Fancy Pointers for Fun and Profit

Bob Steagall
CppCon 2018

Fancy Pointers for Fun and Profit

(Stupid Allocator Tricks)

Bob Steagall
CppCon 2018

Overview

- Motivating problem
- Addressing and allocation
- Framework and synthetic pointer implementation
- Detour – synthetic pointer performance
- Relocatable heap examples
- Summary

Motivating Problem – Persistence

Problem Context and Statement

- I have a set of types/objects
  - Have container data members, possibly nested
  - Have a large number of objects
  - 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 source objects and **serialize** them 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** object state
  - Post-condition: each destination object in the destination process is semantically identical to its corresponding source object
- Traversal-based serialization (TBS)

The Intermediate Format

- The intermediate format can provide several kinds 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 in destination process

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

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
- I want to 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 in a (possibly) different process,

  - 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 the destination process and its address in the destination process
- These types are relocatable
  - Integer types
  - Floating point types
  - A standard layout (POD) type that contains only integer and floating-point types and zero or more standard layout types containing only integers and

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
- Source side
  - Ensure that relocatable type requirements are observed by all contents
  - Construct the master object in the heap at a known address
  - Allocate stuff to be persisted from the heap and accessible via the master object
  - Serialize the heap
- Destination side
  - De-serialize the heap
  - Obtain access to the heap's contents through the master object

Addressing and Allocation

Thinking About Addressing and Memory Allocation

- Structural Management
  - Addressing Model
  - Storage Model
  - Pointer Interface
  - Allocation Strategy
- Concurrency Management
  - Thread Safety
  - Transaction Safety

Aspect – Addressing Model

- Policy type that implements primitive addressing operations
  - Analogous to `void*`
  - Convertible to `void*`
- The addressing model defines
  - The bits used to represent an address
  - How an address is computed from those bits
  - How memory from the storage model is arranged
- Representations
  - Ordinary pointer `void*` (aka natural pointer)
  - Fancy `void` pointer (aka synthetic pointer, pointer-like type)
- Usually closely coupled with storage model

Aspect – 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 of allocation
  - Usually closely coupled with the addressing model

- **Segment**: a large 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
  - `CreateFileMapping()` / `MapViewOfFile()` Windows shared memory

Aspect – Pointer Interface

- Policy type that wraps the addressing model to emulate a pointer to data
  - Analogous to `T*`
  - Provides enough pointer syntax for containers to function
  - Is convertible "in the right direction" to ordinary pointers
  - Is convertible "in the right direction" to other pointer interface types
  - Extends the interface of RandomAccessIterator
- Representations
  - Ordinary pointer `T*` (aka, natural pointer)
  - Synthetic pointer (aka fancy pointer, pointer-like type)

Aspect – 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
    - **Chunk:** A region of memory carved out of a segment to be used by an allocator's client
  - Provides chunks to the client in terms of the pointer interface

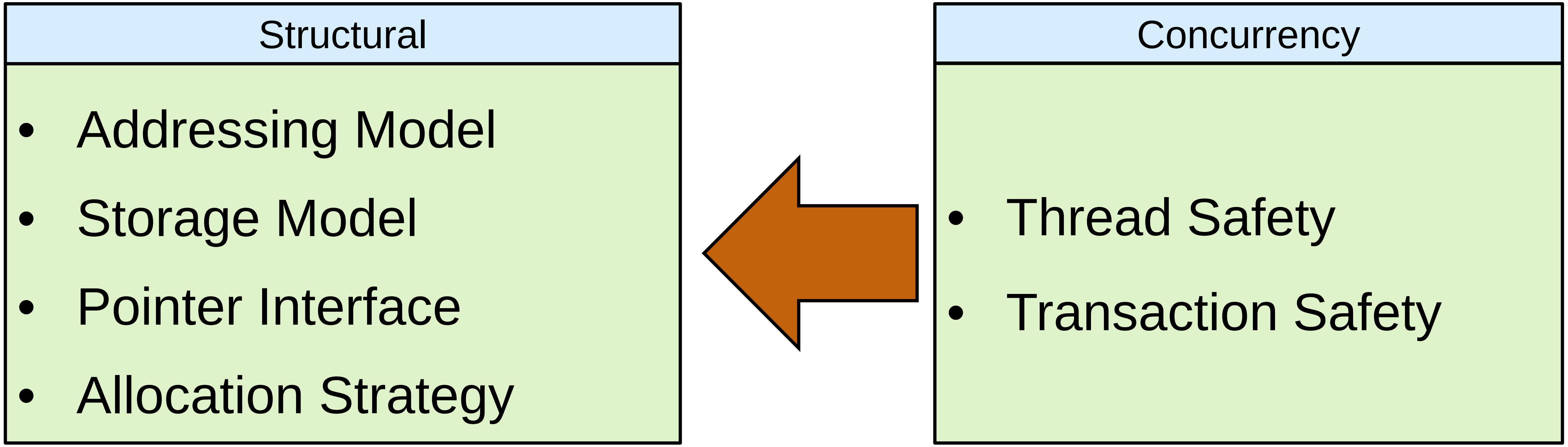
- Analogous to:
  - `malloc()` / `free()`
  - `::operator new()` / `::operator delete()`
  - `tcmalloc` / `jemalloc` / `dlmalloc` / `Hoard`

Aspects – Thread Safety and Transaction Safety

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

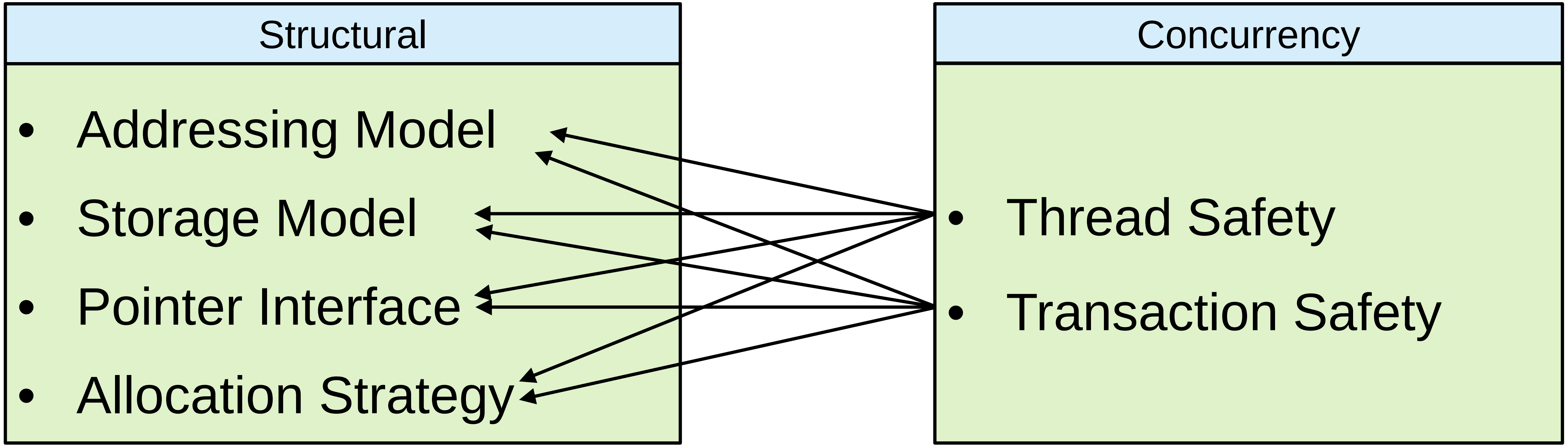
Aspects – Thread Safety and Transaction Safety

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

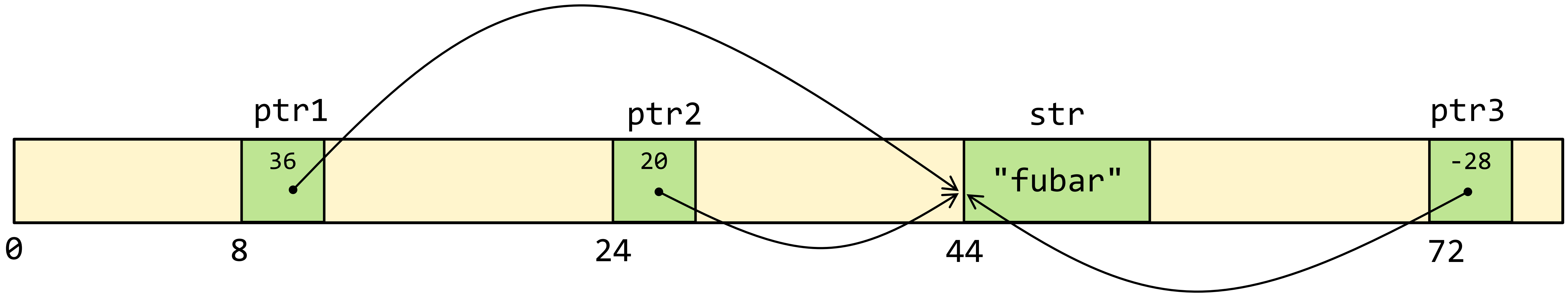


Aspects – Thread Safety and Transaction Safety

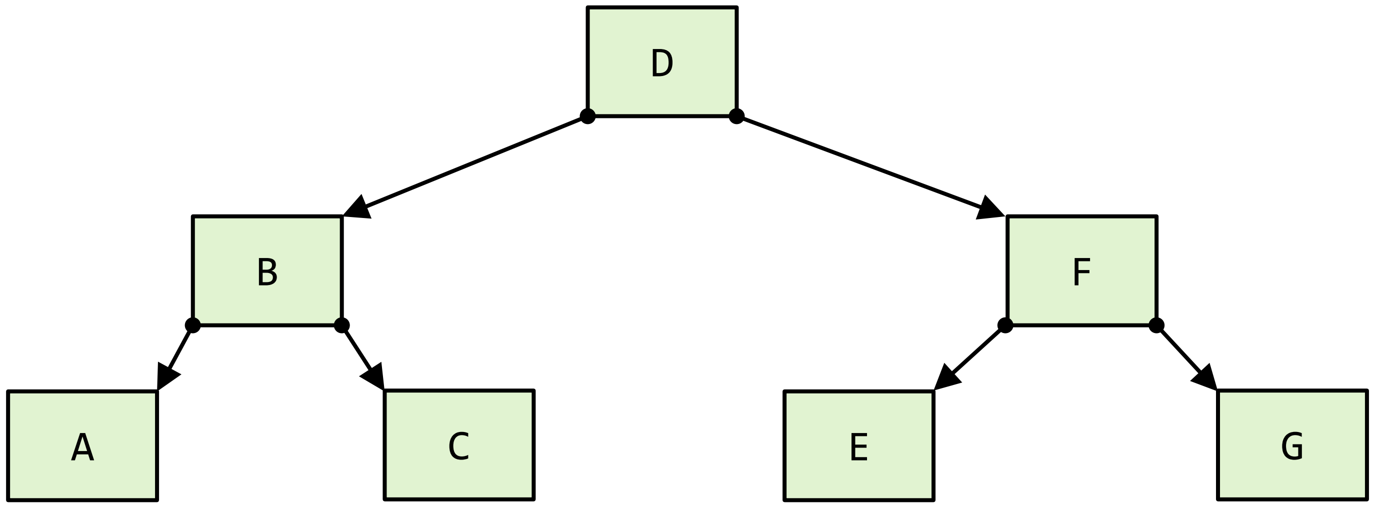
- Thread safety – correct operation with multiple threads/processes
- Transaction safety – supporting allocate/commit/rollback semantics



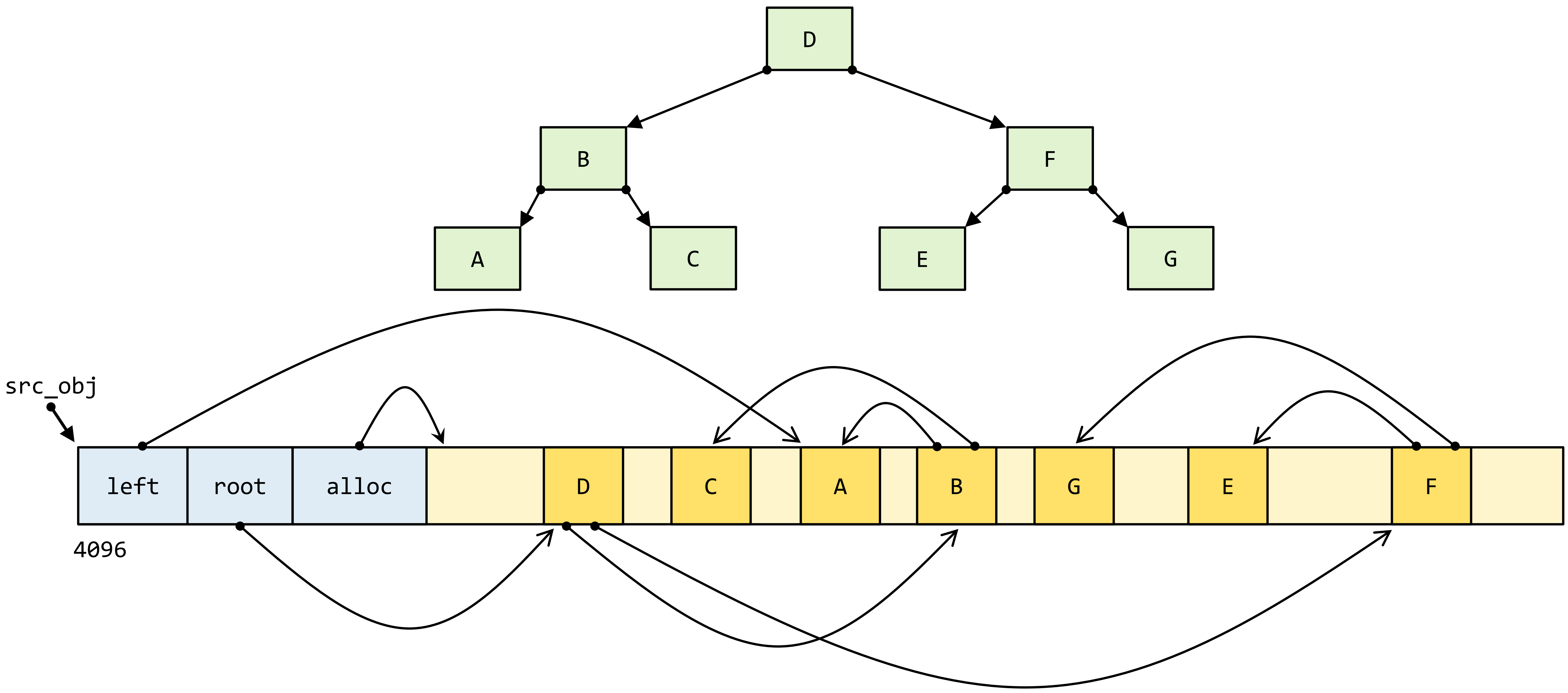
Example Offset Addressing Model



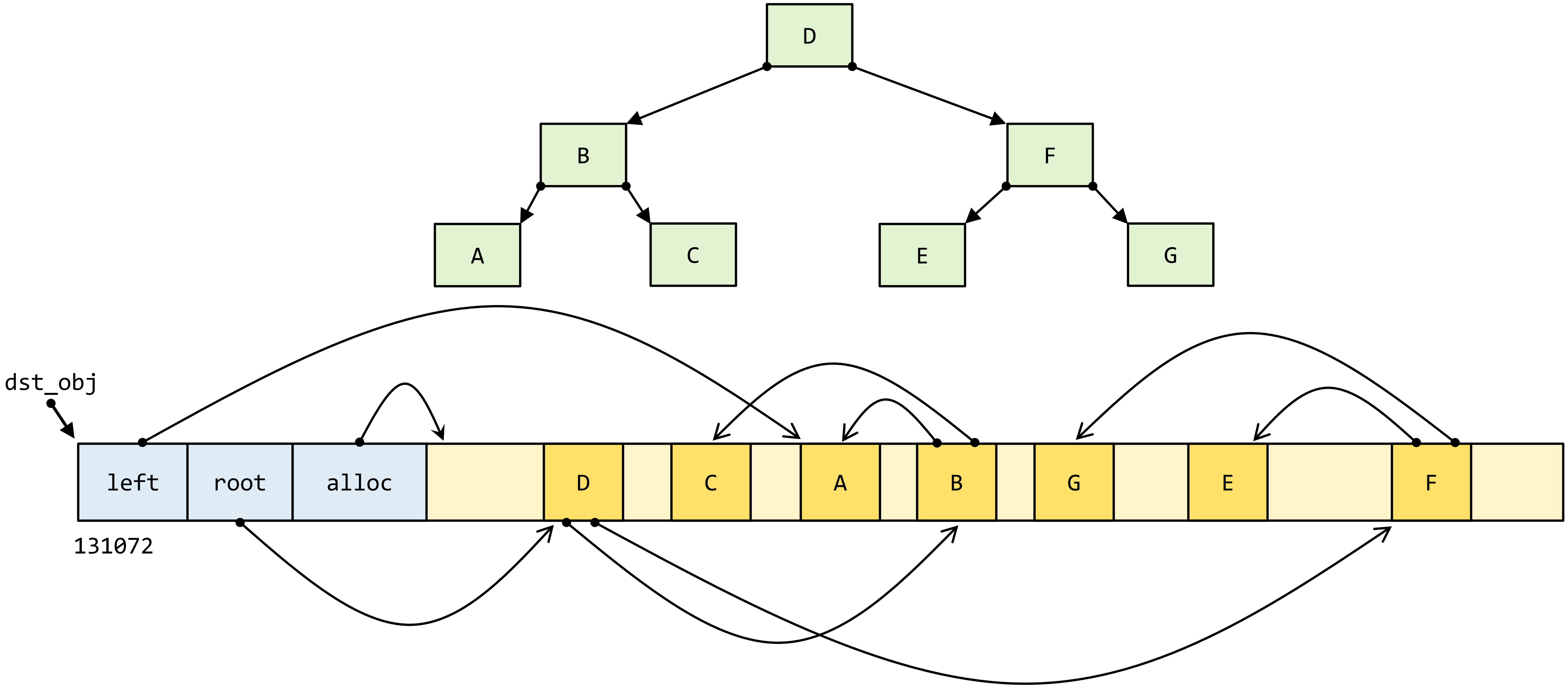
Example Application – Self-Contained DOM



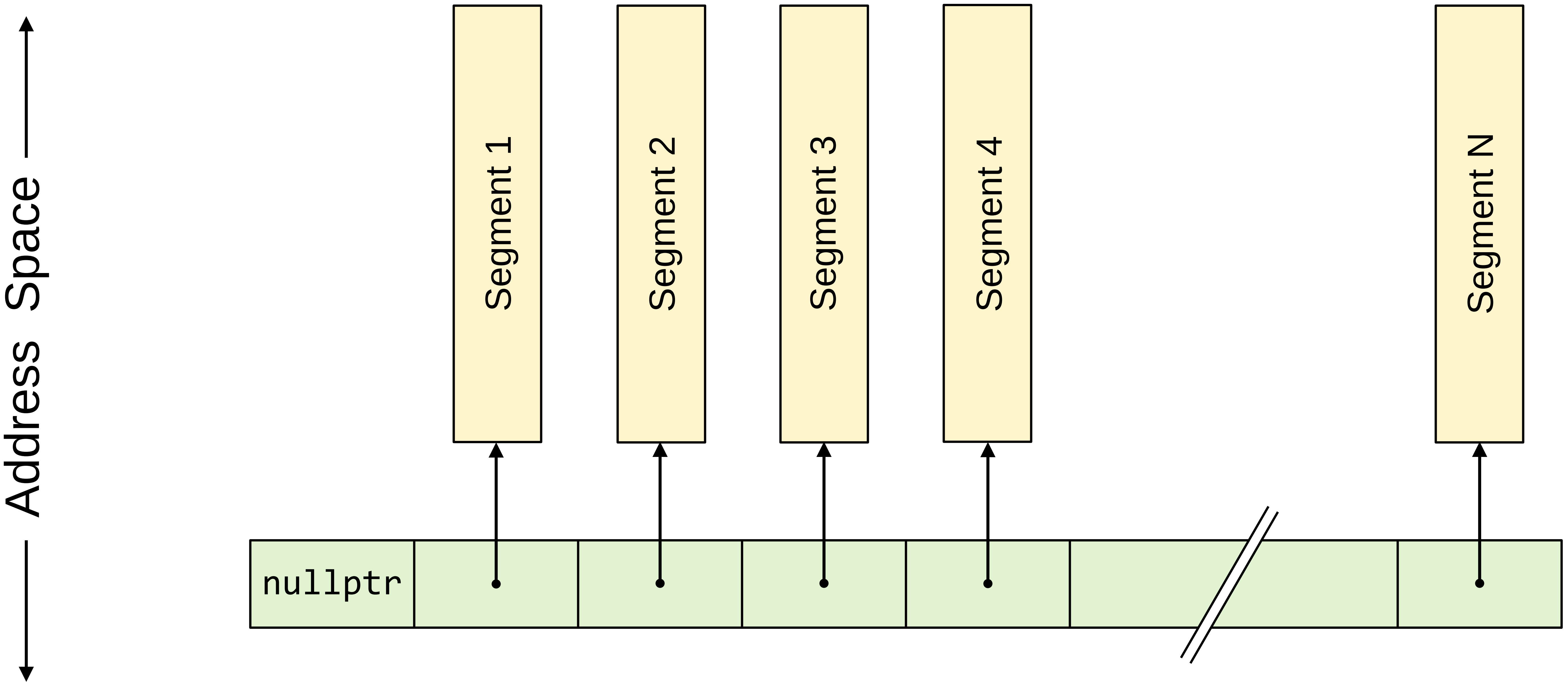
Example Application – Self-Contained DOM



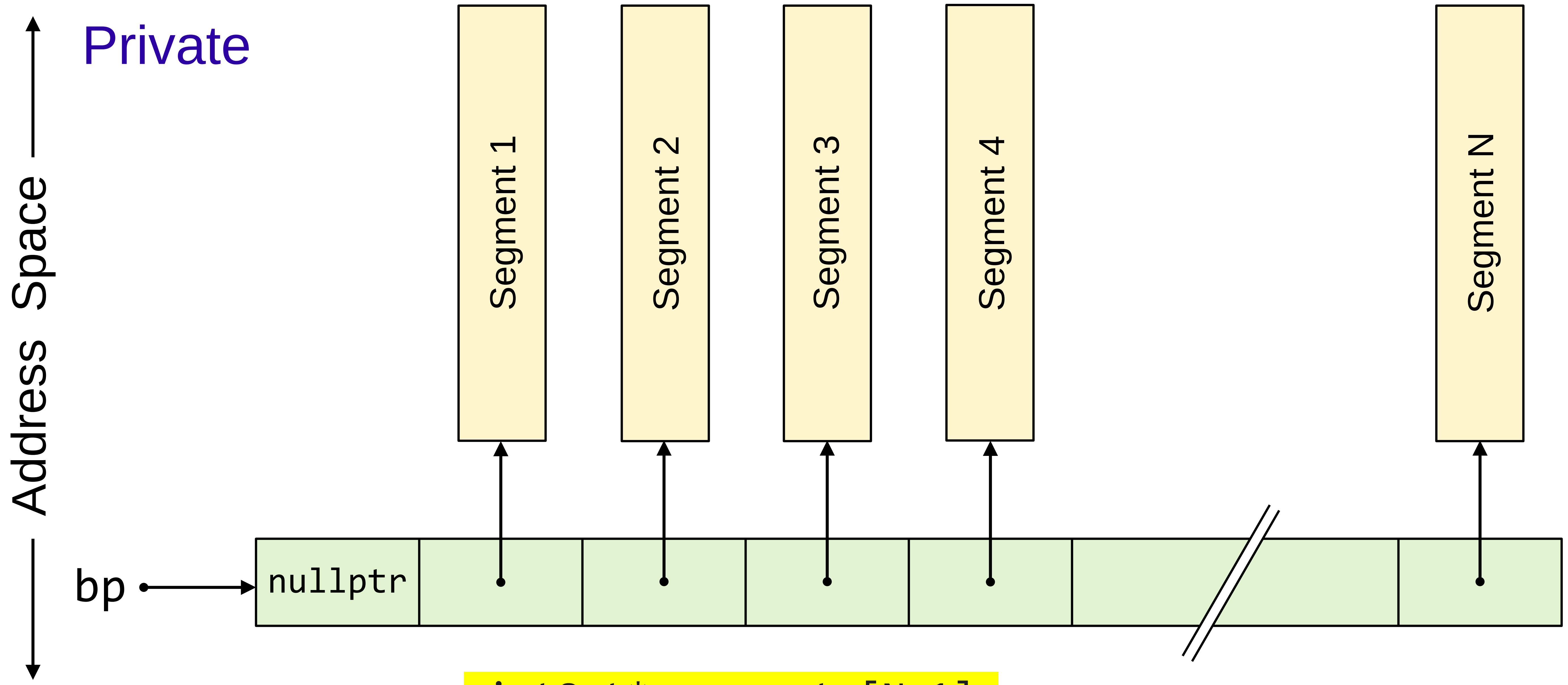
Example Application – Self-Contained DOM



Example 2D Addressing Model

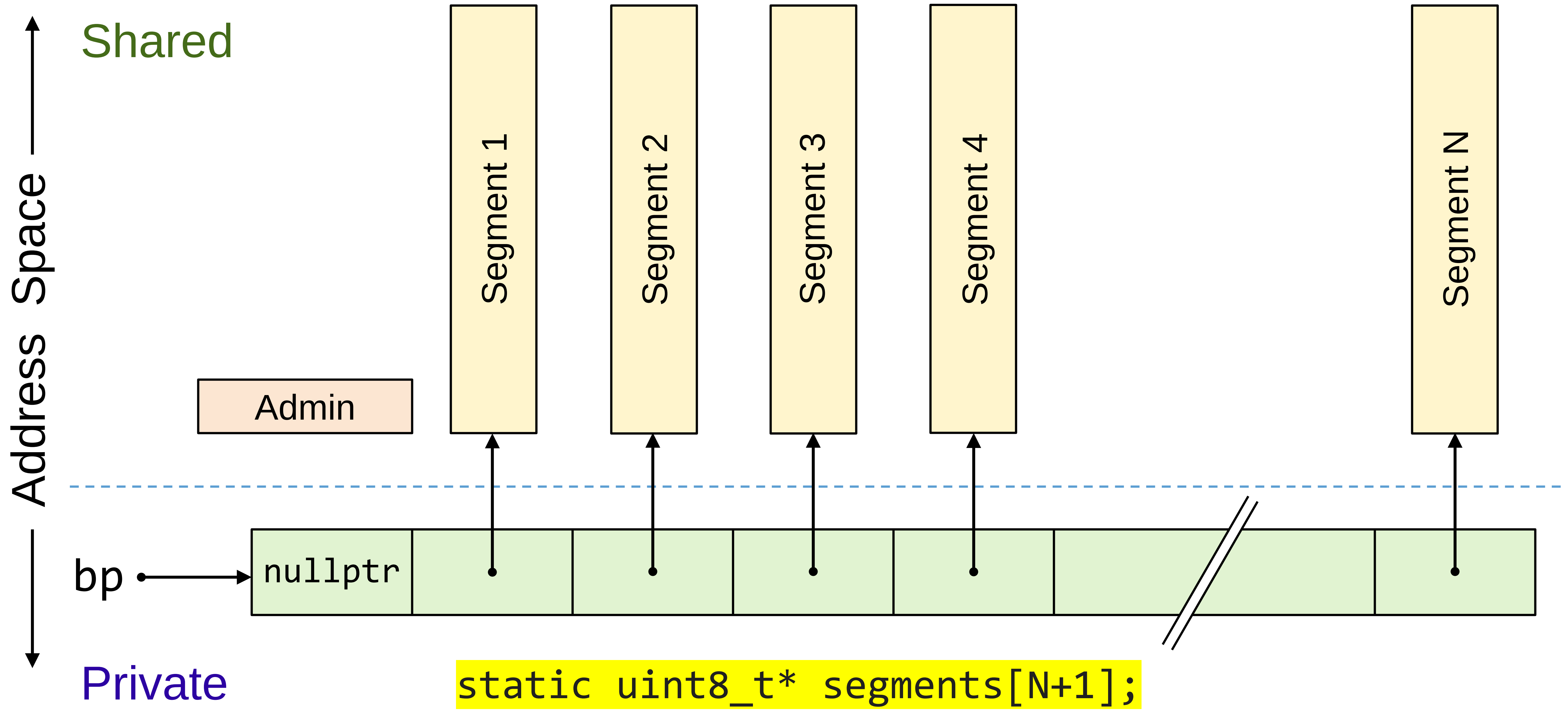


Example 2D Addressing and Storage Models – Private Segments



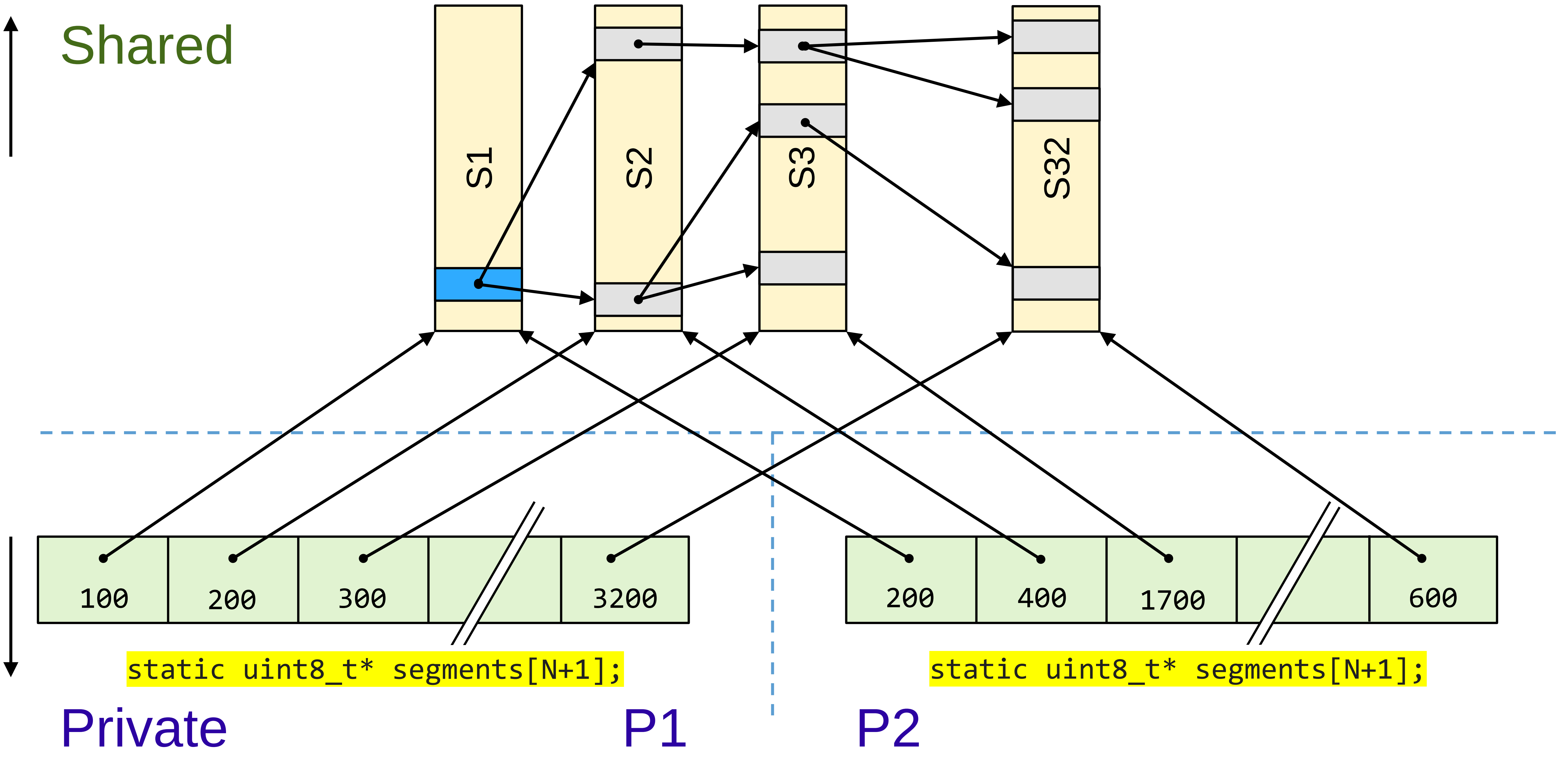

```
uint8_t* segments[N+1];
```

Example 2D Addressing and Storage Models – Shared Segments




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

Example Application – Shared Memory Database



On Synthetic Pointers

Pointer-Like Types (AKA Synthetic / Fancy Pointers)

- Mentioned four times in the standard, but the only substance is in the requirements for *NullablePointer* (see Table 28 in N4687):
  - Must satisfy several requirements:
    - *EqualityComparable*, *DefaultConstructible*, *CopyConstructible*, *CopyAssignable*, and *Destructible*
  - Have swappable lvalues
  - Default initialization may produce an indeterminate result; using may lead to UB
  - Value initialization produces a null result
  - Construction with / assignment from `nullptr` produces a null result
  - May be contextually convertible to `bool`
  - Certain fundamental operations (see Table 28) may not throw exceptions

Fancy Pointer Limitations

- There is a runtime cost
  - The fancy pointer arithmetic we implement will not be as fast as that of ordinary pointers

- Casting is limited to `static_cast<>()`
  - No `const_cast<>()`
  - No `dynamic_cast<>()`
  - No `reinterpret_cast<>()`
  - No C-style cast `(T*)`

Some Framework Classes

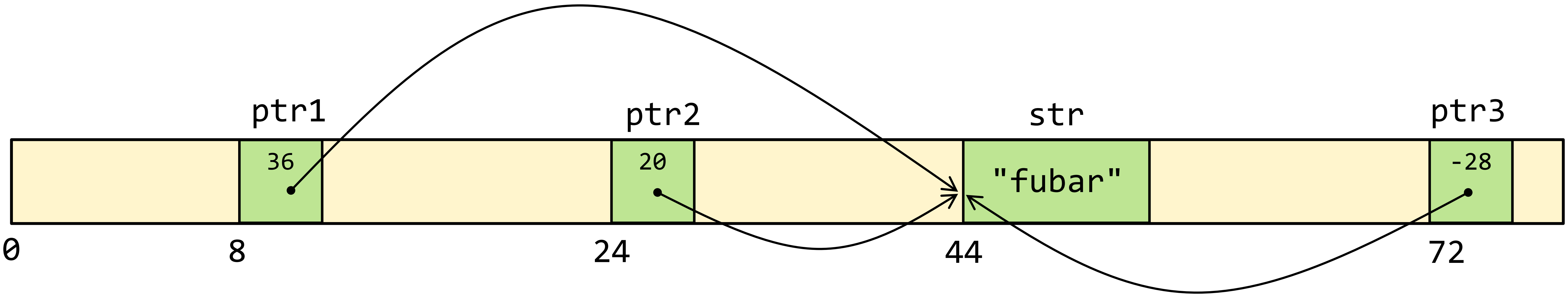
Pointer Support

- Addressing models
  - `offset_addressing_model<SM>`
  - `based_2d_xl_addressing_model<SM> / storage_model_base`
- Pointer interface `syn_ptr<T, AM>`

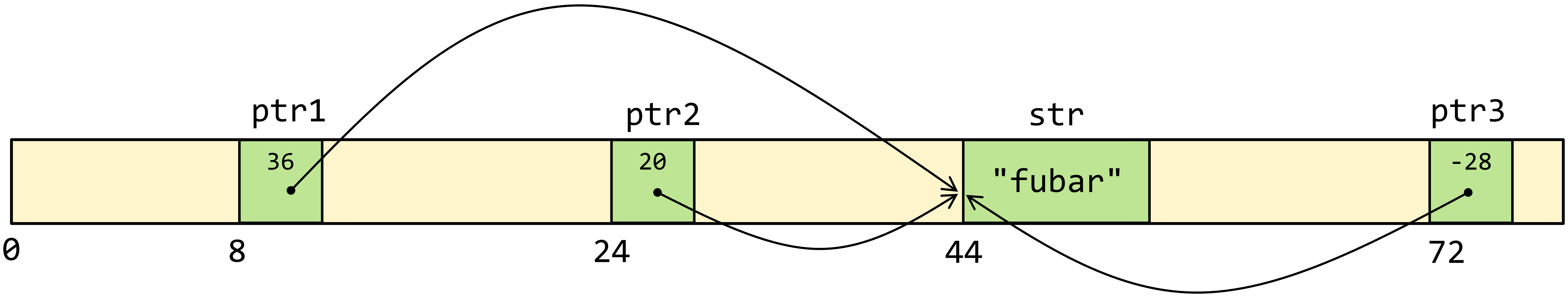
Pointer Support

- Addressing models
  - `offset_addressing_model<SM>`
  - `based_2d_xl_addressing_model<SM> / storage_model_base`
- Pointer interface `syn_ptr<T, AM>`

Example Offset Addressing View

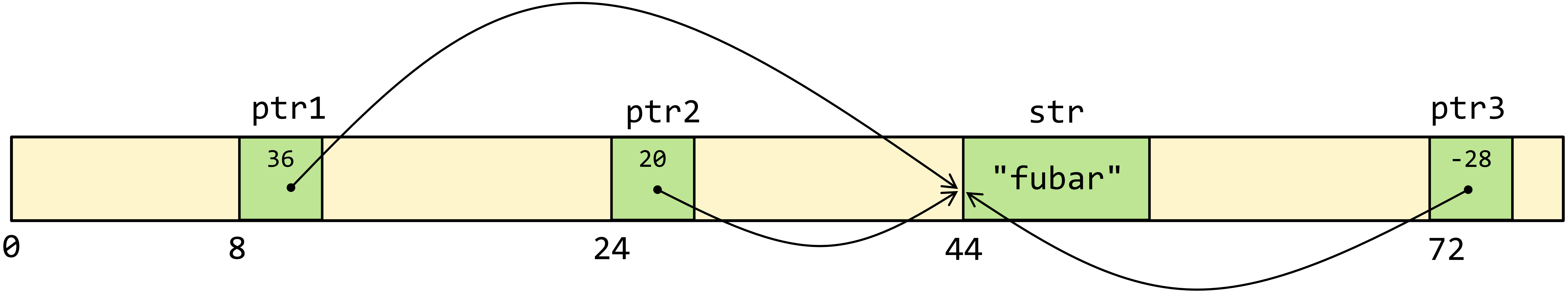


Example Offset Addressing View



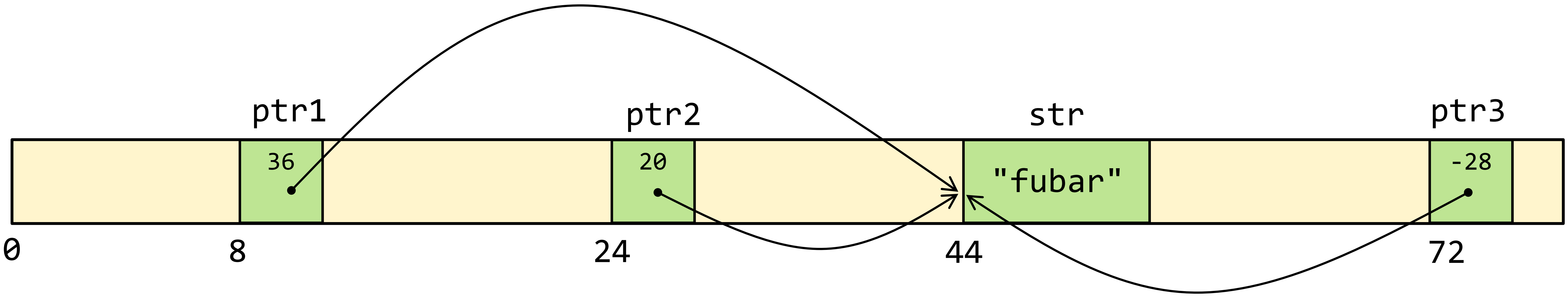
address = (char*)this + offset

Example Offset Addressing View




```
address = (char*)this + offset

addressof(*ptr1) == addressof(*ptr2)
```

Example Offset Addressing View




```
address = (char*)this + offset

addressof(*ptr1) == addressof(*ptr2)

memcmp(&ptr1, &ptr2, sizeof(ptr1)) != 0
```

Offset Addressing Model

```
class offset_addressing_model
{
  public:
    using size_type       = std::size_t;
    using difference_type = std::ptrdiff_t;

  public:
    ~offset_addressing_model() = default;

    offset_addressing_model() noexcept;
    offset_addressing_model(offset_addressing_model&& other) noexcept;
    offset_addressing_model(offset_addressing_model const& other) noexcept;
    offset_addressing_model(std::nullptr_t) noexcept;
    offset_addressing_model(void const* p) noexcept;

    offset_addressing_model&  operator =(offset_addressing_model&& rhs) noexcept;
    offset_addressing_model&  operator =(offset_addressing_model const& rhs) noexcept;
    offset_addressing_model&  operator =(std::nullptr_t) noexcept;
    offset_addressing_model&  operator =(void const* p) noexcept;
    ...
};
```

Offset Addressing Model

```
class offset_addressing_model
{
  public:
    ...

    void*   address() const noexcept;

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

    ...
};
```

Offset Addressing Model

```
class offset_addressing_model
{
    ...

  private:
    using diff_type = difference_type ;

    enum : diff_type { null_offset = 1 };

  private:
    diff_type   m_offset;

  private:
    static  diff_type   offset_between(void const *from, void const *to) noexcept;

    diff_type   offset_to(offset_addressing_model const &other) noexcept;
    diff_type   offset_to(void const *other) noexcept;
};
```

Offset Addressing Model

```
inline offset_addressing_model::difference_type
offset_addressing_model::offset_between(void const *from, void const *to) noexcept
{
    return reinterpret_cast<intptr_t>(to) - reinterpret_cast<intptr_t>(from);
}


inline offset_addressing_model::difference_type
offset_addressing_model::offset_to(offset_addressing_model const &other) noexcept
{
    return (other.m_offset == null_offset) ? null_offset
                                           : (offset_between(this, &other) + other.m_offset);
}


inline offset_addressing_model::difference_type
offset_addressing_model::offset_to(void const *other) noexcept
{
    return (other == nullptr) ? null_offset : offset_between(this, other);
}
```

Offset Addressing Model

```
inline void*
offset_addressing_model::address() const noexcept
{
    return (m_offset == null_offset)
            ? nullptr
            : reinterpret_cast<void*>(reinterpret_cast<uintptr_t>(this) + m_offset);
}


inline void
offset_addressing_model::assign_from(void const* p)
{
    m_offset = offset_to(p);
}


inline void
offset_addressing_model::increment(difference_type inc) noexcept
{
    m_offset += inc;
}
```

Offset Addressing Model

```
inline
offset_addressing_model::offset_addressing_model() noexcept
:   m_offset{null_offset}
{}


inline
offset_addressing_model::offset_addressing_model(offset_addressing_model&& rhs) noexcept
:   m_offset{offset_to(rhs)}
{}


inline
offset_addressing_model::offset_addressing_model(offset_addressing_model const& rhs) noexcept
:   m_offset{offset_to(rhs)}
{}
```

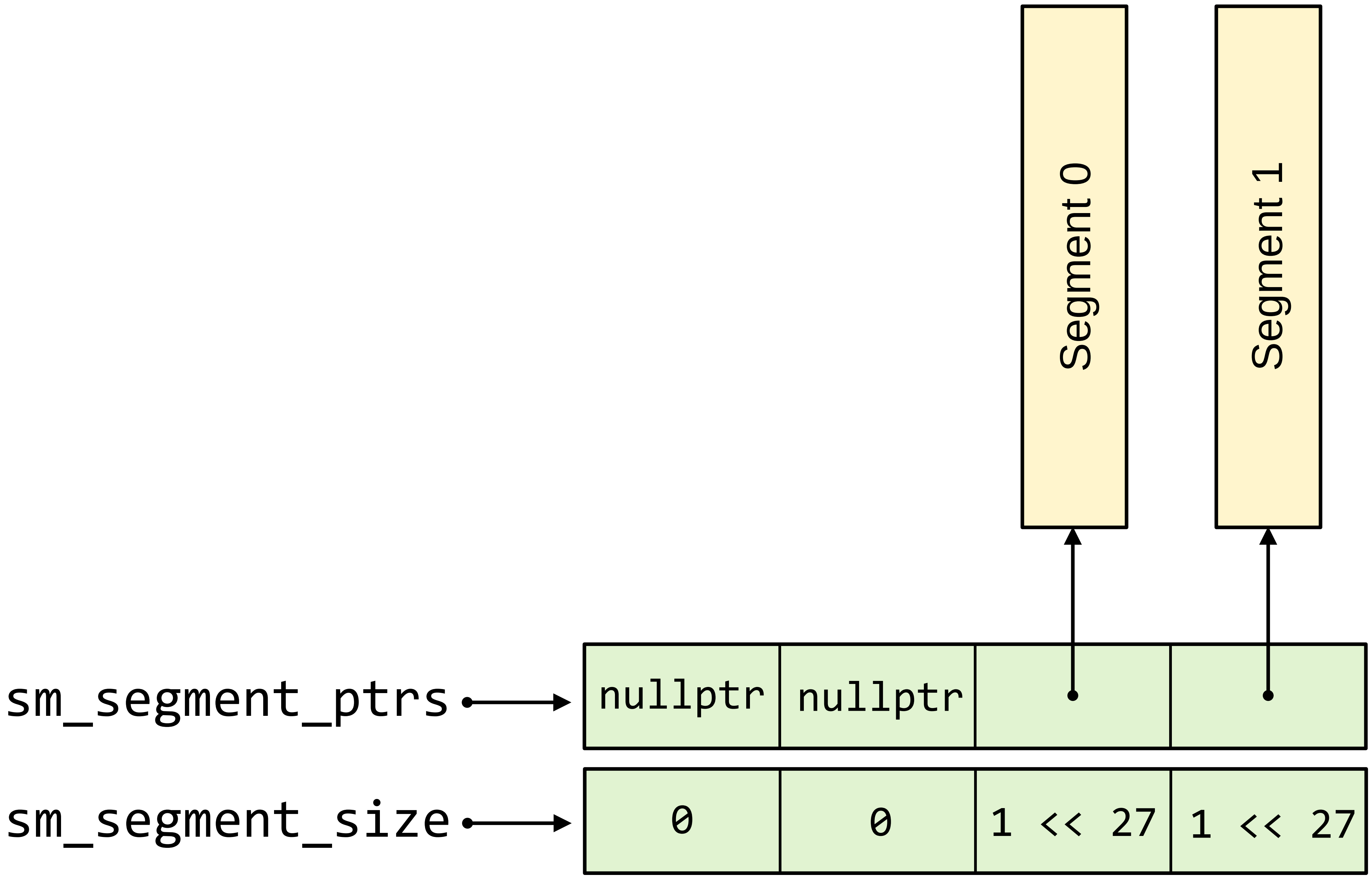
Offset Addressing Model

```
inline offset_addressing_model&
offset_addressing_model::operator =(offset_addressing_model&& rhs) noexcept
{
    m_offset = offset_to(rhs);
    return *this;
}


inline offset_addressing_model&
offset_addressing_model::operator =(offset_addressing_model const& rhs) noexcept
{
    m_offset = offset_to(rhs);
    return *this;
}
```

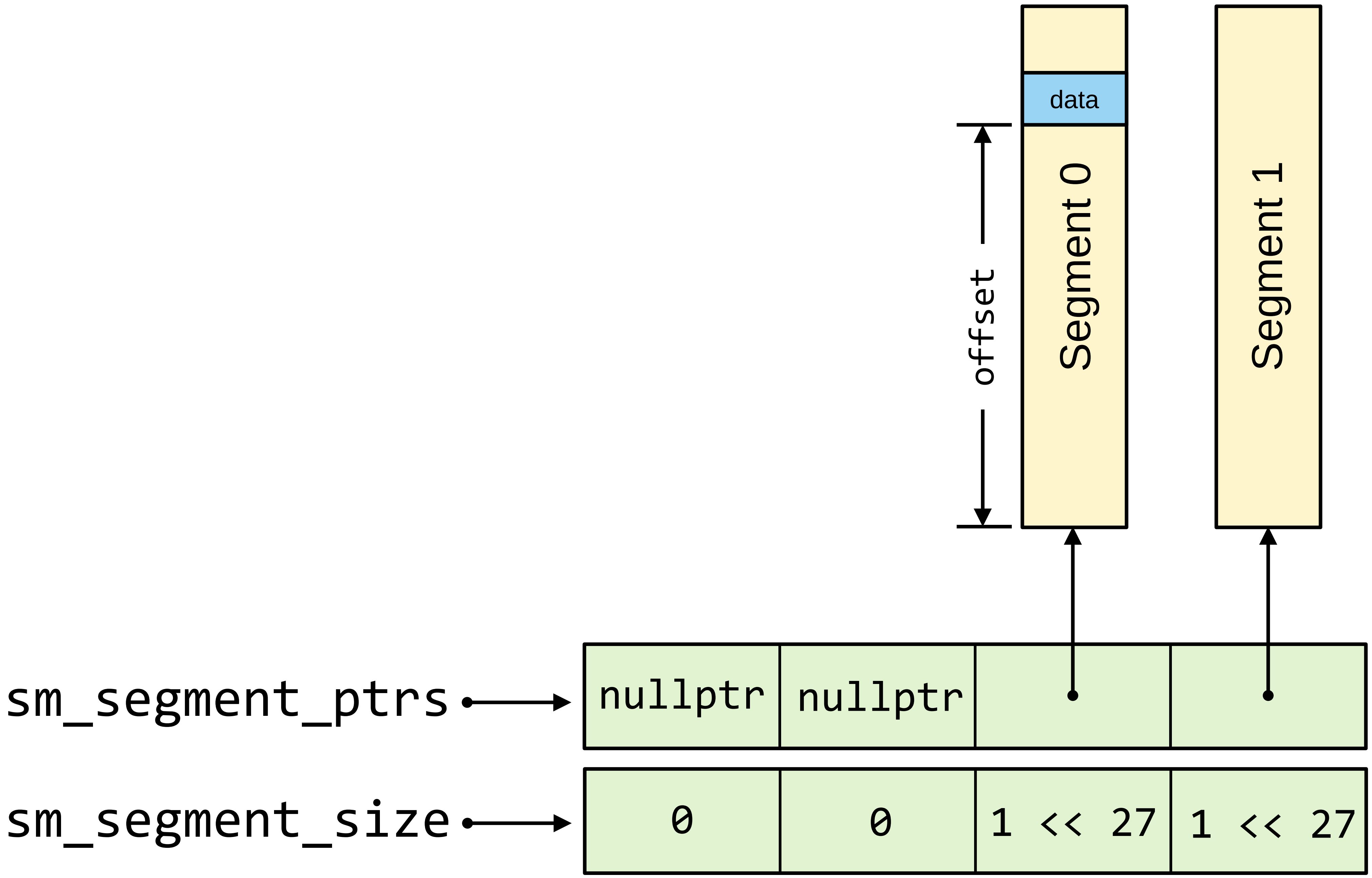
Support Types

- Addressing models
  - `offset_addressing_model<SM>`
  - `based_2d_xl_addressing_model<SM> / storage_model_base`
- Pointer interface `syn_ptr<T, AM>`

2D Storage Model – Memory Structure

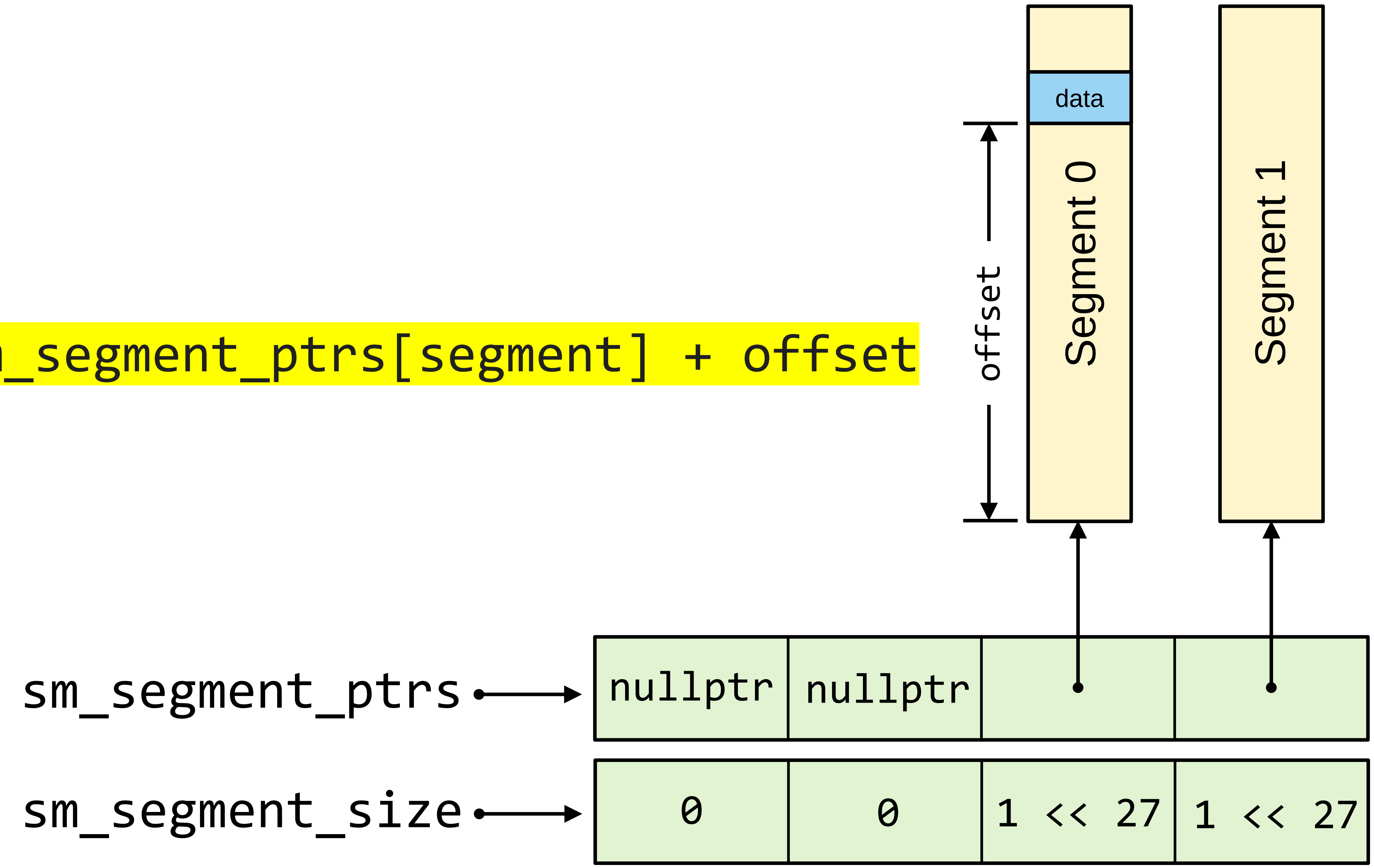


2D Storage Model – Addressing View



2D Storage Model – Addressing View

`address = sm_segment_ptrs[segment] + offset`



2D Storage Model

```
class storage_model_base
{
    ...

    static  char*       segment_address(size_type segment) noexcept;

  protected:
    static  char*       sm_segment_ptrs[max_segments + 2];
    static  char*       sm_shadow_ptrs[max_segments + 2];
    static  size_type   sm_segment_size[max_segments + 2];
    static  bool        sm_ready;
};

//------
//
inline char*
storage_model_base::segment_address(size_type segment) noexcept
{
    return sm_segment_ptrs[segment];
}
```

Based 2D XL Addressing Model

```
template<typename SM>
class based_2d_xl_addressing_model
{
  public:
    using size_type       = std::size_t;
    using difference_type = std::ptrdiff_t;

  public:
    ~based_2d_xl_addressing_model() = default;

    based_2d_xl_addressing_model() noexcept = default;
    based_2d_xl_addressing_model(based_2d_xl_addressing_model&&) noexcept = default;
    based_2d_xl_addressing_model(based_2d_xl_addressing_model const&) noexcept = default;
    based_2d_xl_addressing_model(std::nullptr_t) noexcept;
    based_2d_xl_addressing_model(size_type segment, size_type offset) noexcept;

    based_2d_xl_addressing_model&  operator =(based_2d_xl_addressing_model&&) noexcept = default;
    based_2d_xl_addressing_model&  operator =(based_2d_xl_addressing_model const&)noexcept=def…;
    based_2d_xl_addressing_model&  operator =(std::nullptr_t) noexcept;
    ...
};
```

Based 2D XL Addressing Model

```
template<typename SM>
class based_2d_xl_addressing_model
{
  public:
    ...

    void*   address() const noexcept;

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

  private:
    uint64_t    m_offset;
    uint64_t    m_segment;
};
```

Based 2D XL Addressing Model

```
template<typename SM> inline void*
based_2d_xl_addressing_model<SM>::address() const noexcept
{
    return SM::segment_address(m_segment) + m_offset;
}

template<typename SM> inline void
based_2d_xl_addressing_model<SM>::decrement(difference_type inc) noexcept
{
    m_offset -= inc;
}

template<typename SM> inline void
based_2d_xl_addressing_model<SM>::increment(difference_type inc) noexcept
{
    m_offset += inc;
}
```

Based 2D XL Addressing Model

```
template<typename SM> void
based_2d_xl_addressing_model<SM>::assign_from(void const* p)
{
    char const*     pdata = static_cast<char const*>(p);

    for (size_type idx = SM::first_segment_index();  idx <= SM::last_segment_index();  ++idx)
    {
        char const*     pbottom = SM::segment_address(segment_index);

        if (pbottom != nullptr)
        {
            char const*     ptop = pbottom + SM::segment_size(segment_index);

            if (pbottom <= pdata  &&  pdata < ptop)
            {
                m_offset  = pdata - pbottom;
                m_segment = idx;
                return;
            }
        }
    }
    m_segment = 0;
    m_offset  = pdata - static_cast<char const*>(nullptr);
}
```

Framework Types

- Addressing models
  - `wrapper_addressing_model<SM>`
  - `based_2d_xl_addressing_model<SM> / storage_model_base`
- Pointer interface `syn_ptr<T, AM>`
- Allocation strategy `monotonic_allocation_strategy<SM>`
- Allocator `rhx_allocator<T, AS>`

Synthetic Pointer Interface

```
template<class T, class AM>
class syn_ptr
{
  public:
    [ Special Member Functions ]

    [ Other Constructors ]

    [ Other Assignment Operators ]

    [ Conversion Operators ]

    [ Dereferencing and Pointer Arithmetic ]

    [ Helpers to Support Library Requirements ]

    [ Helpers to Support Comparison Operators ]

  private
    [ Member Data ]
};
```

Synthetic Pointer Interface – Helper Traits for SFINAE

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

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

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

template<class T, class U>
using enable_if_non_void_t =
    typename std::enable_if<!std::is_void<U>::value && std::is_same<T, U>::value, bool>::type;

template<class T>
using get_type_or_void_t =
    typename std::conditional<std::is_void<T>::value, void,
                              typename std::add_lvalue_reference<T>::type>::type;
```

Synthetic Pointer Interface – Helper Traits for SFINAE

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

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

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

template<class T, class U>
using enable_if_non_void_t =
    typename std::enable_if<!std::is_void<U>::value && std::is_same<T, U>::value, bool>::type;

template<class T>
using get_type_or_void_t =
    typename std::conditional<std::is_void<T>::value, void,
                              typename std::add_lvalue_reference<T>::type>::type;
```

Synthetic Pointer Interface – Helper Traits for SFINAE

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

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

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

template<class T, class U>
using enable_if_non_void_t =
    typename std::enable_if<!std::is_void<U>::value && std::is_same<T, U>::value, bool>::type;

template<class T>
using get_type_or_void_t =
    typename std::conditional<std::is_void<T>::value, void,
                              typename std::add_lvalue_reference<T>::type>::type;
```

Synthetic Pointer Interface – Helper Traits for SFINAE

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

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

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

template<class T, class U>
using enable_if_non_void_t =
    typename std::enable_if<!std::is_void<U>::value && std::is_same<T, U>::value, bool>::type;

template<class T>
using get_type_or_void_t =
    typename std::conditional<std::is_void<T>::value, void,
                              typename std::add_lvalue_reference<T>::type>::type;
```

Synthetic Pointer Interface – Nested Type Aliases

```
template<class T, class AM>
class syn_ptr
{
  public:
    //- Re-binder alias required for std::pointer_traits<T> / C++11 support.
    //
    template<class U> using rebind = syn_ptr<U, AM>;

    //- Other aliases required by allocator_traits<T>, pointer_traits<T>, and the containers.
    //
    using difference_type   = typename AM::difference_type;
    using size_type         = typename AM::size_type;
    using element_type      = T;
    using value_type        = T;
    using reference         = get_type_or_void_t<T>;
    using pointer           = syn_ptr;

    using iterator_category = std::random_access_iterator_tag;

    ...
};
```

Synthetic Pointer Interface – Special Member Functions

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

    //- Special member functions.
    //
    ~syn_ptr() noexcept = default;

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

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

    ...
};
```

Synthetic Pointer Interface – Other Constructors

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

    //- User-defined construction.  Allow only implicit conversion at compile time.
    //
    syn_ptr(AM am);
    syn_ptr(std::nullptr_t);

    template<class U, enable_if_convertible_t<U, T> = true>
    syn_ptr(U* p);

    template<class U, enable_if_convertible_t<U, T> = true>
    syn_ptr(syn_ptr<U, AM> const& p);

    ...
};
```

Synthetic Pointer Interface – Other Assignment Operators

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

    //- User-defined assignment.
    //
    syn_ptr&   operator =(std::nullptr_t);

    template<class U, enable_if_convertible_t<U, T> = true>
    syn_ptr&   operator =(U* p);

    template<class U, enable_if_convertible_t<U, T> = true>
    syn_ptr&   operator =(syn_ptr<U, AM> const& p);

    ...
};
```

Synthetic Pointer Interface – Conversion Operators

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

    //- User-defined conversion.
    //
    explicit    operator bool() const;

    template<class U, enable_if_convertible_t<T, U> = true>
                operator U* () const;

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

    template<class U, enable_if_not_convertible_t<T, U> = true>
    explicit    operator syn_ptr<U, AM>() const;

    ...
};
```

Synthetic Pointer Interface – De-referencing

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

    //- De-referencing and indexing.
    //
    template<class U = T, enable_if_non_void_t<T, U> = true>
    U*  operator ->() const;

    template<class U = T, enable_if_non_void_t<T, U> = true>
    U&  operator  *() const;

    template<class U = T, enable_if_non_void_t<T, U> = true>
    U&  operator [](size_type n) const;

    ...
};
```

Test Pointer Interface – Pointer Arithmetic

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

    //- Pointer arithmetic operators.
    //
    template<class U = T, enable_if_non_void_t<T, U> = true>
    difference_type operator -(const syn_ptr& p) const;

    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr         operator -(difference_type n) const;

    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr         operator +(difference_type n) const;

    ...
};
```

Synthetic Pointer Interface – Pointer Arithmetic

```
template<class T, class AM>
class syn_ptr
{
    ...
    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr&        operator ++();
    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr const   operator ++(int);

    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr&        operator --();
    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr const   operator --(int);

    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr&        operator +=(difference_type n);
    template<class U = T, enable_if_non_void_t<T, U> = true>
    syn_ptr&        operator -=(difference_type n);
    ...
};
```

Synthetic Pointer Interface

```
template<class T, class AM>
class syn_ptr
{
    ...
    //- Helper function required by pointer_traits<T>.
    //
    template<class U = T, enable_if_non_void_t<T, U> = true>
    static  syn_ptr pointer_to(U& e);

    //- Helper functions used to implement the comparison operators.
    //
    bool    equals(std::nullptr_t) const;

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

    template<class U, enable_if_comparable_t<T, U> = true>
    bool    equals(syn_ptr<U, AM> const& p) const;

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

Synthetic Pointer Interface

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

 private:
    template<class OT, class OAM> friend class syn_ptr;   //- For parametrized conversion ctor

    AM  m_addrmodel;
};
```

Synthetic Pointer Interface - Casting

```
//  template<class U, enable_if_convertible_t<T, U> = true>
//              operator U* () const;

template<class T, class AM>
template<class U, enable_if_convertible_t<T, U>> inline
syn_ptr<T, AM>::operator U* () const
{
    return static_cast<U*>(m_addrmodel.address());
}


//  template<class U, enable_if_not_convertible_t<T, U> = true>
//  explicit    operator U* () const;

template<class T, class AM>
template<class U, enable_if_not_convertible_t<T, U>> inline
syn_ptr<T, AM>::operator U* () const
{
    return static_cast<U*>(m_addrmodel.address());
}
```

Synthetic Pointer Interface - Dereferencing

```
//  template<class U = T, enable_if_non_void_t<T, U> = true>
//  U*  operator ->() const;

template<class T, class AM>
template<class U, enable_if_non_void_t<T, U>> inline U*
syn_ptr<T, AM>::operator ->() const
{
    return static_cast<U*>(m_addrmodel.address());
}


//  template<class U = T, enable_if_non_void_t<T, U> = true>
//  U&  operator  *() const;

template<class T, class AM>
template<class U, enable_if_non_void_t<T, U>> inline U&
syn_ptr<T, AM>::operator *() const
{
    return *static_cast<U*>(m_addrmodel.address());
}
```

Detour – Synthetic Pointer Performance

Synthetic Pointer Performance Test Outline

- 2 addressing models
  - `based_2d_xl`, `offset`
- 2 data types
  - `uint64_t`, `test_struct`
- 13 array sizes
  - 100, 200, 500, 1000, 2000, 5000, …, 1000000
- 1 algorithm
  - `sort()`
- 3 compilers
  - GCC 7.2 with libstdc++
  - Clang 5.0.1 with libstdc++
  - VC++ 2017 (15.4.4)

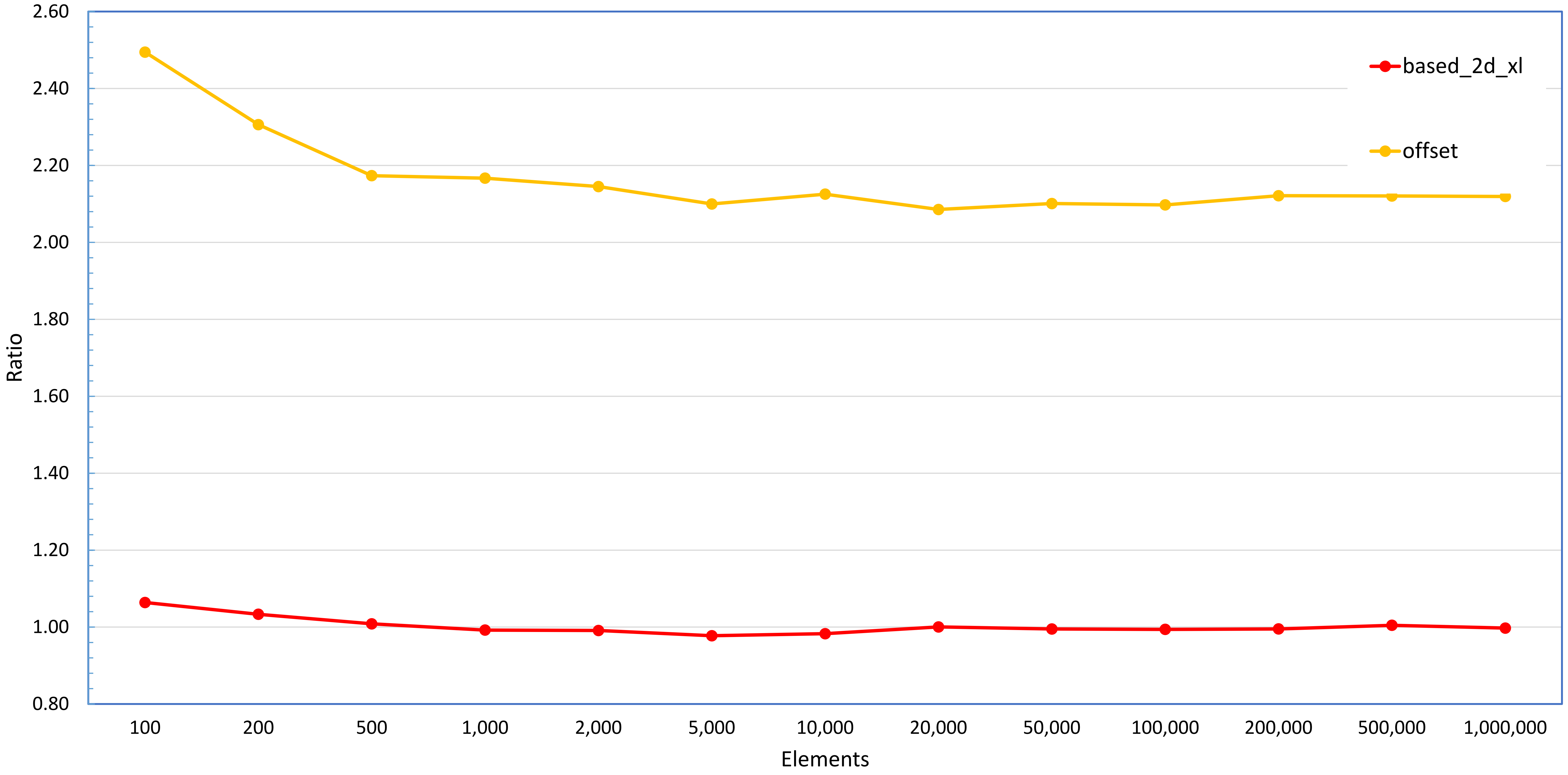
Synthetic Pointer Performance Test Environment

- Core-i7 16GB RAM Windows 10
- GCC 7.2 / Clang 5.0.1 on Ubuntu 18.04
  - Running on VMware 11.1 on Windows 10
- VS2017 on Windows 10
- All tests in a single thread and timed using `std::chrono`

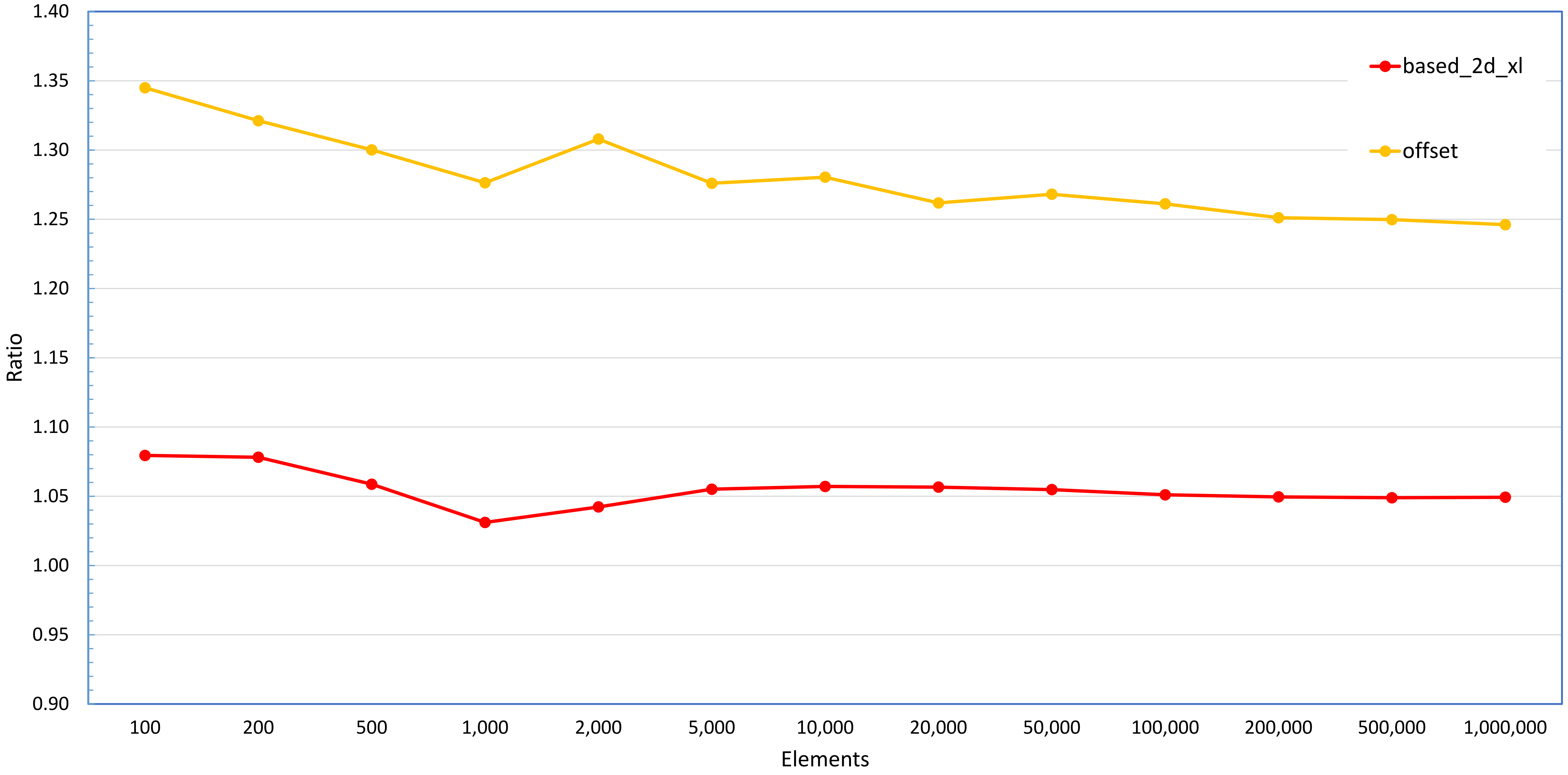
Synthetic Pointer Performance Test Environment

- Copy operations were repeated for a total of 10,000,000 copies
  - For example, copying 1,000,000 elements was done 10 times
- Sorts performed 10 to 100 times, depending on array size
  - Smaller arrays were sorted more times
- GCC and Clang both used `libstdc++` for std library facilities
- All results are **ratios** – time_for_syn_ptr / time_for_ordinary_ptr

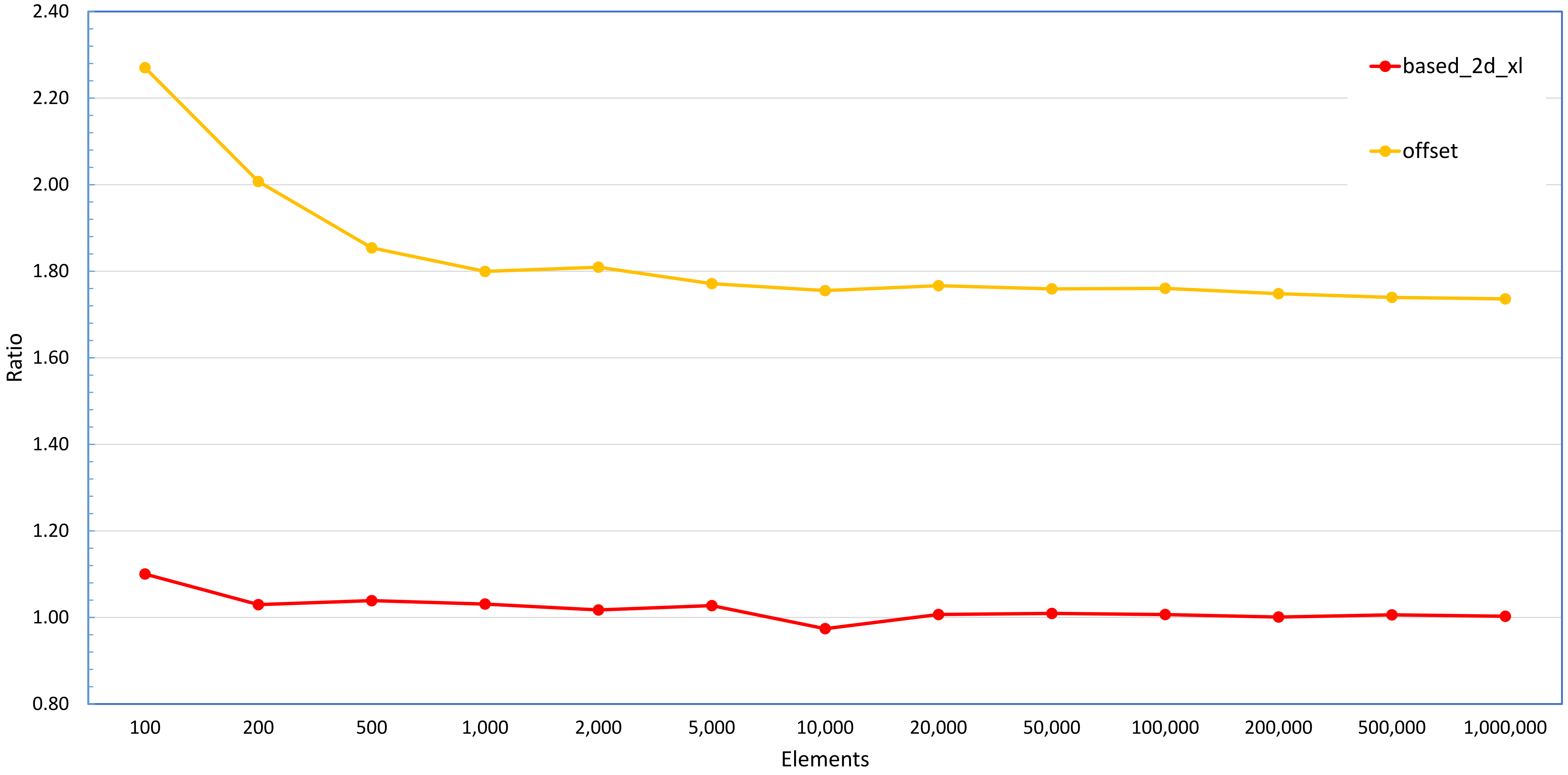
Clang 5.0.1 / `sort()` / `uint64_t`



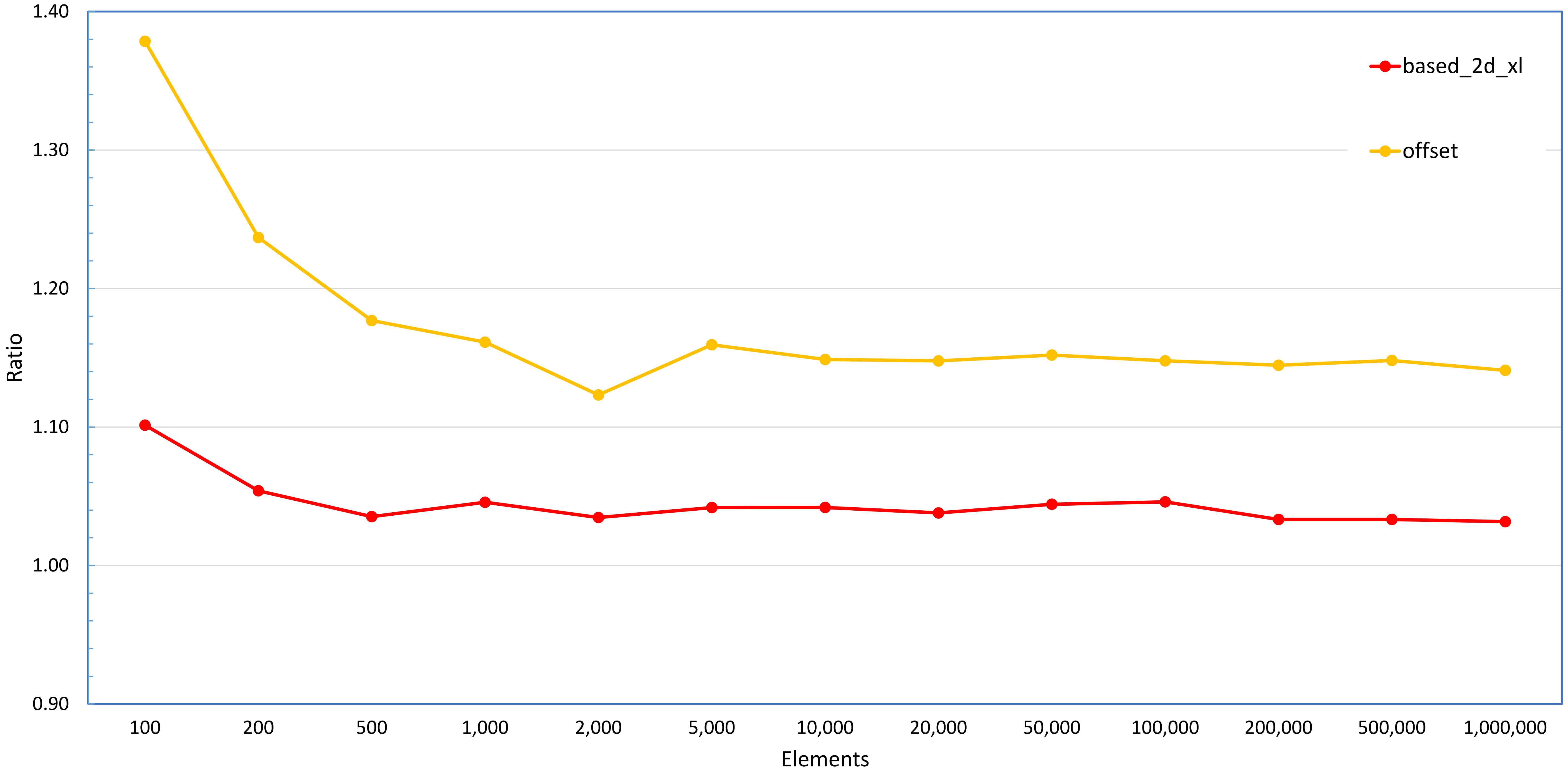
Clang 5.0.1 / `sort()` / `test_struct`



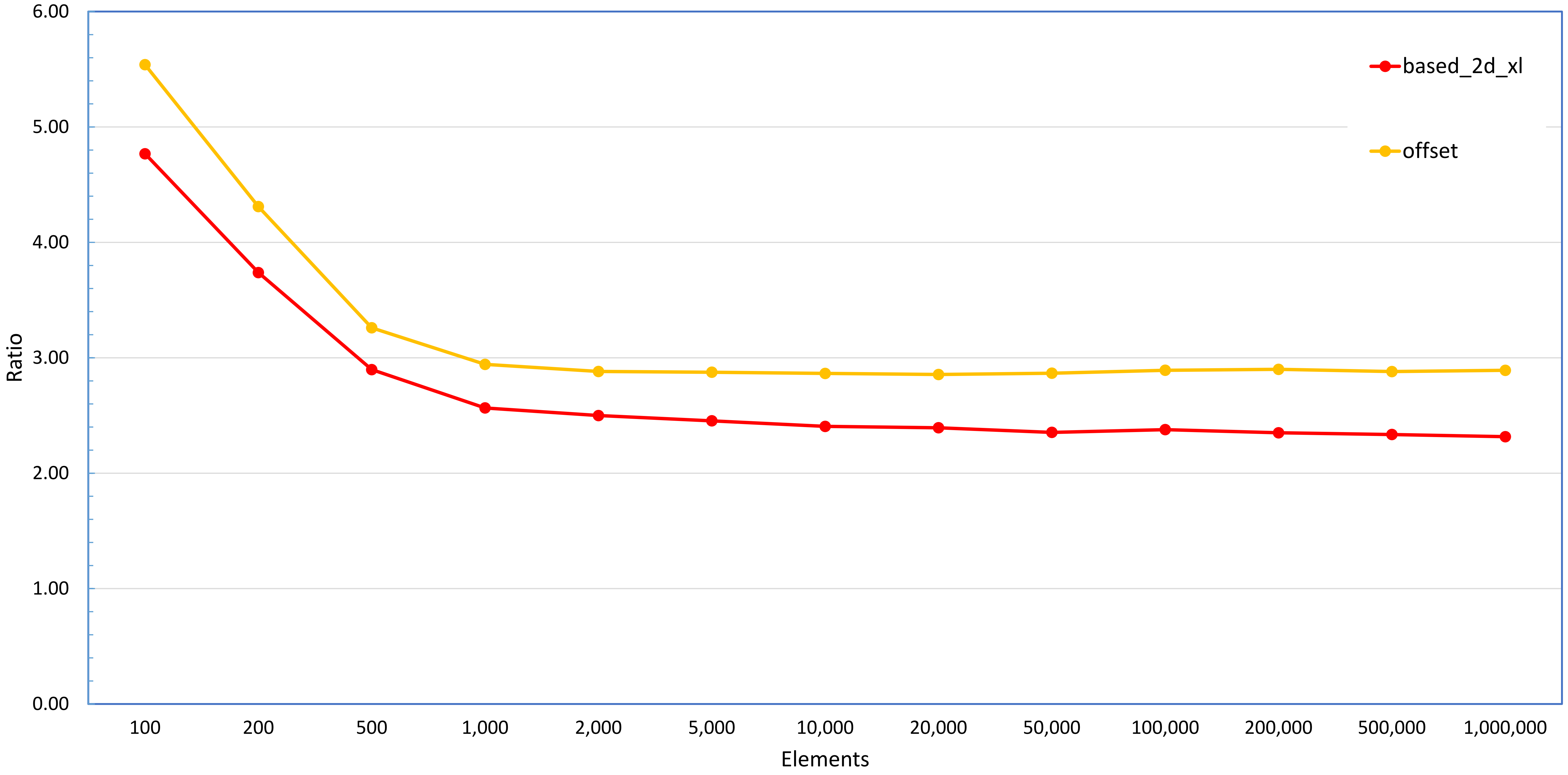
GCC 7.2 / sort() / uint64_t



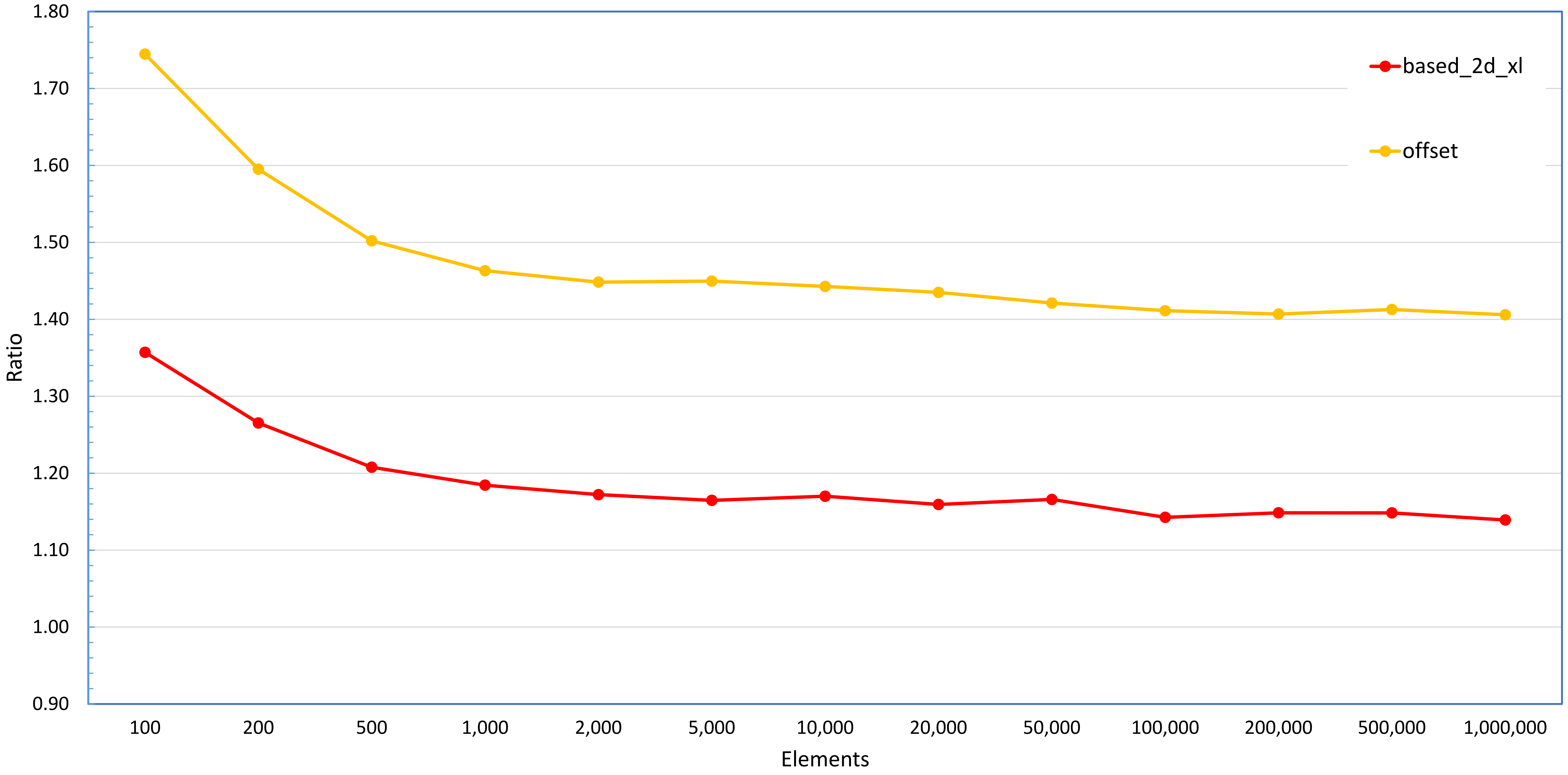
GCC 7.2 / `sort()` / `test_struct`



VS2017 / sort() / uint64_t



VS2017 / sort() / test_struct



Self-Contained DOM Demo

Self-Contained DOM Demo

```
template<size_t N>
struct scd_raw_heap
{
    size_t      m_hwm;          //- High water mark
    uint8_t     m_buf[N];       //- The fixed-size segment

    using void_pointer = syn_ptr<void, offset_addressing_model>;

    scd_raw_heap();

    void_pointer    allocate(size_t n);
    void            deallocate(void_pointer p);
    static  size_t  round_up(size_t x, size_t r);
};

template<size_t N> inline typename scd_raw_heap<N>::void_pointer
scd_raw_heap<N>::allocate(size_t n)
{
    void_pointer    p(m_buf + m_hwm);
    m_hwm += round_up(n, 8);
    return p;
}
```

Self-Contained DOM Demo

```
template<class T, size_t N>
class scd_allocator
{
  public:
    ...
    using heap_type              = scd_raw_heap<N>;
    using difference_type        = ptrdiff_t;
    using size_type              = size_t;
    using void_pointer           = syn_ptr<void, offset_addressing_model>;
    using const_void_pointer     = syn_ptr<void const, offset_addressing_model>;
    using pointer                = syn_ptr<T, offset_addressing_model>;
    using const_pointer          = syn_ptr<T const, offset_addressing_model>;
    using reference              = T&;
    using const_reference        = T const&;
    using value_type             = T;

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

Self-Contained DOM Demo

```
template<class T, size_t N>
class scd_allocator
{
  public:
    ~scd_allocator() = default;

    scd_allocator() = default;
    scd_allocator(const scd_allocator& src) noexcept = default;
    template<class U>
    scd_allocator(const scd_allocator<U, N>& src) noexcept;
    scd_allocator(scd_raw_heap<N>* pheap);

    scd_allocator&  operator =(const scd_allocator& vRhs) noexcept = default;

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

Self-Contained DOM Demo

```
template<class T, size_t N>
class scd_allocator
{
    ...

  private:
    using heap_pointer = syn_ptr<scd_raw_heap<N>, offset_addressing_model>;

    heap_pointer    mp_heap;
};
```

Self-Contained DOM Demo

```
template<size_t N>
class scd_message
{
    using heap_type        = scd_raw_heap<N>;

    using syn_string_alloc = scd_allocator<char, N>;
    using syn_string       = simple_string<syn_string_alloc>;

    using syn_list_alloc   = scd_allocator<syn_string, N>;
    using syn_list         = std::list<syn_string, syn_list_alloc>;

    using syn_less         = std::less<syn_string>;
    using syn_pair         = std::pair<syn_string const, syn_list>;
    using syn_map_alloc    = scd_allocator<syn_pair, N>;
    using syn_map          = std::map<syn_string, syn_list, syn_less, syn_map_alloc>;
    ...

};
```

Self-Contained DOM Demo

```
template<size_t N>
class scd_message
{
    using heap_type          = scd_raw_heap<N>;

    using syn_string_alloc = scd_allocator<char, N>;
    using syn_string         = simple_string<syn_string_alloc>;

    using syn_list_alloc   = scd_allocator<syn_string, N>;
    using syn_list           = std::list<syn_string, syn_list_alloc>;

    using syn_less           = std::less<syn_string>;
    using syn_pair           = std::pair<syn_string const, syn_list>;
    using syn_map_alloc    = scd_allocator<syn_pair, N>;
    using syn_map            = std::map<syn_string, syn_list, syn_less, syn_map_alloc>;
    ...

};
```

Self-Contained DOM Demo

```
template<size_t N>
class scd_message
{
    using heap_type        = scd_raw_heap<N>;

    using syn_string_alloc = scd_allocator<char, N>;
    using syn_string       = simple_string<syn_string_alloc>;

    using syn_list_alloc   = scd_allocator<syn_string, N>;
    using syn_list         = std::list<syn_string, syn_list_alloc>;

    using syn_less         = std::less<syn_string>;
    using syn_pair         = std::pair<syn_string const, syn_list>;
    using syn_map_alloc    = scd_allocator<syn_pair, N>;
    using syn_map          = std::map<syn_string, syn_list, syn_less, syn_map_alloc>;
    ...

};
```

Self-Contained DOM Demo

```
template<size_t N>
class scd_message
{
    using heap_type        = scd_raw_heap<N>;

    using syn_string_alloc = scd_allocator<char, N>;
    using syn_string       = simple_string<syn_string_alloc>;

    using syn_list_alloc   = scd_allocator<syn_string, N>;
    using syn_list         = std::list<syn_string, syn_list_alloc>;

    using syn_less         = std::less<syn_string>;
    using syn_pair         = std::pair<syn_string const, syn_list>;
    using syn_map_alloc    = scd_allocator<syn_pair, N>;
    using syn_map          = std::map<syn_string, syn_list, syn_less, syn_map_alloc>;
    ...

};
```

Self-Contained DOM Demo

```
template<size_t N>
class scd_message
{
    ...

  public:
    scd_message();
    scd_message(scd_message const&);

    scd_message&    operator =(scd_message const&);

    void            add_data(int key_start, int val_start, int count);
    void            print_values() const;

  private:
    heap_type   m_heap;
    syn_map     m_map;
};
```

Self-Contained DOM Demo

```
template<size_t N> void
scd_message<N>::add_data(int key_start, int val_start, int count)
{
    char                key_buf[128], val_buf[128];
    syn_string_alloc    str_alloc(&m_heap);
    syn_list_alloc      list_alloc(&m_heap);

    syn_string  key_str(str_alloc);
    syn_string  val_str(str_alloc);
    syn_list    val_list(list_alloc);

    for (int i = val_start;  i < (val_start + count);  ++i)
    {
        sprintf(val_buf, "this is value string #%d", i+100);
        val_str.assign(val_buf);
        val_list.push_back(std::move(val_str));
    }
    sprintf(key_buf, "this is key string #%d", key_start);
    key_str.assign(key_buf);
    m_map.emplace(std::move(key_str), std::move(val_list));
}
```

Self-Contained DOM Demo

```
template<size_t N> void
scd_message<N>::add_data(int key_start, int val_start, int count)
{
    char                    key_buf[128], val_buf[128];
    syn_string_alloc    str_alloc(&m_heap);
    syn_list_alloc      list_alloc(&m_heap);

    syn_string  key_str(str_alloc);
    syn_string  val_str(str_alloc);
    syn_list    val_list(list_alloc);

    for (int i = val_start;  i < (val_start + count);  ++i)
    {
        sprintf(val_buf, "this is value string #%d", i+100);
        val_str.assign(val_buf);
        val_list.push_back(std::move(val_str));
    }
    sprintf(key_buf, "this is key string #%d", key_start);
    key_str.assign(key_buf);
    m_map.emplace(std::move(key_str), std::move(val_list));
}
```

Self-Contained DOM Demo

```
template<size_t N> void
scd_message<N>::add_data(int key_start, int val_start, int count)
{
    char                key_buf[128], val_buf[128];
    syn_string_alloc    str_alloc(&m_heap);
    syn_list_alloc      list_alloc(&m_heap);

    syn_string  key_str(str_alloc);
    syn_string  val_str(str_alloc);
    syn_list    val_list(list_alloc);

    for (int i = val_start;  i < (val_start + count);  ++i)
    {
        sprintf(val_buf, "this is value string #%d", i+100);
        val_str.assign(val_buf);
        val_list.push_back(std::move(val_str));
    }
    sprintf(key_buf, "this is key string #%d", key_start);
    key_str.assign(key_buf);
    m_map.emplace(std::move(key_str), std::move(val_list));
}
```

Self-Contained DOM Demo

```
template<size_t N> void
scd_message<N>::add_data(int key_start, int val_start, int count)
{
    char                    key_buf[128], val_buf[128];
    syn_string_alloc    str_alloc(&m_heap);
    syn_list_alloc      list_alloc(&m_heap);

    syn_string  key_str(str_alloc);
    syn_string  val_str(str_alloc);
    syn_list    val_list(list_alloc);

    for (int i = val_start;  i < (val_start + count);  ++i)
    {
        sprintf(val_buf, "this is value string #%d", i+100);
        val_str.assign(val_buf);
        val_list.push_back(std::move(val_str));
    }
    sprintf(key_buf, "this is key string #%d", key_start);
    key_str.assign(key_buf);
    m_map.emplace(std::move(key_str), std::move(val_list));
}
```

Self-Contained DOM Demo

```
template<size_t N> void
scd_message<N>::add_data(int key_start, int val_start, int count)
{
    char                    key_buf[128], val_buf[128];
    syn_string_alloc    str_alloc(&m_heap);
    syn_list_alloc      list_alloc(&m_heap);

    syn_string  key_str(str_alloc);
    syn_string  val_str(str_alloc);
    syn_list    val_list(list_alloc);

    for (int i = val_start;  i < (val_start + count);  ++i)
    {
        sprintf(val_buf, "this is value string #%d", i+100);
        val_str.assign(val_buf);
        val_list.push_back(std::move(val_str));
    }
    sprintf(key_buf, "this is key string #%d", key_start);
    key_str.assign(key_buf);
    m_map.emplace(std::move(key_str), std::move(val_list));
}
```

Self-Contained DOM Demo

```
void test_scd()
{
    scd_message<8192>   msg;

    for (int i = 0;  i < 3;  ++i)
    {
        msg.add_data((i+1)*10, (i+1)*200, 4);
    }
    msg.print_values();

    char    bytes[sizeof(scd_message<8192>)];
    memcpy(&bytes[0], &msg, sizeof(scd_message<8192>));

    auto const*     pmsg = reinterpret_cast<scd_message<8192>*>(&bytes[0]);
    pmsg->print_values();

    std::vector<char>   vmsg(sizeof(scd_message<8192>));
    memcpy(vmsg.data(), &msg, sizeof(scd_message<8192>));

    auto const*     pmsg2 = reinterpret_cast<scd_message<8192>*>(vmsg.data());
    pmsg2->print_values();
}
```

Self-Contained DOM Demo

```
void test_scd()
{
    scd_message<8192>   msg;

    for (int i = 0;  i < 3;  ++i)
    {
        msg.add_data((i+1)*10, (i+1)*200, 4);
    }
    msg.print_values();

    char    bytes[sizeof(scd_message<8192>)];
    memcpy(&bytes[0], &msg, sizeof(scd_message<8192>));

    auto const*     pmsg = reinterpret_cast<scd_message<8192>*>(&bytes[0]);
    pmsg->print_values();

    std::vector<char>   vmsg(sizeof(scd_message<8192>));
    memcpy(vmsg.data(), &msg, sizeof(scd_message<8192>));

    auto const*     pmsg2 = reinterpret_cast<scd_message<8192>*>(vmsg.data());
    pmsg2->print_values();
}
```

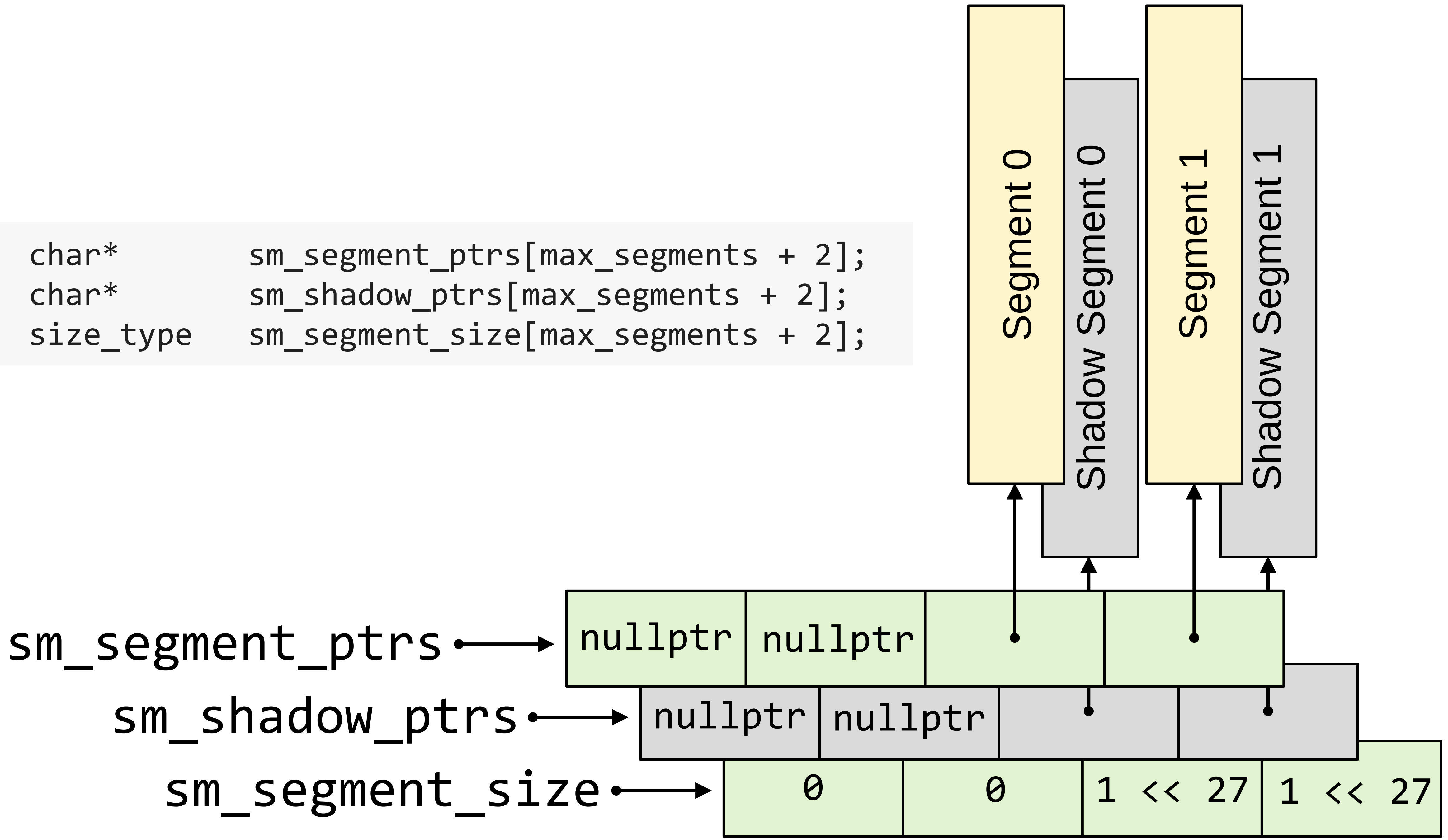
Self-Contained DOM Demo

```
void test_scd()
{
    scd_message<8192>   msg;

    for (int i = 0;  i < 3;  ++i)
    {
        msg.add_data((i+1)*10, (i+1)*200, 4);
    }
    msg.print_values();

    char    bytes[sizeof(scd_message<8192>)];
    memcpy(&bytes[0], &msg, sizeof(scd_message<8192>));

    auto const*     pmsg = reinterpret_cast<scd_message<8192>*>(&bytes[0]);
    pmsg->print_values();

    std::vector<char>   vmsg(sizeof(scd_message<8192>));
    memcpy(vmsg.data(), &msg, sizeof(scd_message<8192>));

    auto const*     pmsg2 = reinterpret_cast<scd_message<8192>*>(vmsg.data());
    pmsg2->print_values();
}
```

2D Relocation Demo

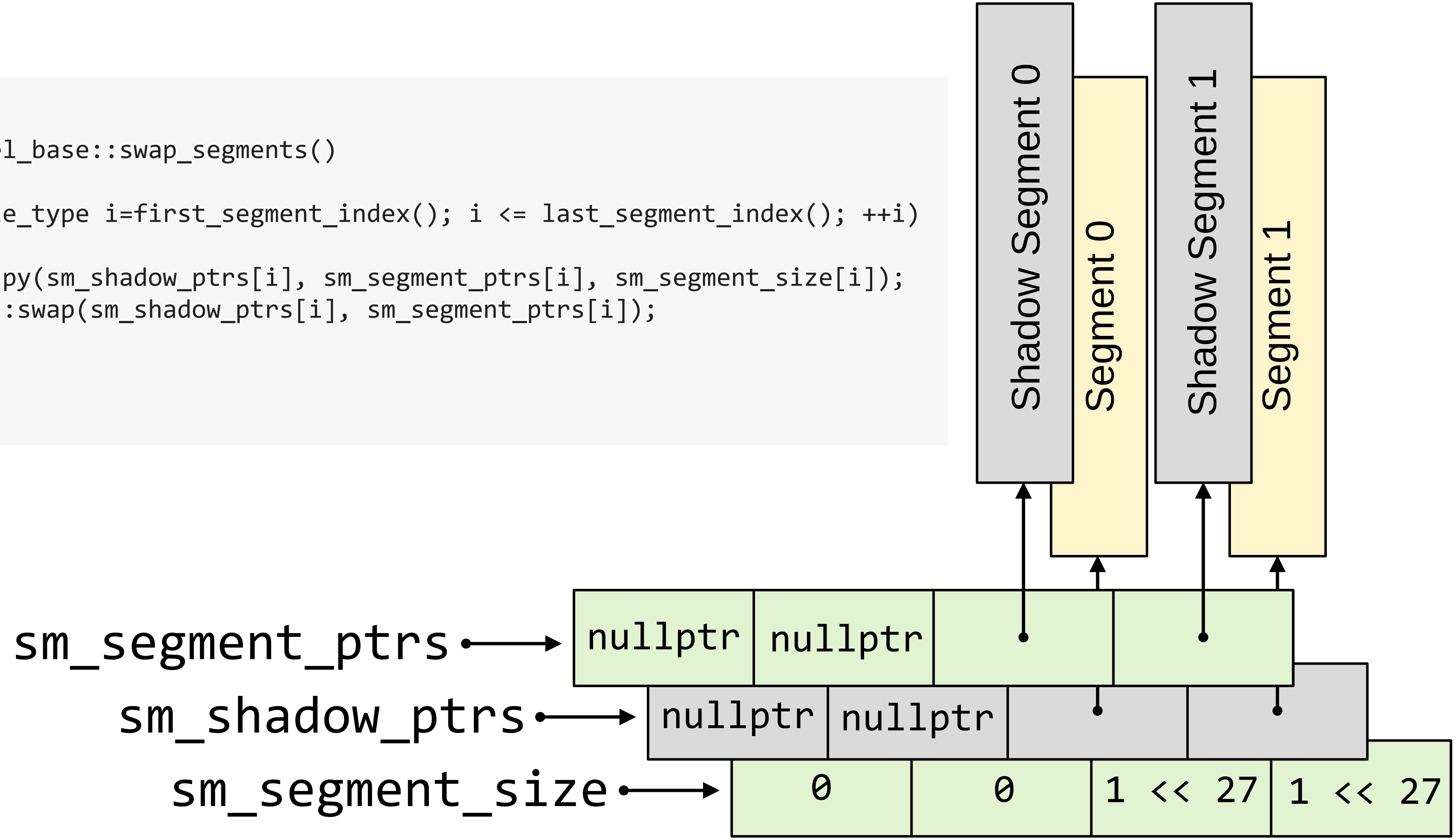
Storage Model Base Class – Segment Swap

```
static   char*        sm_segment_ptrs[max_segments + 2];
static   char*        sm_shadow_ptrs[max_segments + 2];
static   size_type    sm_segment_size[max_segments + 2];
```



Storage Model Base Class – Segment Swap

```
void
storage_model_base::swap_segments()
{
    for (size_type i=first_segment_index(); i <= last_segment_index(); ++i)
    {
        memcpy(sm_shadow_ptrs[i], sm_segment_ptrs[i], sm_segment_size[i]);
        std::swap(sm_shadow_ptrs[i], sm_segment_ptrs[i]);
    }
}
```



Monotonic Allocation Strategy

```
template<class SM>
class monotonic_allocation_strategy
{
  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       = syn_ptr<void, addressing_model>;
    using const_void_pointer = syn_ptr<void const, addressing_model>;

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

  public:
    void_pointer    allocate(size_type n);
    void            deallocate(void_pointer) {};

    static  void    reset_buffers();
    static  void    swap_buffers();
};
```

Allocator rhx_allocator

```
template<class T, class AS>
class rhx_allocator
{
  public:
    using propagate_on_container_copy_assignment = std::true_type;
    using propagate_on_container_move_assignment = std::true_type;
    using propagate_on_container_swap            = std::true_type;

    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;
    ...
};
```

Allocator rhx_allocator

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

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

    ...

    pointer     allocate(size_type n);
    pointer     allocate(size_type n, const_void_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;
};
```

Allocator rhx_allocator – Allocation and Deallocation

```
template<class T, class AS> inline
typename rhx_allocator<T, AS>::pointer
rhx_allocator<T, AS>::allocate(size_type n)
{
    return static_cast<pointer>(m_heap.allocate(n * sizeof(T)));
}


template<class T, class AS> inline void
rhx_allocator<T, AS>::deallocate(pointer p, size_type)
{
    m_heap.deallocate(p);
}
```

Relocatable Heap Demo

```
template<typename AllocStrategy>
void  do_map_test(char const* strategy_name, bool do_reloc)
{
    //- Various type aliases to aid readability.
    //
    using strategy   = AllocStrategy;

    using syn_string = simple_string<rhx_allocator<char, strategy>>;

    using syn_list   = std::list<syn_string, rhx_allocator<syn_string, strategy>>;

    using syn_less   = std::less<syn_string>;
    using syn_pair   = std::pair<const syn_string, syn_list>;
    using syn_alloc  = rhx_allocator<syn_pair, strategy>;
    using syn_map    = std::map<syn_string, syn_list, syn_less, syn_alloc>;

    auto    spmap = allocate<syn_map, strategy>();
    auto    spkey = allocate<syn_string, strategy>();
    auto    spval = allocate<syn_string, strategy>();
    ...
}
```

Relocatable Heap Demo

```
template<typename AllocStrategy>
void  do_map_test(char const* strategy_name, bool do_reloc)
{
    //- Various type aliases to aid readability.
    //
    using strategy   = AllocStrategy;

    using syn_string = simple_string<rhx_allocator<char, strategy>>;

    using syn_list   = std::list<syn_string, rhx_allocator<syn_string, strategy>>;

    using syn_less   = std::less<syn_string>;
    using syn_pair   = std::pair<const syn_string, syn_list>;
    using syn_alloc  = rhx_allocator<syn_pair, strategy>;
    using syn_map    = std::map<syn_string, syn_list, syn_less, syn_alloc>;

    auto    spmap = allocate<syn_map, strategy>();
    auto    spkey = allocate<syn_string, strategy>();
    auto    spval = allocate<syn_string, strategy>();
    ...
}
```

Relocatable Heap Demo

```
template<typename AllocStrategy>
void  do_map_test(char const* strategy_name, bool do_reloc)
{
    //- Various type aliases to aid readability.
    //
    using strategy   = AllocStrategy;

    using syn_string = simple_string<rhx_allocator<char, strategy>>;

    using syn_list   = std::list<syn_string, rhx_allocator<syn_string, strategy>>;

    using syn_less   = std::less<syn_string>;
    using syn_pair   = std::pair<const syn_string, syn_list>;
    using syn_alloc  = rhx_allocator<syn_pair, strategy>;
    using syn_map    = std::map<syn_string, syn_list, syn_less, syn_alloc>;

    auto    spmap = allocate<syn_map, strategy>();
    auto    spkey = allocate<syn_string, strategy>();
    auto    spval = allocate<syn_string, strategy>();
    ...
}
```

Relocatable Heap Demo

```
template<typename AllocStrategy>
void  do_map_test(char const* strategy_name, bool do_reloc)
{
    //- Various type aliases to aid readability.
    //
    using strategy   = AllocStrategy;

    using syn_string = simple_string<rhx_allocator<char, strategy>>;

    using syn_list   = std::list<syn_string, rhx_allocator<syn_string, strategy>>;

    using syn_less   = std::less<syn_string>;
    using syn_pair   = std::pair<const syn_string, syn_list>;
    using syn_alloc  = rhx_allocator<syn_pair, strategy>;
    using syn_map    = std::map<syn_string, syn_list, syn_less, syn_alloc>;

    auto    spmap = allocate<syn_map, strategy>();
    auto    spkey = allocate<syn_string, strategy>();
    auto    spval = allocate<syn_string, strategy>();
    ...
}
```

Relocatable Heap Demo

```
template<typename AllocStrategy>
void  do_map_test(char const* strategy_name, bool do_reloc)
{
    ...
    char    key_str[128], val_str[128];

    for (int i = outer;  i < (outer + 3);  ++i)
    {
        sprintf(key_str, "this is key string #%d", i);
        spkey->assign(key_str);

        for (int j = inner;  j < (inner + 5);  ++j)
        {
            sprintf(val_str, "this is string #%d created for syn_map<syn_string, syn_list>", j);
            spval->assign(val_str);
            (*spmap)[*spkey].push_back(*spval);
        }
    }
    print_map(*spmap);
    ...
}
```

Relocatable Heap Demo

```
template<typename AllocStrategy>
void  do_map_test(char const* strategy_name, bool do_reloc)
{
    ...

    if (do_reloc)
    {
        strategy::swap_segments();
        print_map(*spmap);
    }
}
```

Summary

Comments

- Synthetic pointers and relocatable heaps are like parachutes…
- Possible applications:
  - Relocatable heap for private use
  - Relocatable heap for shared memory
  - Self-contained messages/DOMs
  - Instrumented debug allocator
- This is a work in progress – stay tuned…

Questions?

Thank You for Attending!

Talk: https://github.com/BobSteagall/CppCon2018

Blog: https://bobsteagall.com