadow buffers

    for (auto const& kvp : *spmap)             //- Print the elements, again
    {
        cout << kvp.first << endl;
        for (auto const& lv : kvp.second)
        {
            cout << "    " <<  lv << endl;
        }
    }
}
```

Example Program

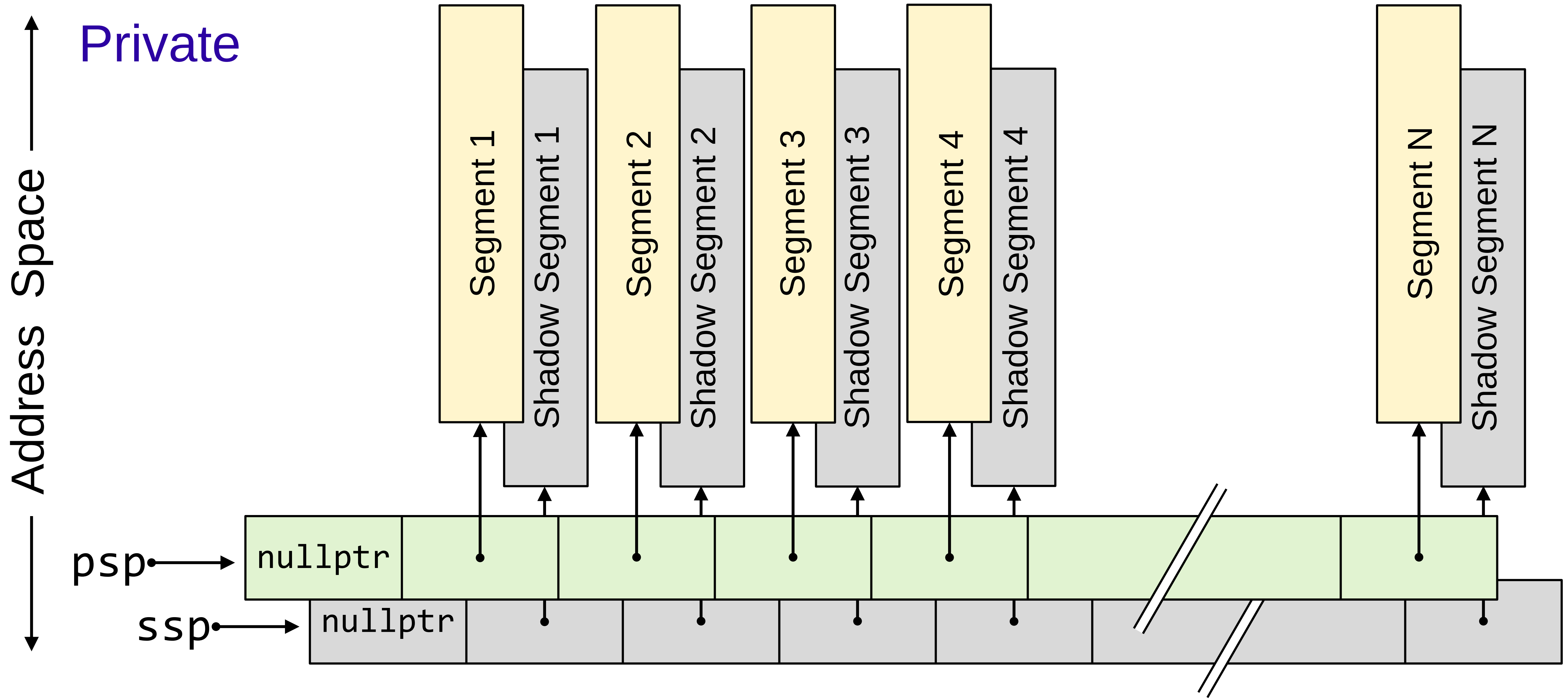
```
    ...
    for (auto const& kvp : *spmap)                //- Print the elements
    {
        cout << kvp.first << endl;
        for (auto const& lv : kvp.second)
        {
            cout << "    " <<  lv << endl;
        }
    }

    test_heap::swap_buffers();                   //- Swap in the shadow buffers

    for (auto const& kvp : *spmap)               //- Print the elements, again
    {
        cout << kvp.first << endl;
        for (auto const& lv : kvp.second)
        {
            cout << "    " <<  lv << endl;
        }
    }
}
```

Demo



Private
Address Space
Segment 1
Segment 2
Segment 3
Segment 4
Segment N
psp
nullptr
uint8_t* segments[N+1];

Demo



Private
Address Space
Segment 1
Shadow Segment 1
Segment 2
Shadow Segment 2
Segment 3
Shadow Segment 3
Segment 4
Shadow Segment 4
Segment N
Shadow Segment N
psp
nullptr
ssp
nullptr

Comments

- Possible applications
  - Relocatable heap for private use
  - Relocatable heap for shared memory
  - Instrumented debug allocator
- This is one possible implementation of these concepts – many are possible
- This is a work in progress – stay tuned…

This is the end, this is the end, my friend…