accu
2023

TRIVIAL RELOCATION THROUGH TIME

MUNGO GILL

Trivial Relocation Through Time

A historical perspective on *trivial relocation* and `memcpy`

ACCU 2023
April 21, 2023

Mungo Gill
Senior Software Engineer, BDE Team

TechAtBloomberg.com

Bloomberg Engineering

Introduction: Who am I?

- Senior Software Engineer at Bloomberg
- Over 20 years of C++ experience
- Various technology and finance companies

Contents

- What problem are we trying to solve?
- How are libraries working around the issue now?
- What proposals are there to solve this?
    - In the past
    - In the present
- Where do we go from here?
- Questions?

Why do we care about trivial relocation?

- What problem are we trying to solve?
- Some background: a short history lesson
    - The C++03 way
    - The C++11 way
    - Going further

Bloomberg

Engineering

Growing a vector the C++03 way

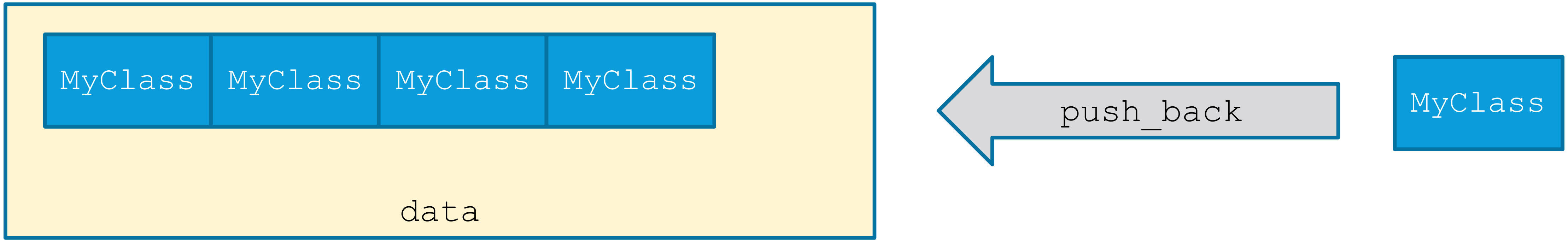
```
class MyClass {
  public:
    MyClass();
    MyClass(const MyClass &);
};

int main()
{
    std::vector<MyClass> data;

    data.push_back(MyClass());
    // … 3 more times

    data.push_back(MyClass());
}
```

What happens here?

Bloomberg

Engineering

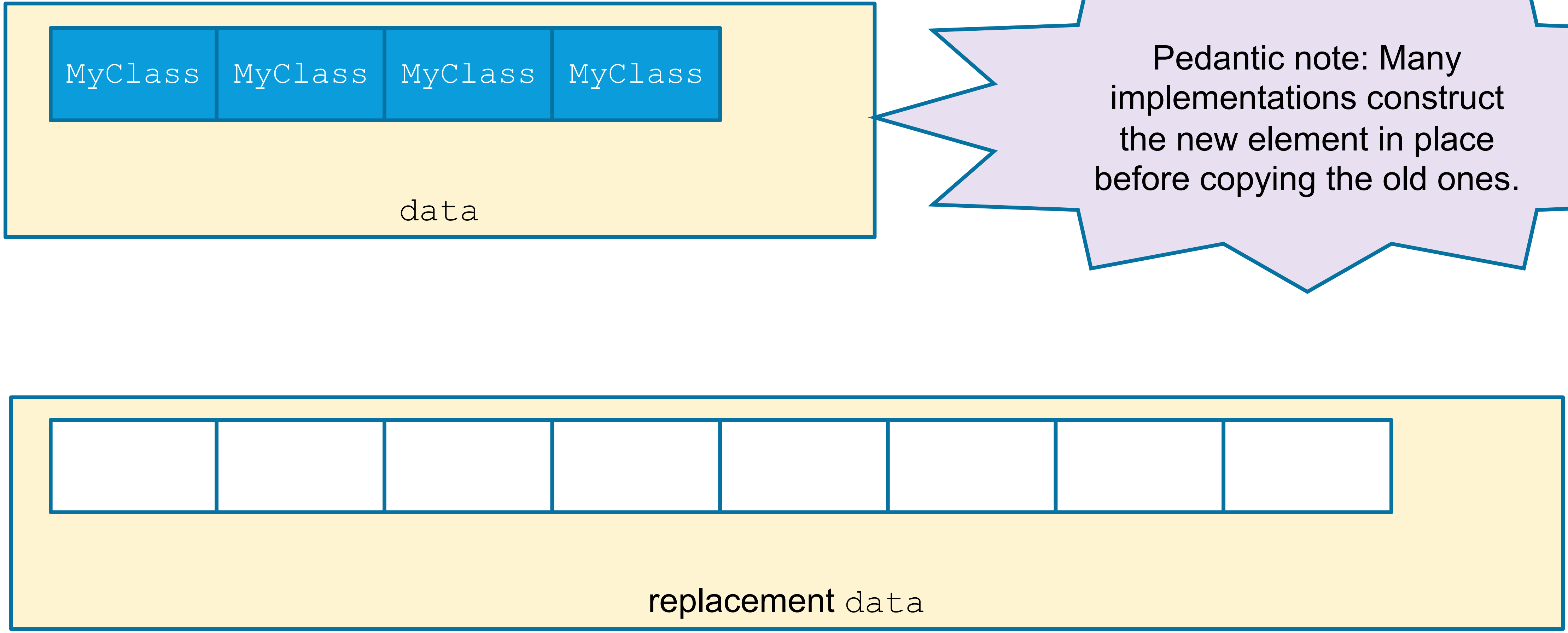
Growing a vector the C++03 way



`data` **is not big enough!**

It needs to grow!

Growing a vector the C++03 way

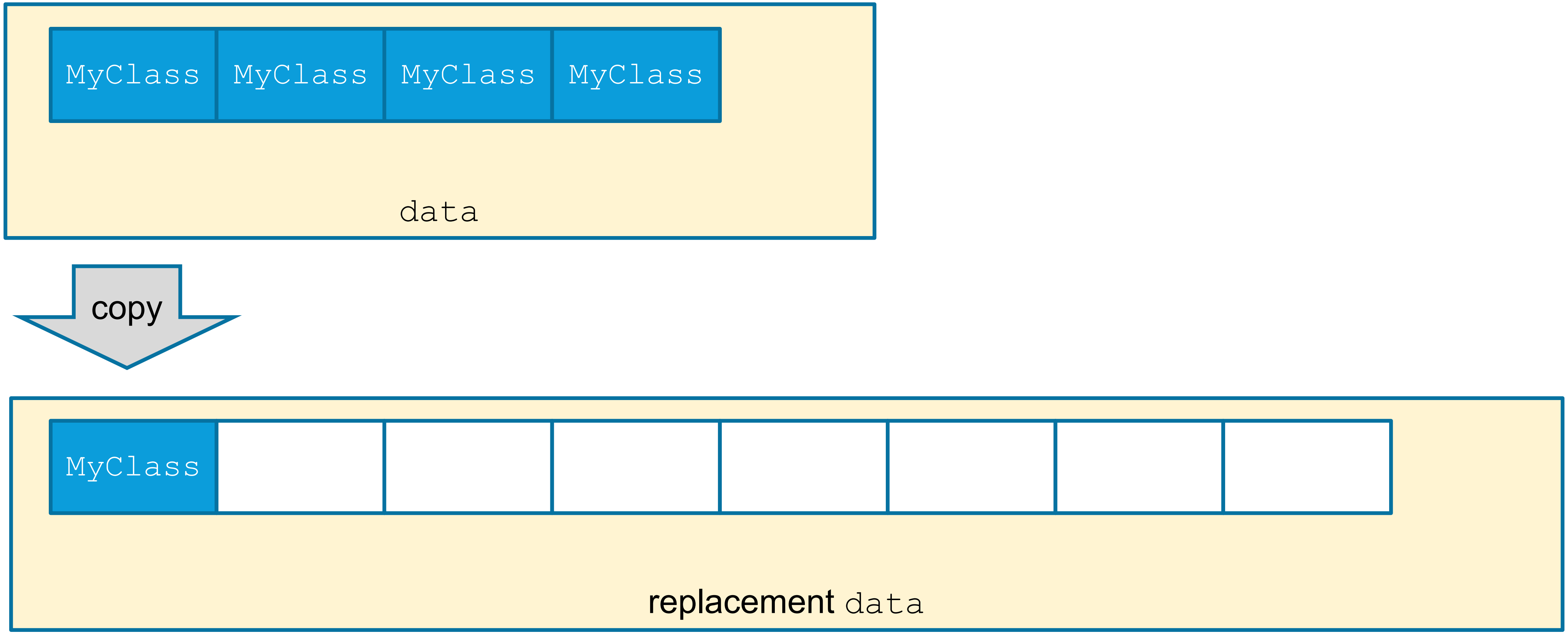


Pedantic note: Many implementations construct the new element in place before copying the old ones.

Bloomberg

Engineering

Growing a vector the C++03 way



MyClass MyClass MyClass MyClass

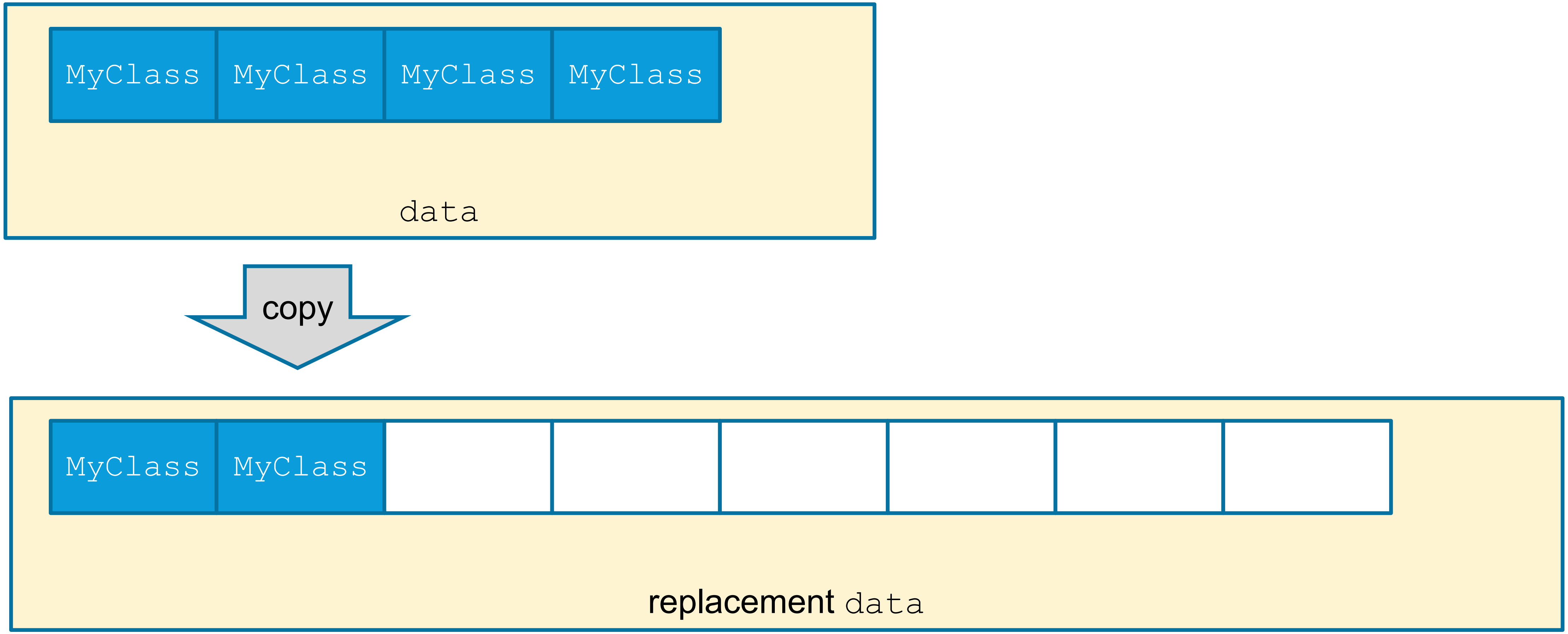
data

copy

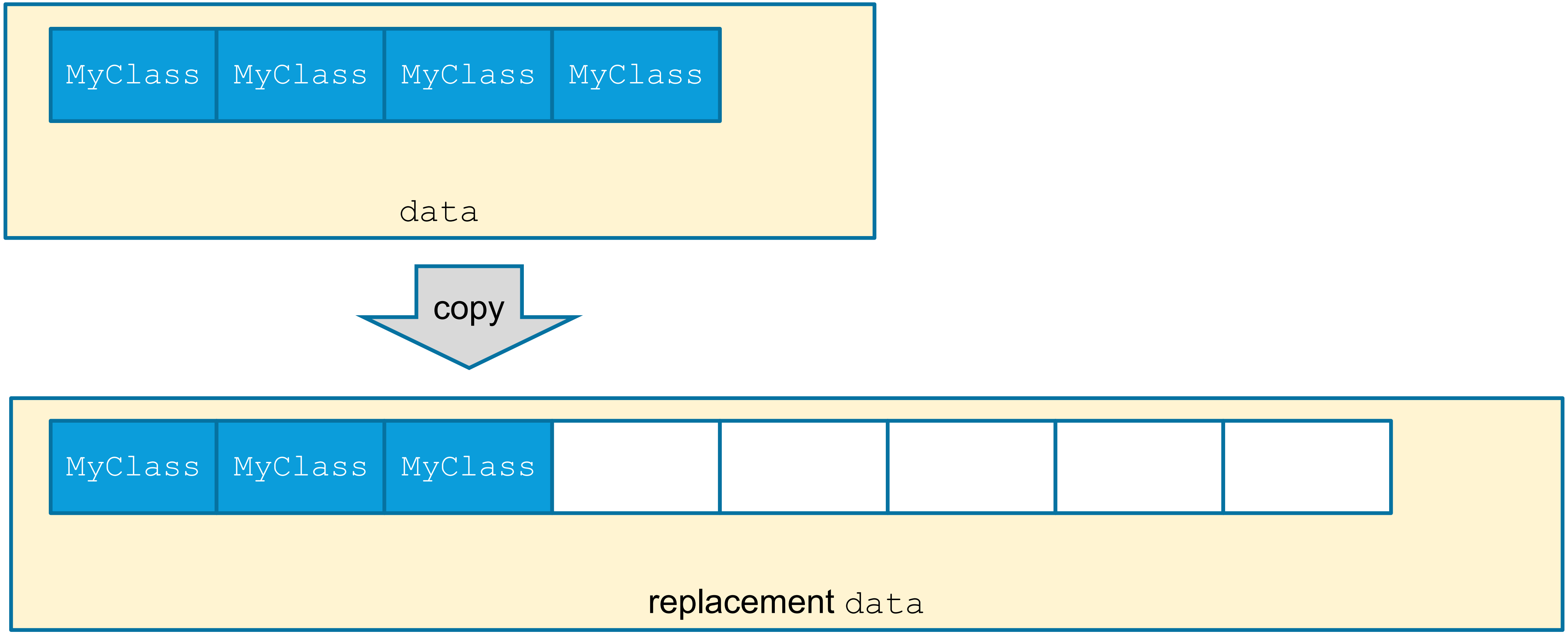
MyClass

replacement data

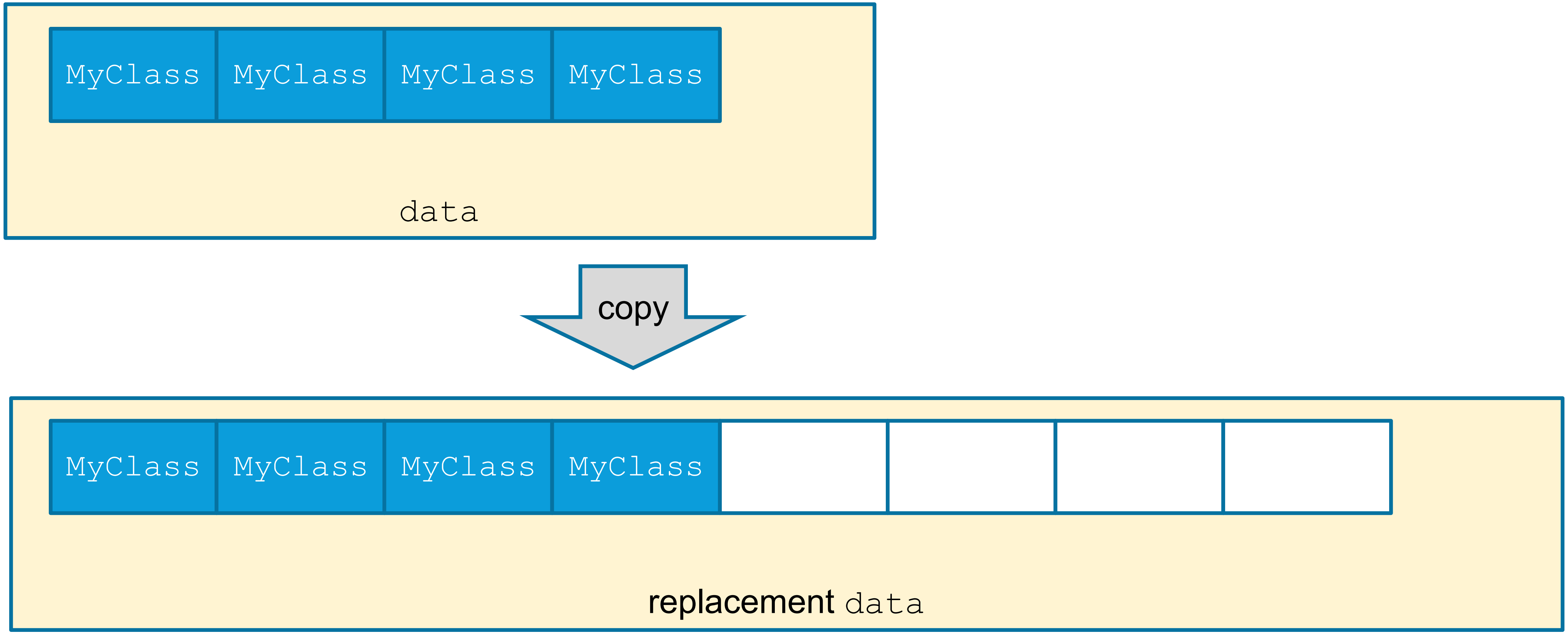
Growing a vector the C++03 way



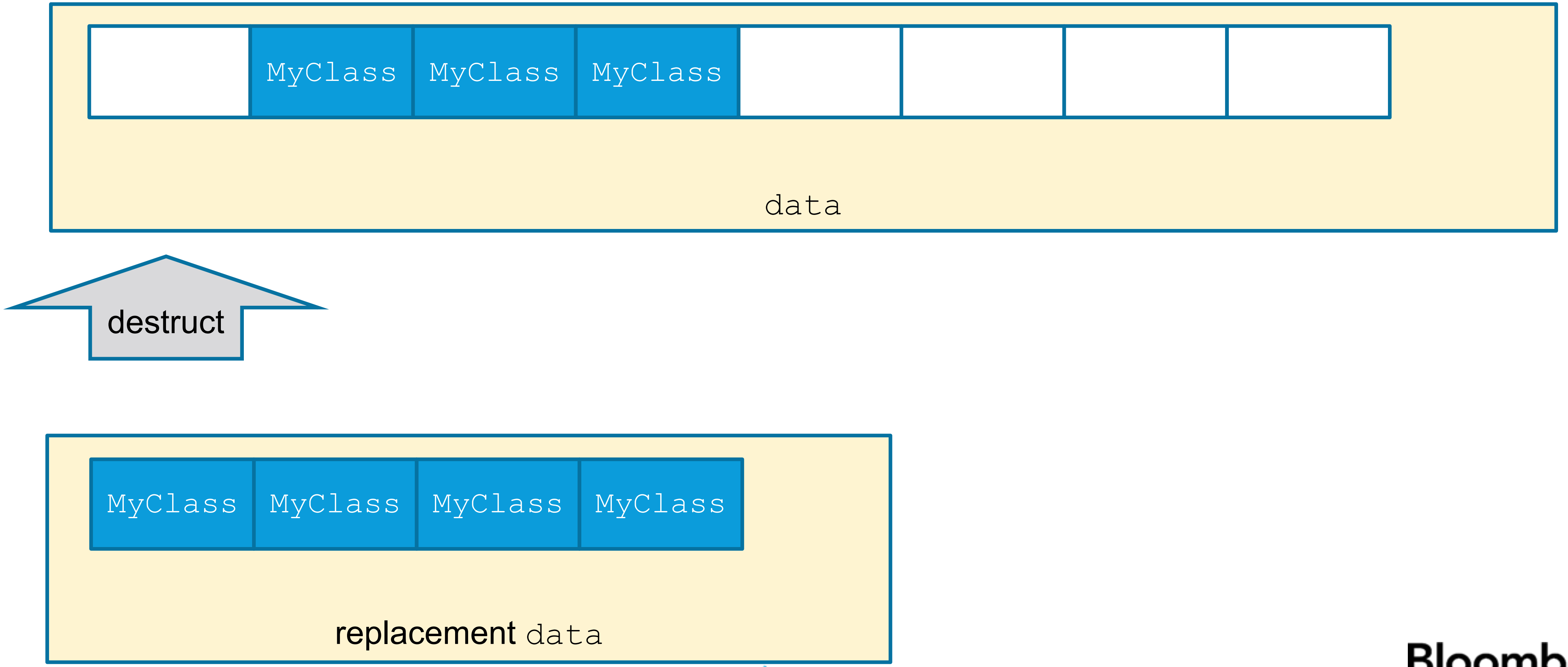
Growing a vector the C++03 way



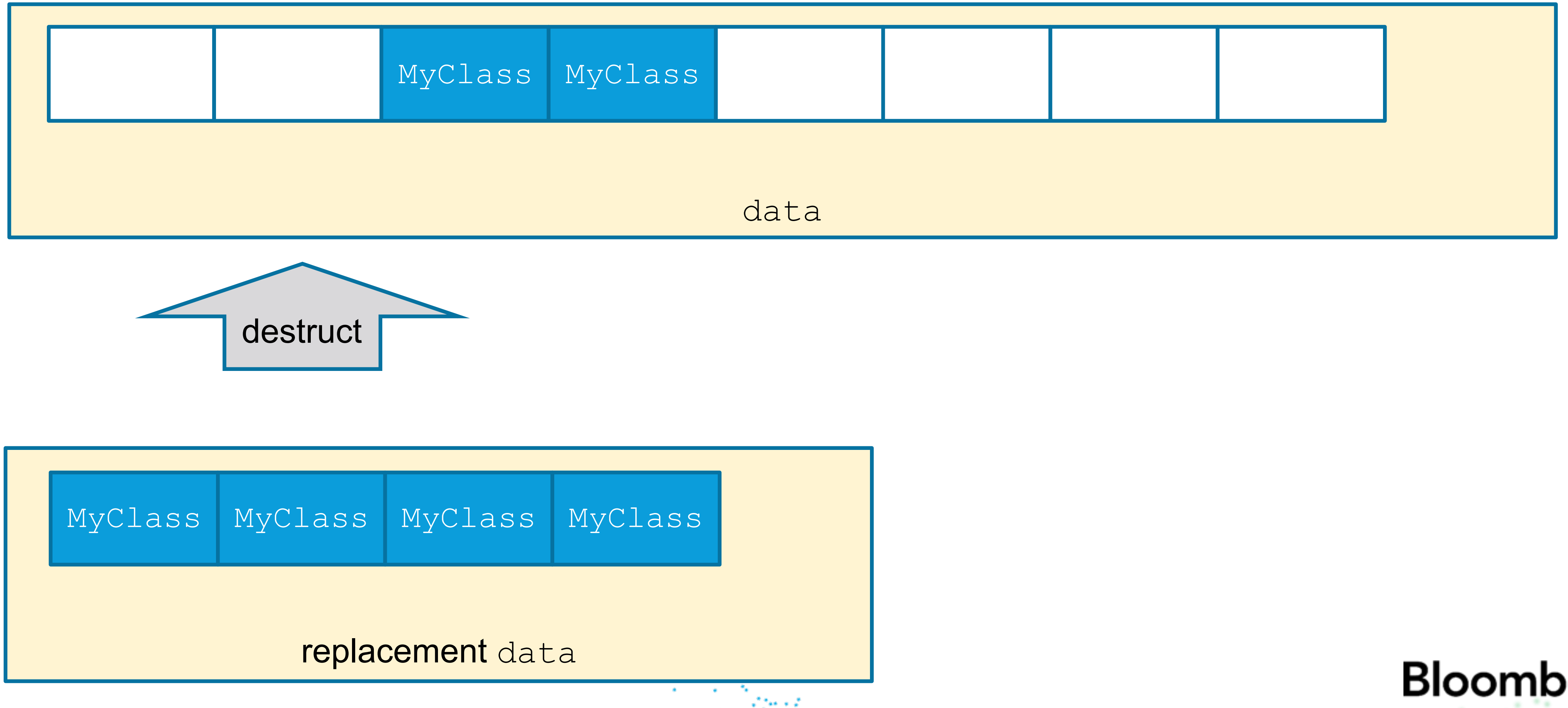
Growing a vector the C++03 way



Growing a vector the C++03 way



Growing a vector the C++03 way



MyClass MyClass

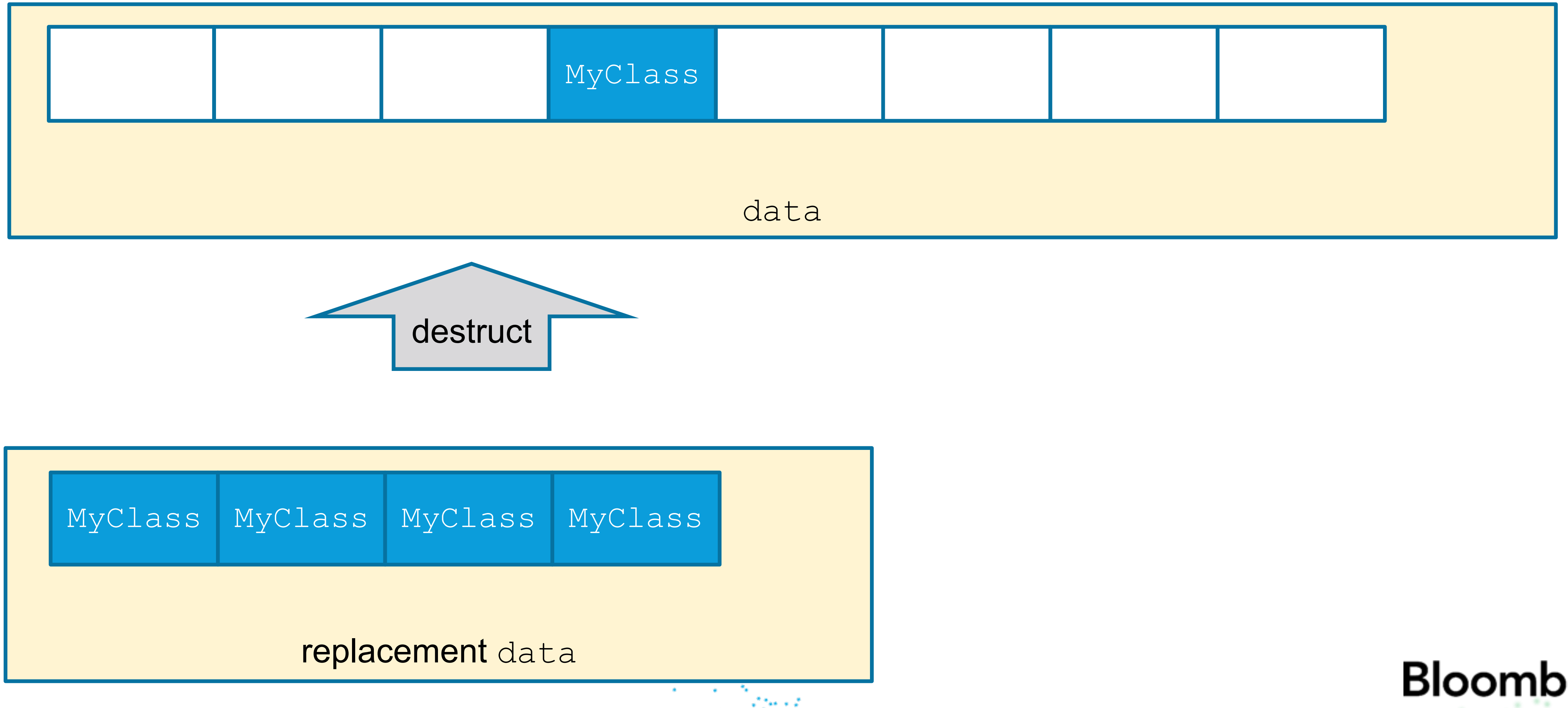
data

destruct

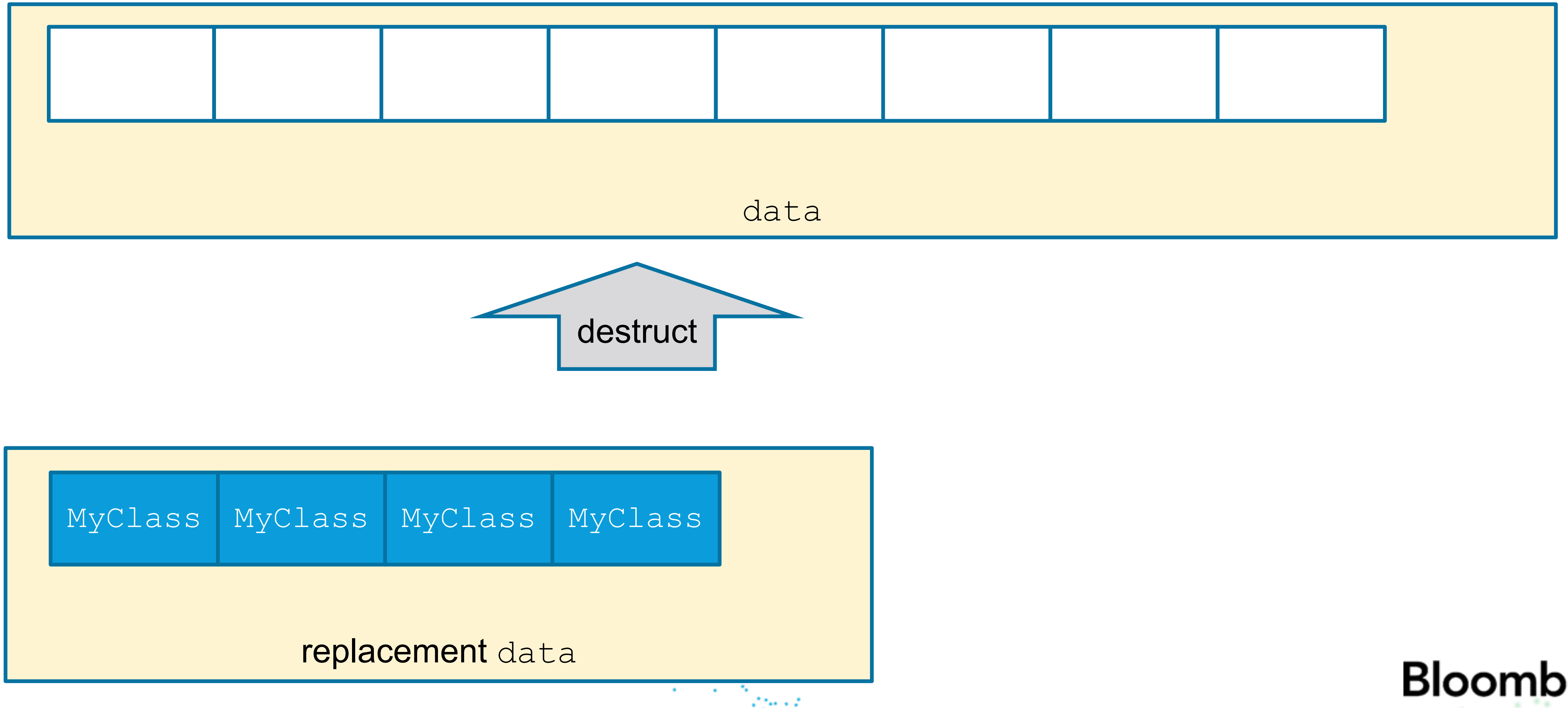
MyClass MyClass MyClass MyClass

replacement data

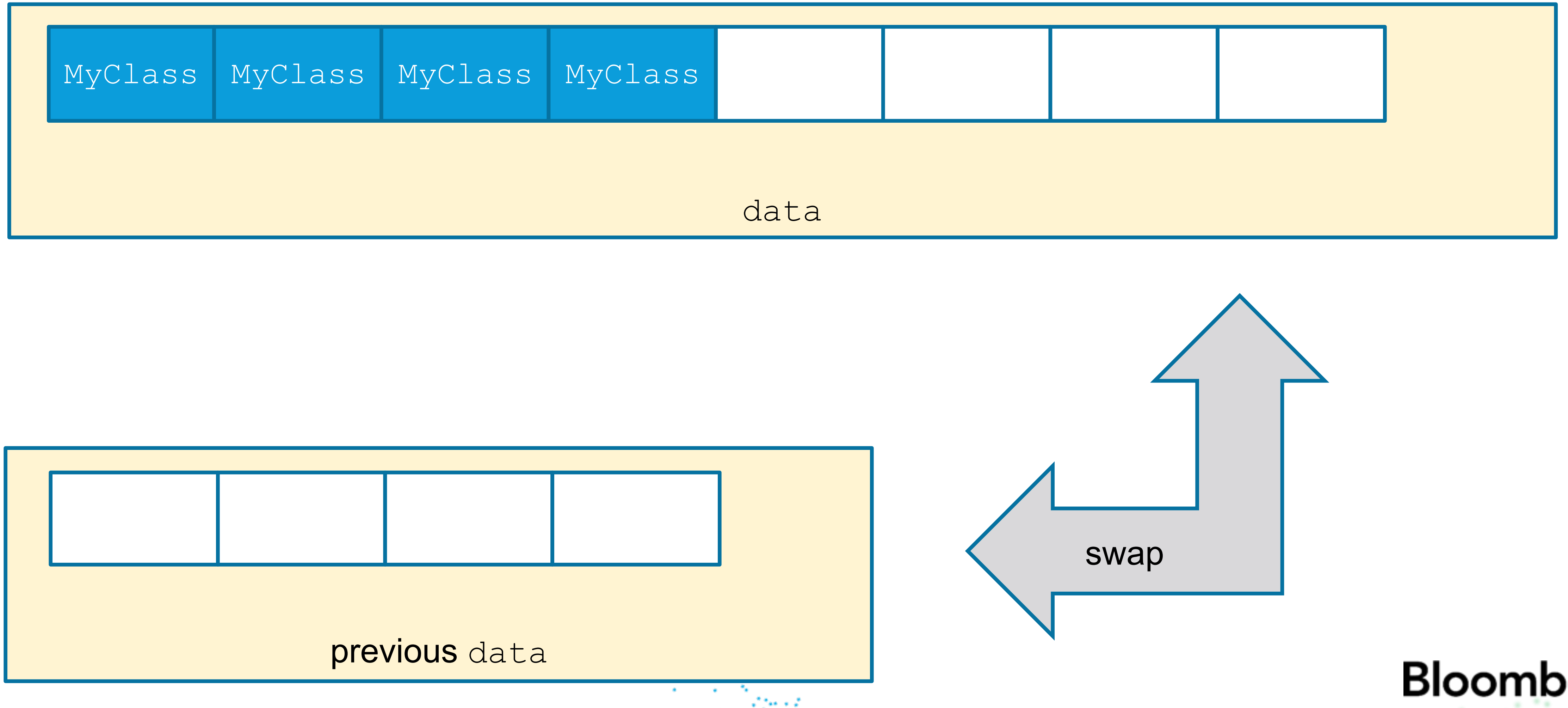
Growing a vector the C++03 way



Growing a vector the C++03 way

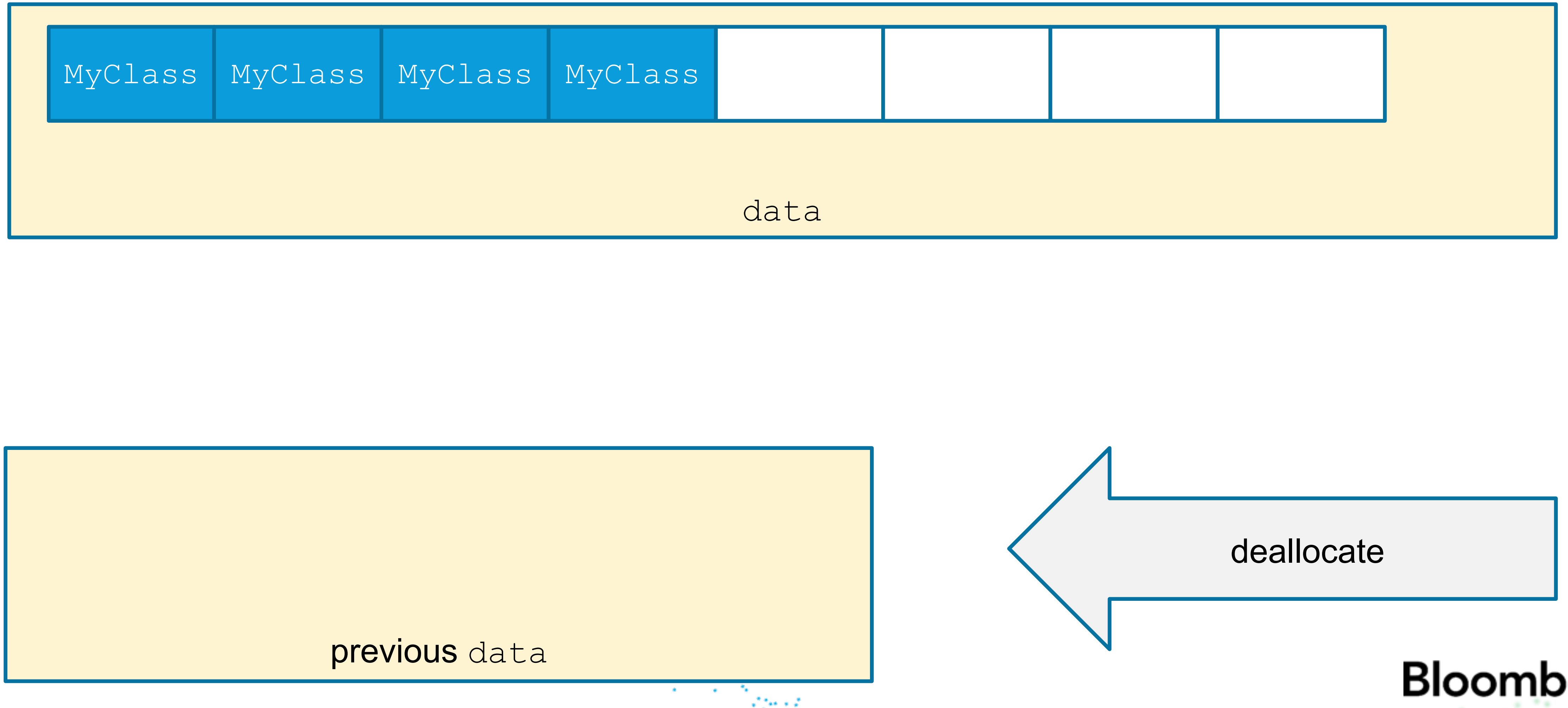


Growing a vector the C++03 way



Bloomberg
Engineering

Growing a vector the C++03 way



How can we make this better?

C++11 gave us move constructors!

Growing a vector the C++11 way

```
class MyClass {
  public:
    MyClass();
    MyClass(const MyClass &);
    MyClass(MyClass &&) noexcept;
};

int main()
{
    std::vector<MyClass> data;

    data.push_back(MyClass());
    // … 3 more times

    data.push_back(MyClass());
}
```

What happens now?

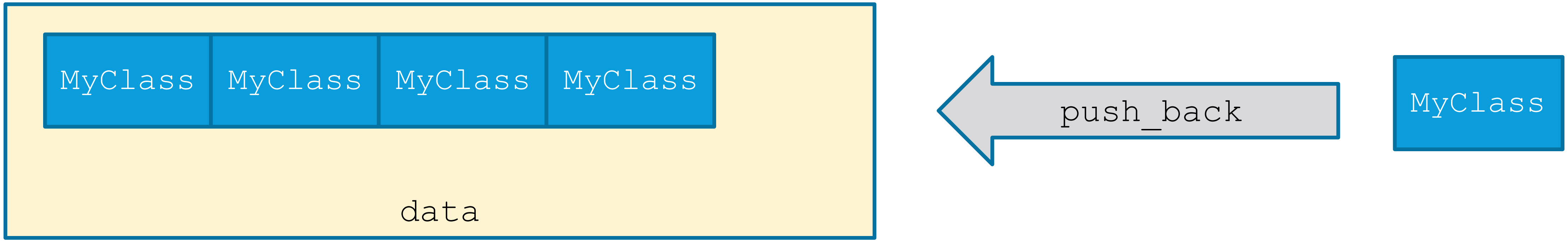
Growing a vector the C++11 way

There are now two possible paths.

1. If MyClass **does not** have a `noexcept` move constructor and is copy constructible, then we do what we did in C++03.

2. If MyClass **does** have a `noexcept` move constructor, then we have a new, more efficient approach.

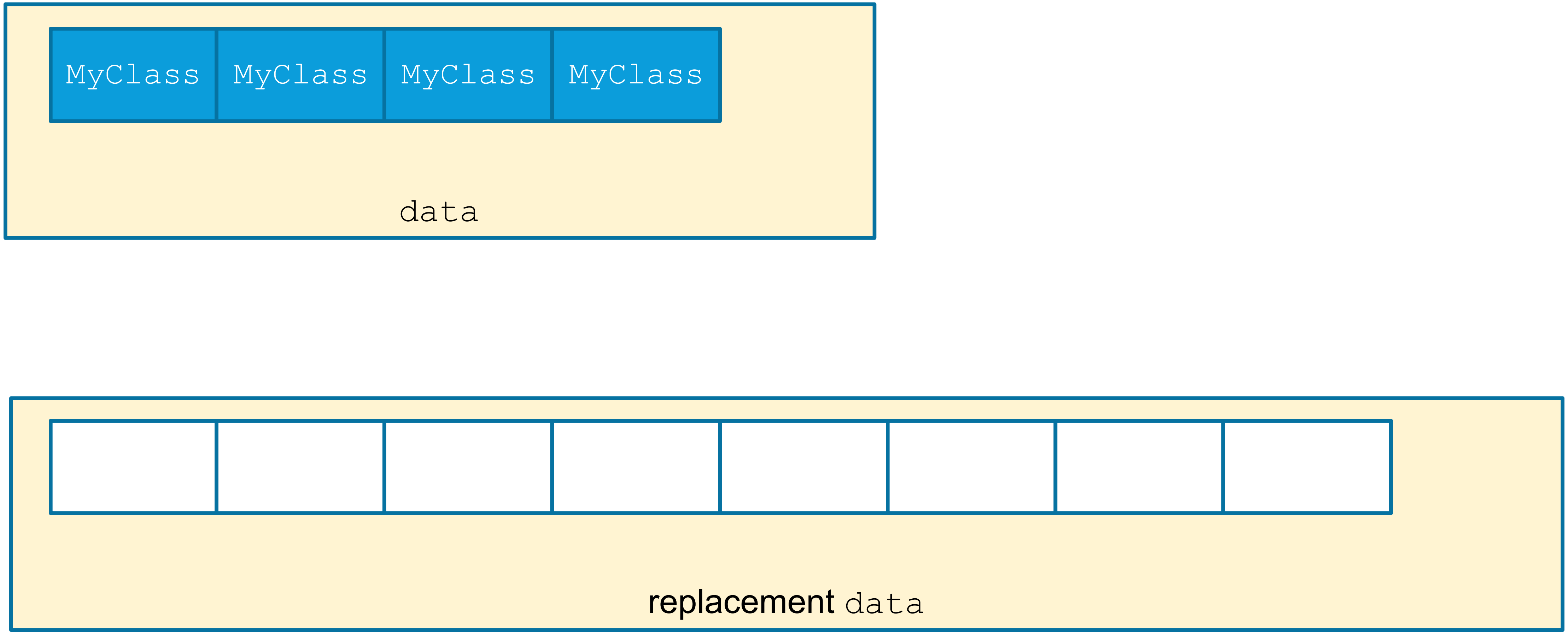
Growing a vector the C++11 way



`data` is not big enough!

It needs to grow!

Growing a vector the C++11 way



MyClass MyClass MyClass MyClass

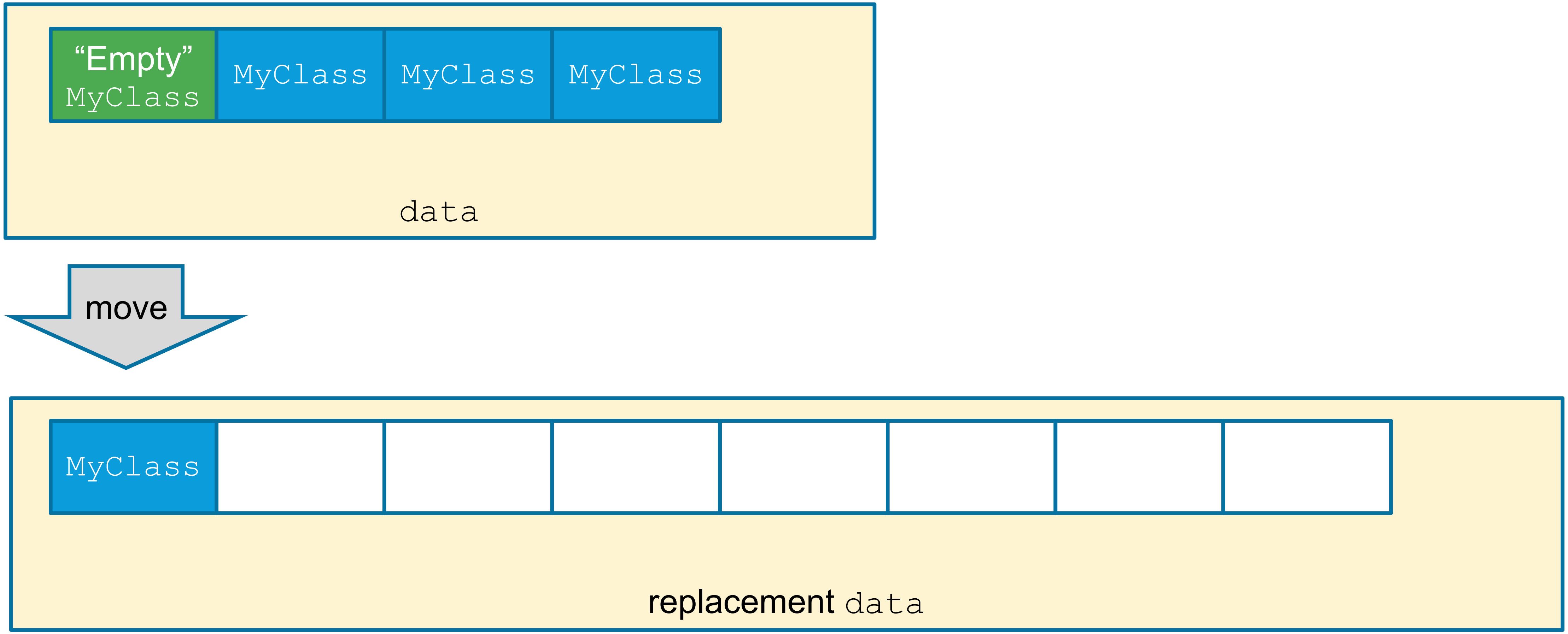
data

replacement data

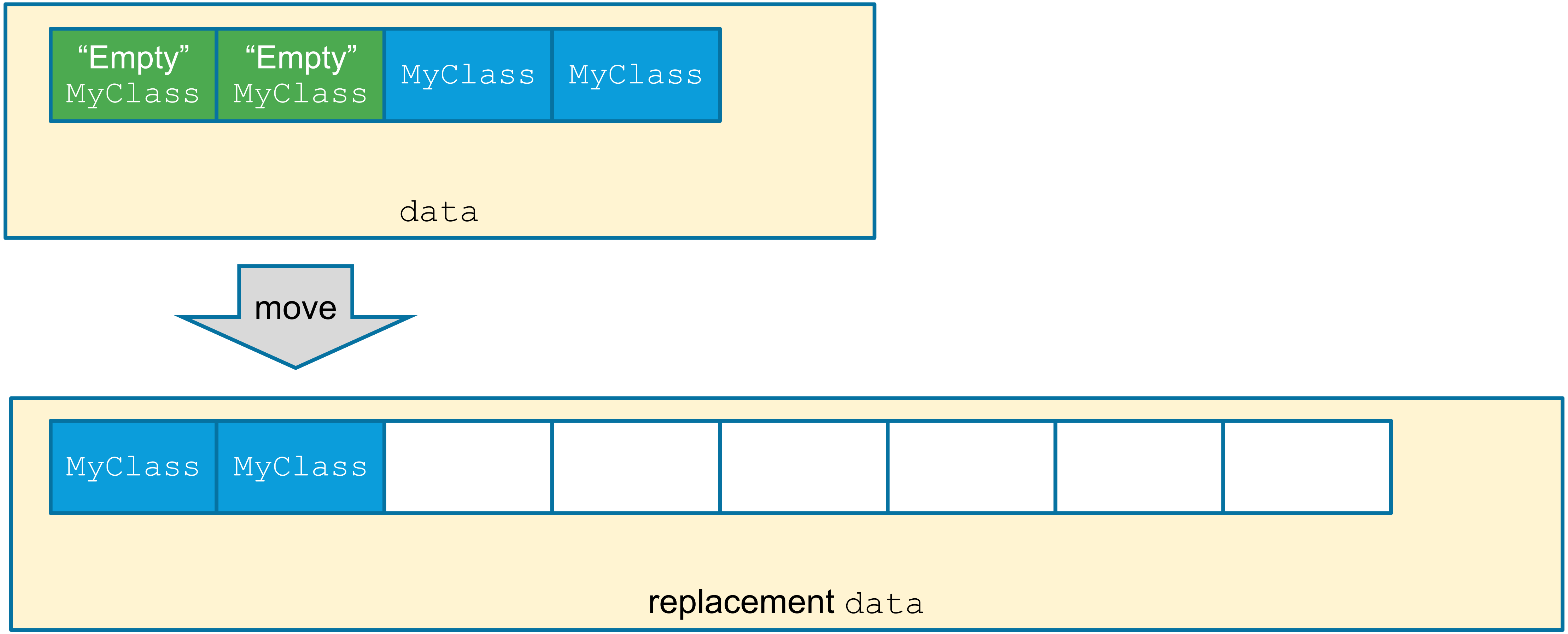
Bloomberg

Engineering

Growing a vector the C++11 way

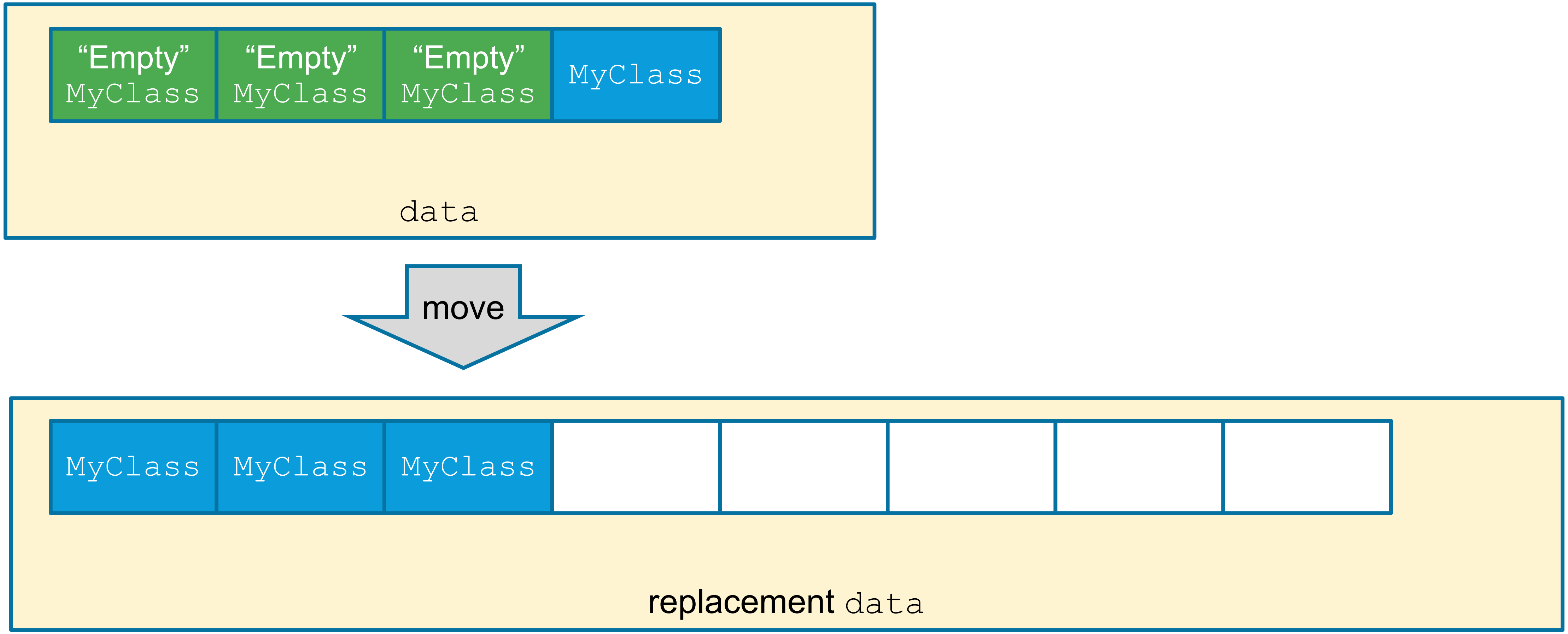


Growing a vector the C++11 way

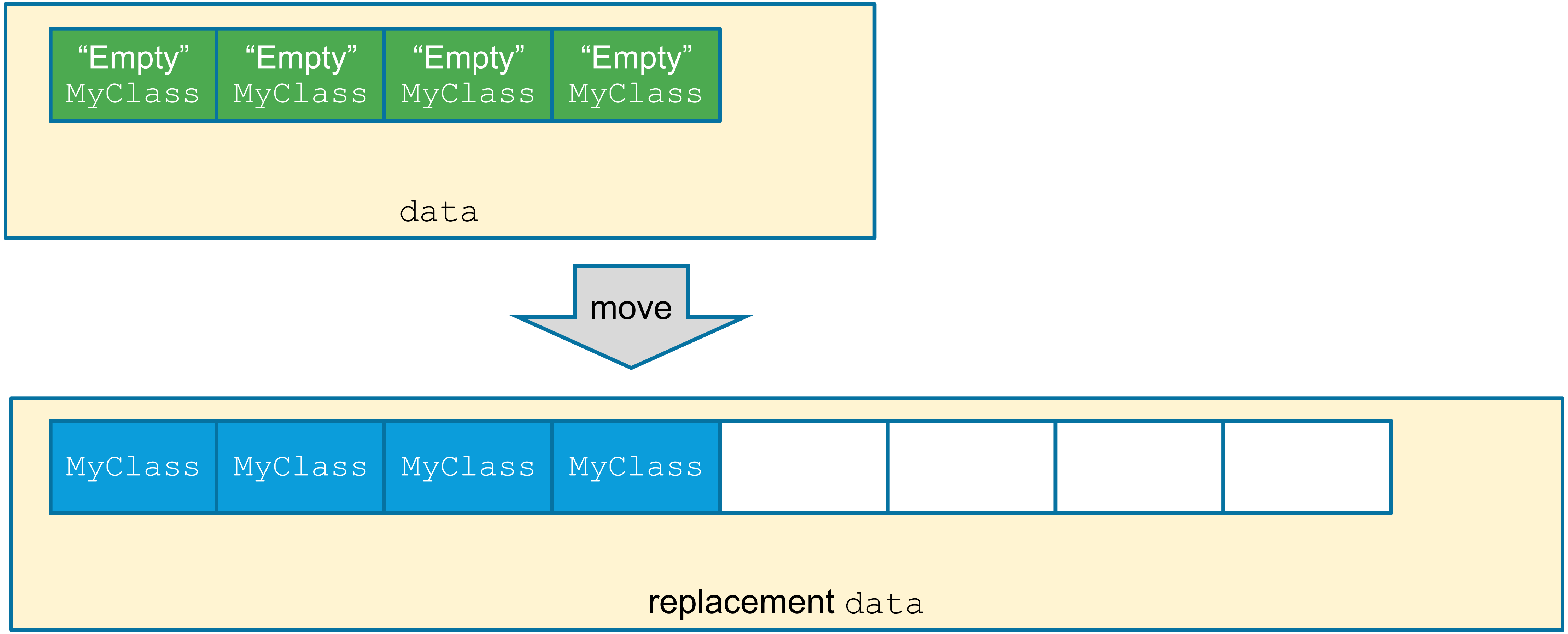


Bloomberg Engineering

Growing a vector the C++11 way



Growing a vector the C++11 way



“Empty” `MyClass` | “Empty” `MyClass` | “Empty” `MyClass` | “Empty” `MyClass`

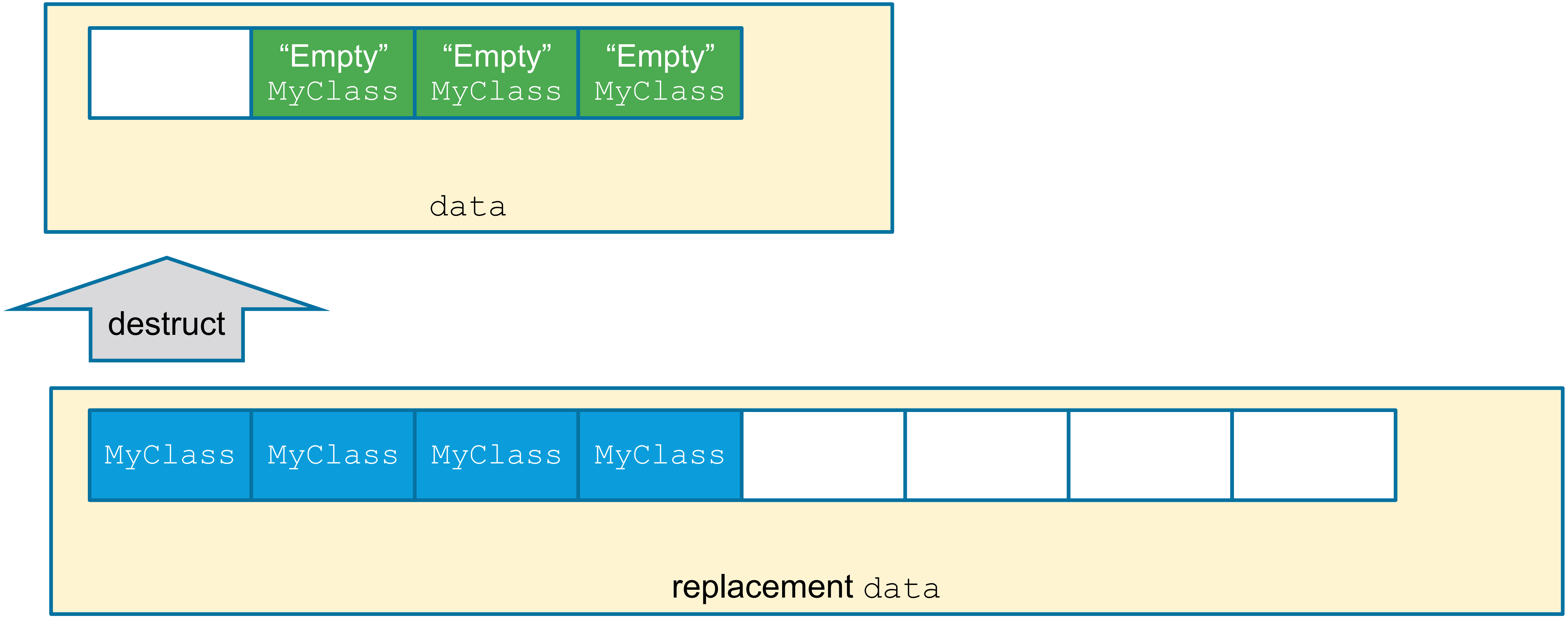
`data`

move

`MyClass` | `MyClass` | `MyClass` | `MyClass`

replacement `data`

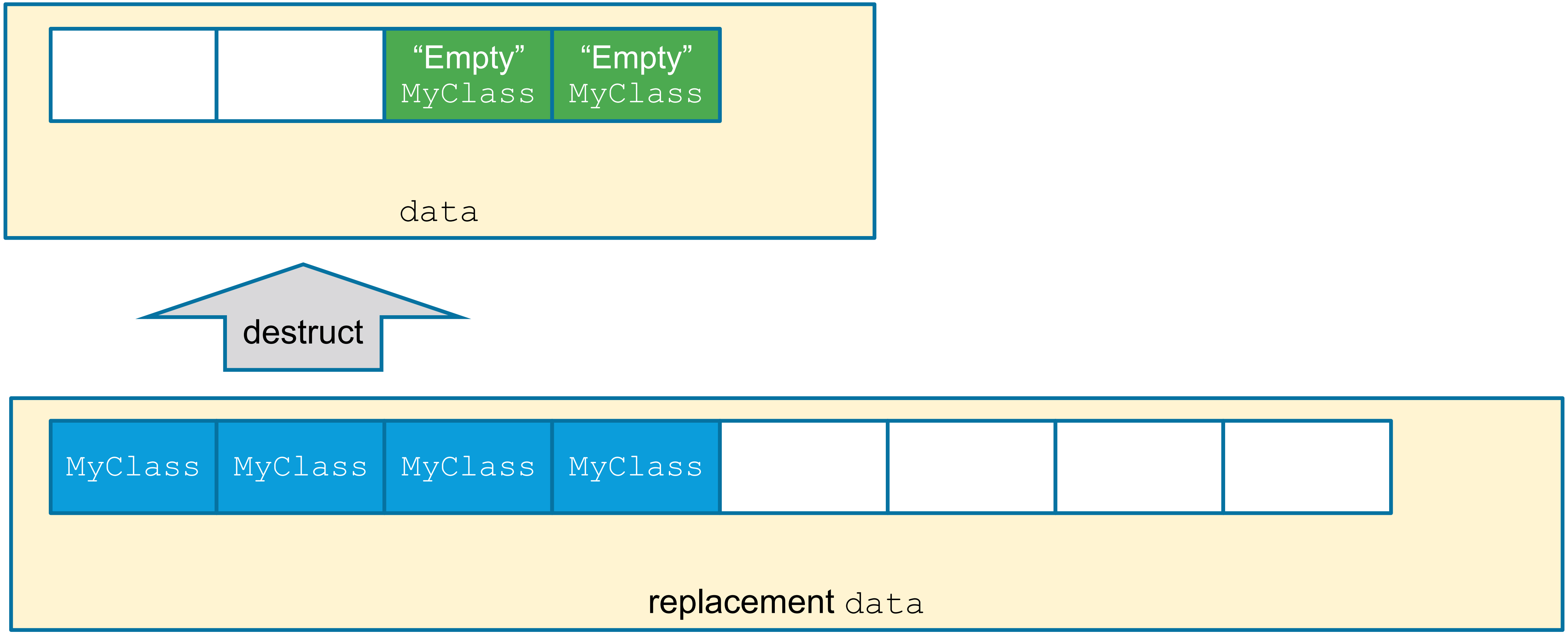
Growing a vector the C++11 way



Bloomberg

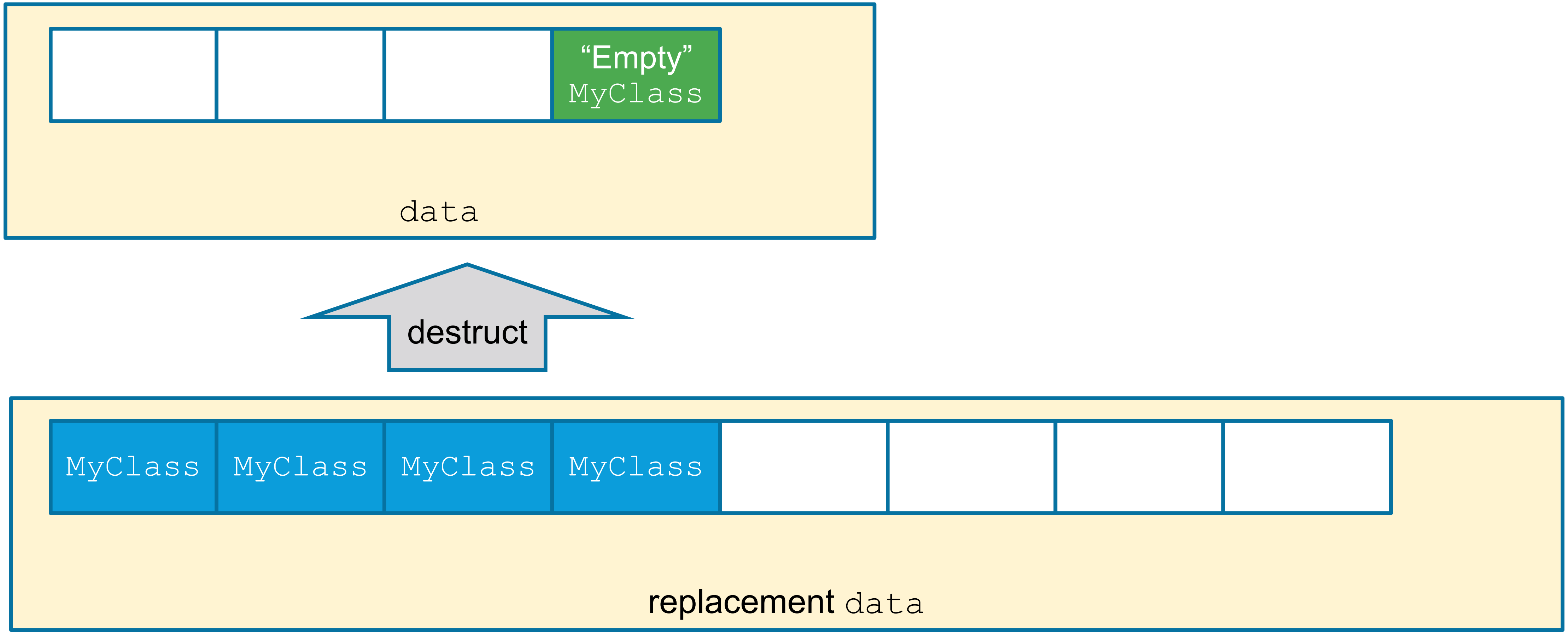
Engineering

Growing a vector the C++11 way

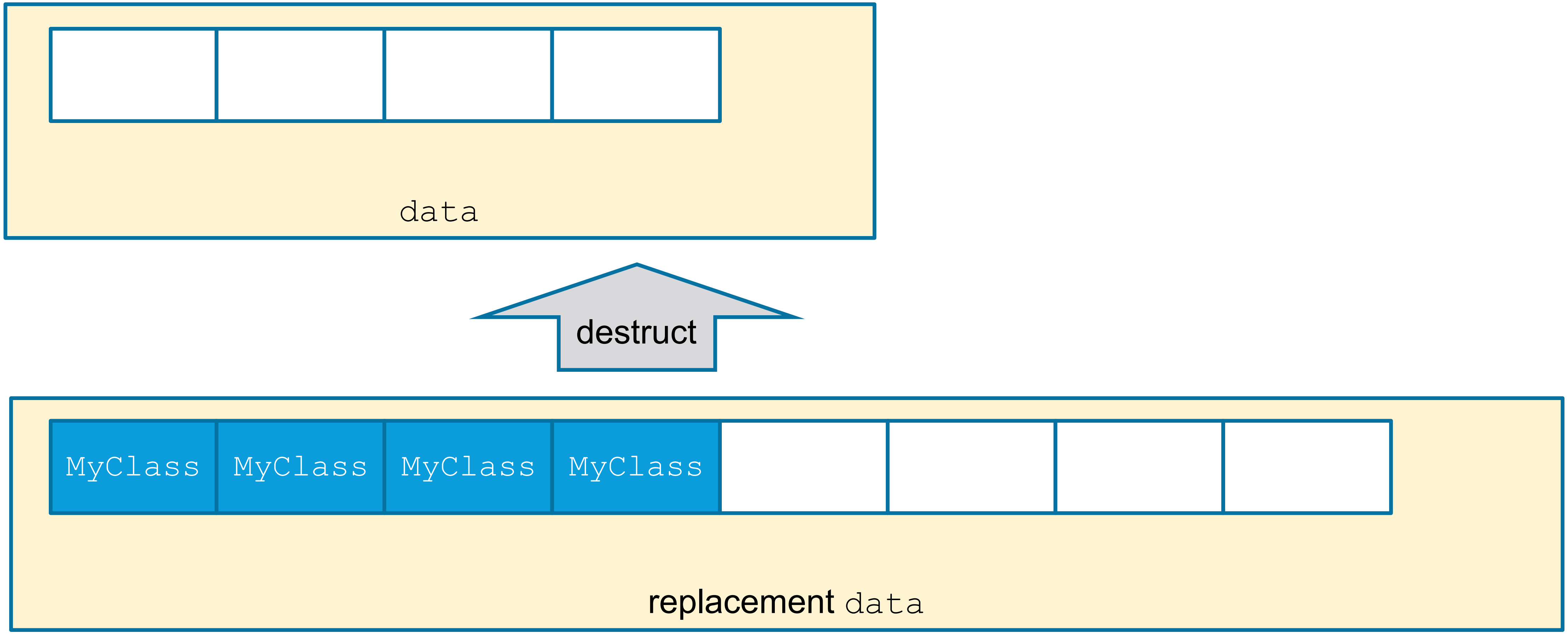


Bloomberg Engineering

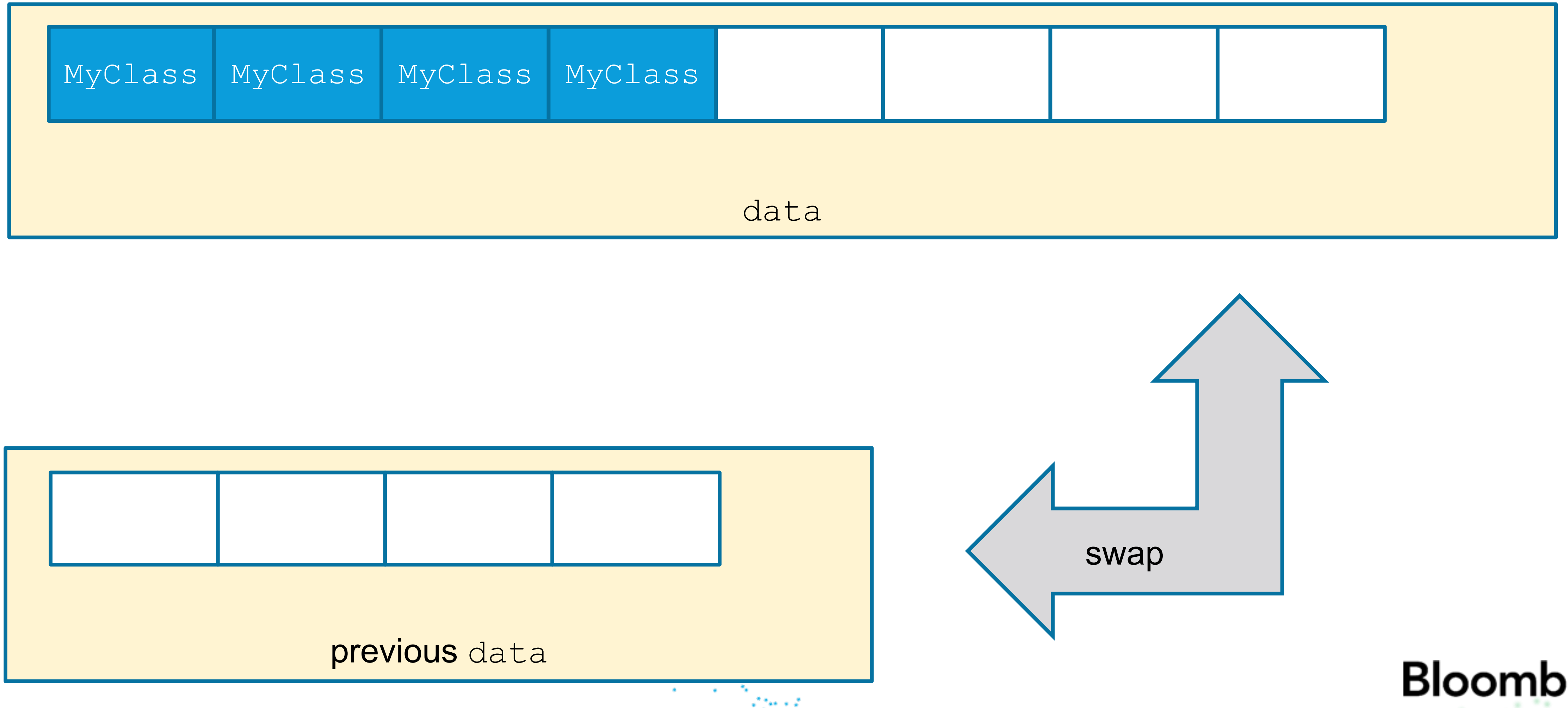
Growing a vector the C++11 way



Growing a vector the C++11 way

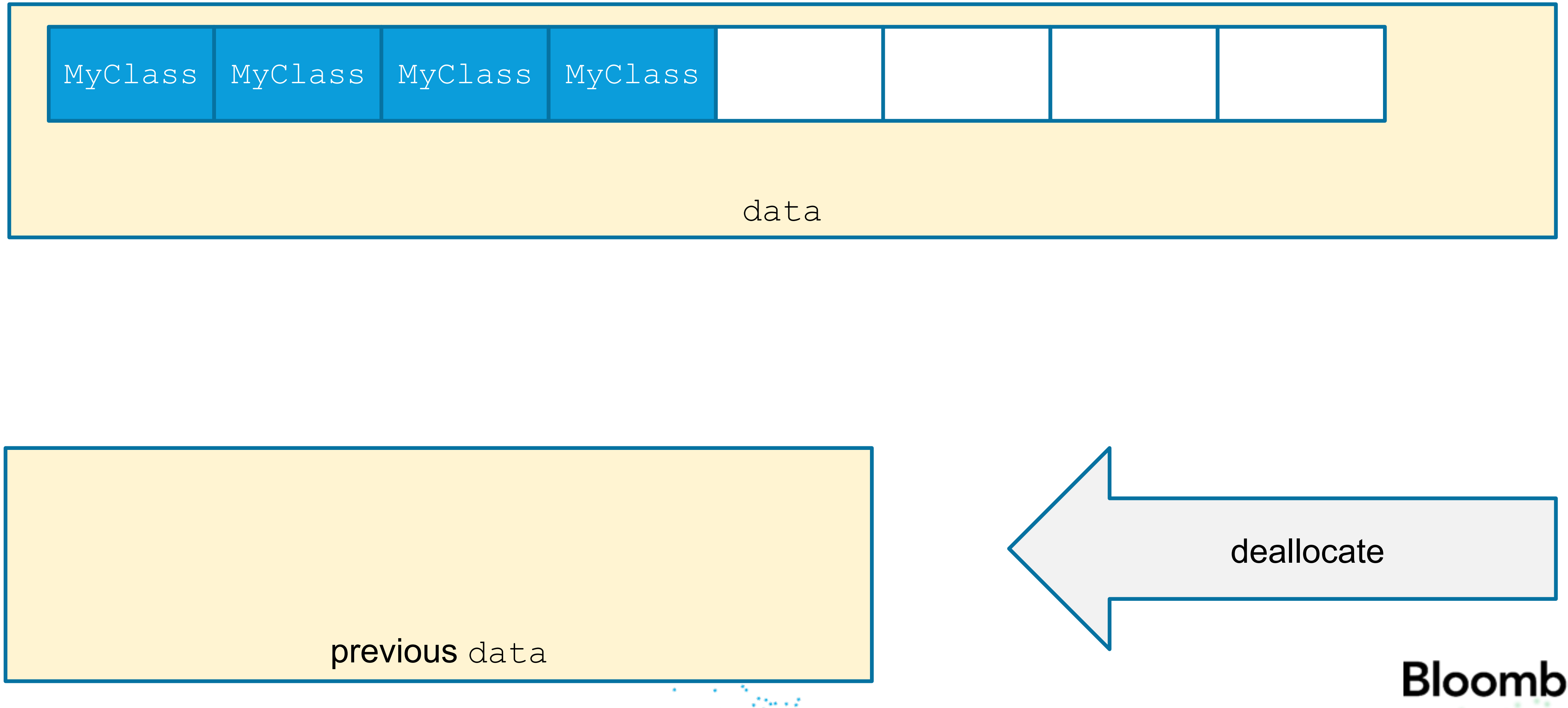


Growing a vector the C++11 way



Bloomberg Engineering

Growing a vector the C++11 way



Growing a vector the C++11 way

- For every entry in an array, we would have to call the move constructor on the destination and the destructor on the source.

- Let us consider `vector<unique_ptr>`.

Constructor (once per element)	Destructor (once per element)

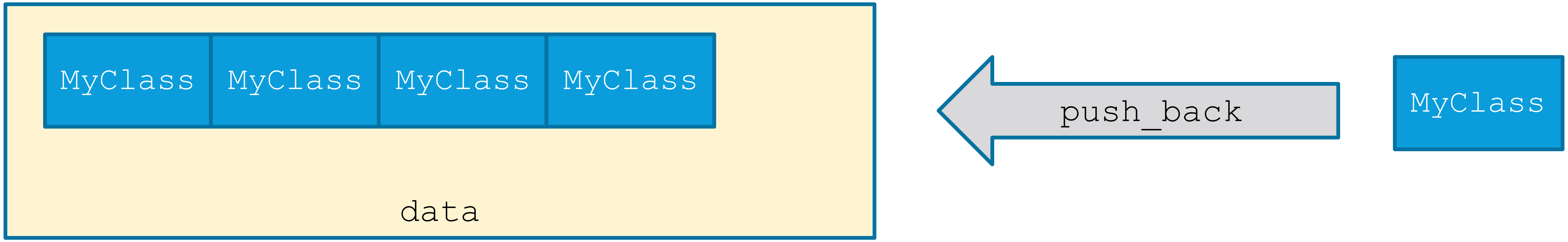
`unique_ptr::unique_ptr(`
`        unique_ptr&& other)`
`{`
`    pointer = other.pointer;`
`    deleter = other.deleter;`
`    other.pointer = nullptr;`
`}`
	`unique_ptr::~unique_ptr()`
	`{`
	`    // In our case, pointer is`
	`    // always null.`
	`    if (pointer)`
	`        deleter(pointer);`
	`}`

Growing a vector with byte copies

- How can we make this even faster?

- Would it be faster if we were allowed to just copy the bytes?

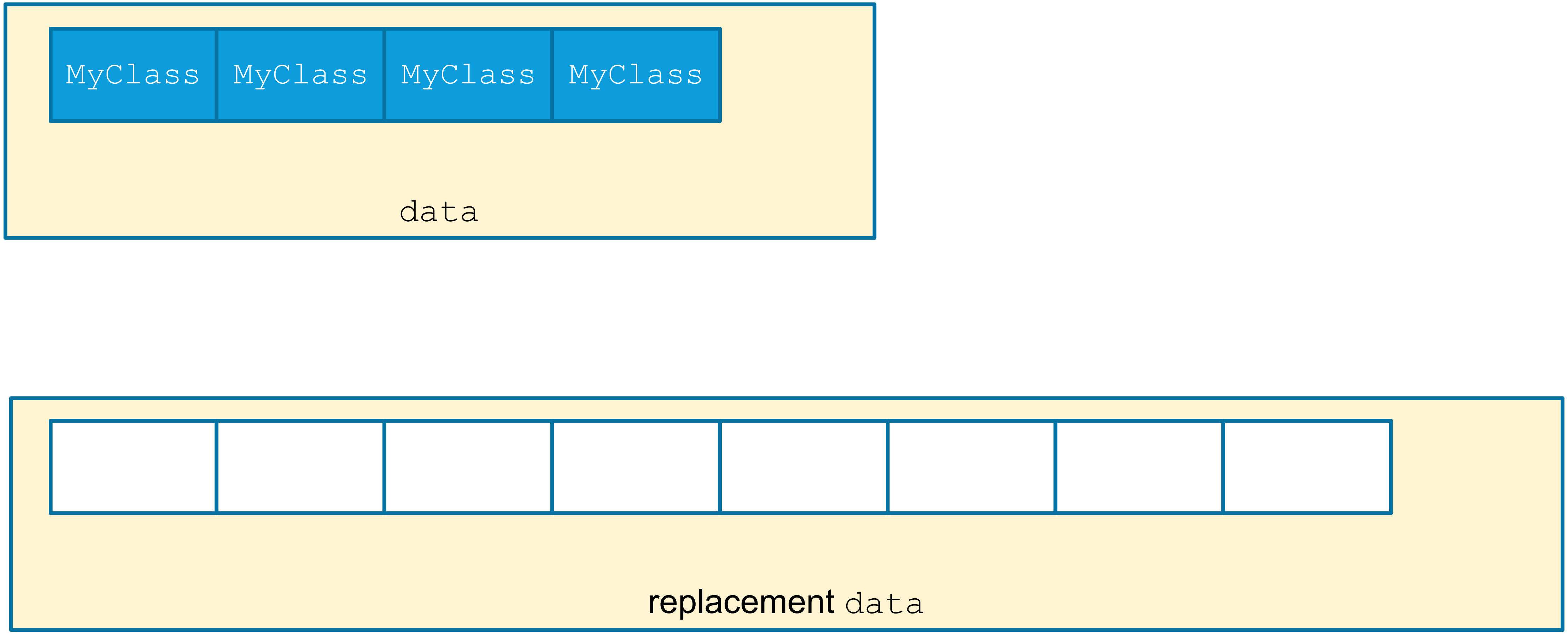
Growing a vector with byte copies



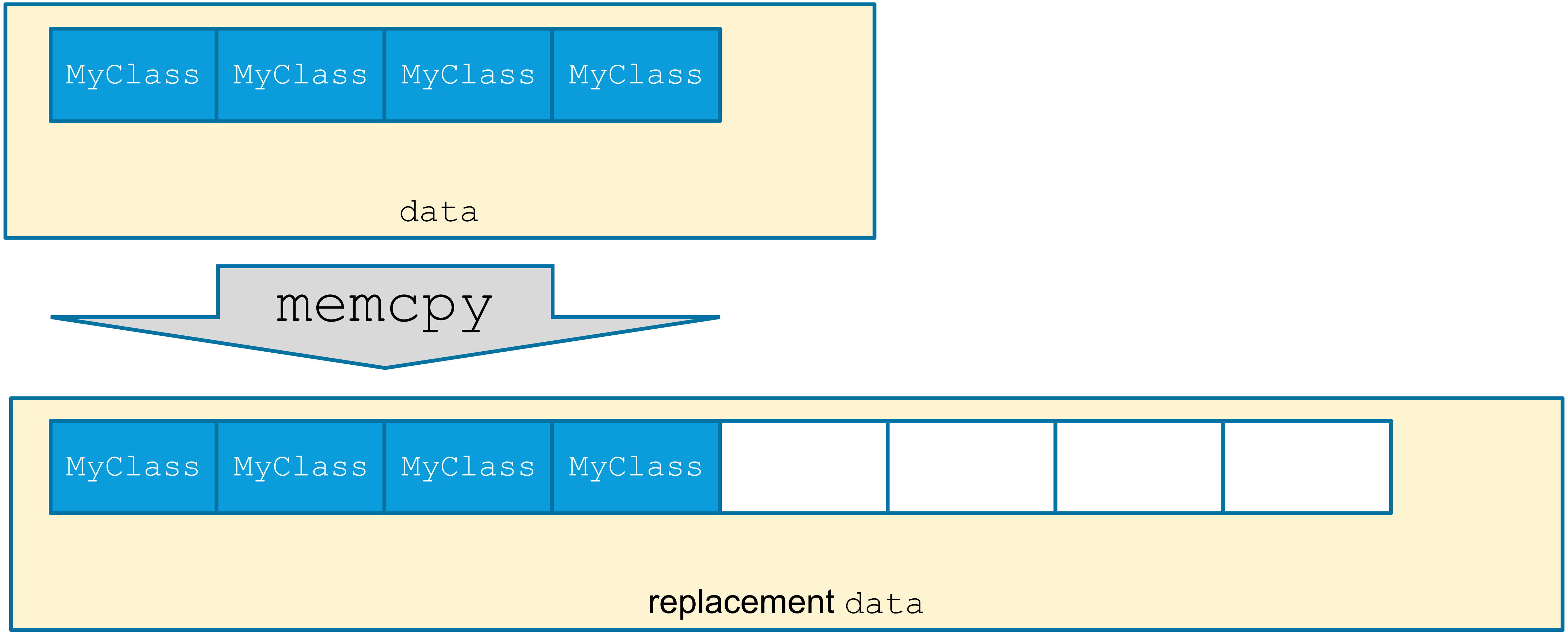
`data` is not big enough!

It needs to grow!

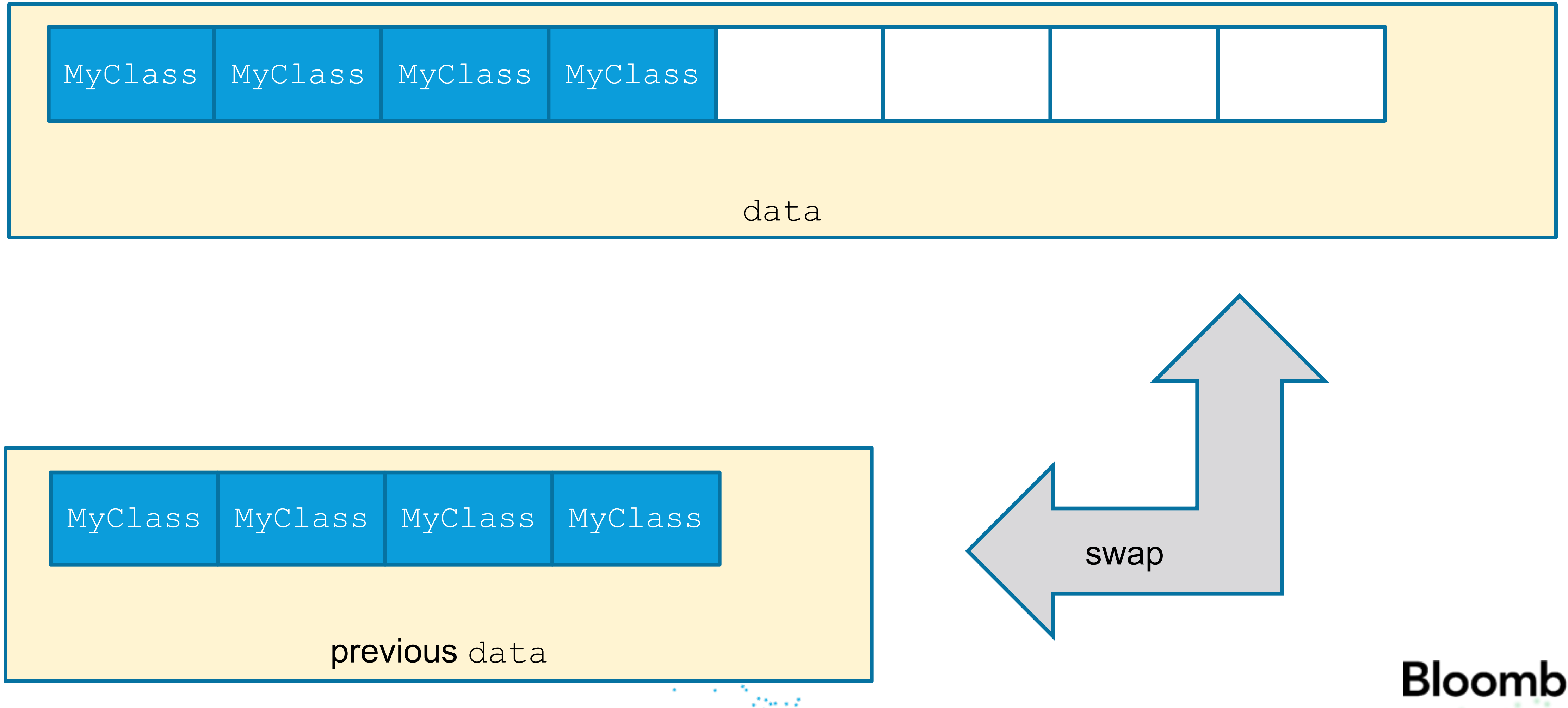
Growing a vector with byte copies



Growing a vector with byte copies



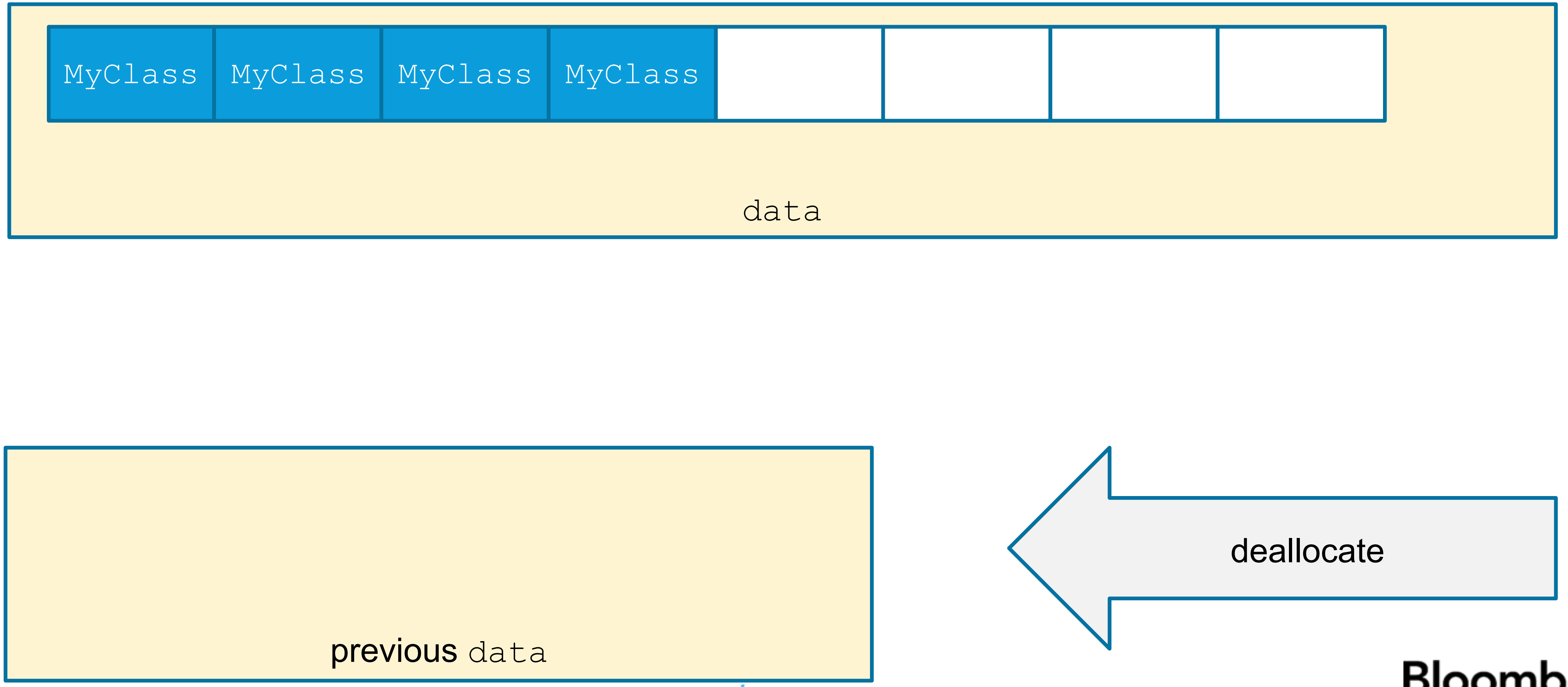
Growing a vector with byte copies



Bloomberg

Engineering

Growing a vector with byte copies



Bloomberg

Engineering

Growing a vector with byte copies

- Consider growing a 4-element `vector<unique_ptr>`.

Using move construction	Using byte copying

tmp=::operator new(8*sizeof(unique_ptr));
tmp[0].pointer = src[0].pointer;
tmp[0].deleter = src[0].deleter;
src[0].pointer = 0;
tmp[1].pointer = src[1].pointer;
tmp[1].deleter = src[1].deleter;
src[1].pointer = 0;
tmp[2].pointer = src[2].pointer;
tmp[2].deleter = src[2].deleter;
src[2].pointer = 0;
tmp[3].pointer = src[3].pointer;
tmp[3].deleter = src[3].deleter;
src[3].pointer = 0;
if(src[0].pointer) …
if(src[1].pointer) …
if(src[2].pointer) …
if(src[3].pointer) …
::operator delete(src);
src=tmp
	tmp=
	 ::operator new(8*sizeof(unique_ptr));
	memcpy(tmp,
	       src,
	       4*sizeof(unique_ptr));
	::operator delete(src);
	src=tmp

Growing a vector with byte copies

- We can compare the optimised assembly to grow `vector<unique_ptr>`.

Using move construction

```
movsxd  r15, esi
lea     rdi, [8*r15]
call    operator new(unsigned long)@PLT
mov     rbx, rax
test    r15d, r15d
jle     .LBB0_1
mov     r15d, ebp
cmp     ebp, 4
jae     .LBB0_6
xor     eax, eax
jmp     .LBB0_5
.LBB0_1:
mov     rdi, r14
call    operator delete(void*)@PLT
jmp     .LBB0_2
.LBB0_6:
mov     eax, r15d
and     eax, -4
lea     rcx, [8*r15]
and     rcx, -32
xor     edx, edx
xorps   xmm0, xmm0
.LBB0_7: # =>This Inner Loop Header:
             Depth=1
movups  xmm1, xmmword ptr [r14 + rdx]
movups  xmm2, xmmword ptr [r14 + rdx +
16]
movups  xmmword ptr [rbx + rdx], xmm1
movups  xmmword ptr [rbx + rdx + 16],
xmm2
movups  xmmword ptr [r14 + rdx], xmm0
movups  xmmword ptr [r14 + rdx + 16],
xmm0
add     rdx, 32
cmp     rcx, rdx
jne     .LBB0_7
cmp     rax, r15
je      .LBB0_9
```

```
.LBB0_5: # =>This Inner Loop Header:
             Depth=1
mov     rcx, qword ptr [r14 + 8*rax]
mov     qword ptr [rbx + 8*rax], rcx
mov     qword ptr [r14 + 8*rax], 0
inc     rax
cmp     r15, rax
jne     .LBB0_5
.LBB0_9:
mov     rdi, r14
call    operator delete(void*)@PLT
test    ebp, ebp
jle     .LBB0_2
xor     r14d, r14d
jmp     .LBB0_11
.LBB0_13: #   in Loop: Header=BB0_11
              Depth=1
inc     r14
cmp     r15, r14
je      .LBB0_2
.LBB0_11: # =>This Inner Loop Header:
              Depth=1
mov     rdi, qword ptr [rbx + 8*r14]
mov     qword ptr [rbx + 8*r14], 0
test    rdi, rdi
je      .LBB0_13
call    operator delete(void*)@PLT
jmp     .LBB0_13
```

Using byte copying

```
movsxd  r14, esi
shl     r14, 3
mov     rdi, r14
call    operator new(unsigned long)@PLT
mov     r15, rax
mov     rdi, rax
mov     rsi, rbx
mov     rdx, r14
call    memcpy@PLT
mov     rdi, rbx
call    operator delete(void*)@PLT
```

Growing a vector with byte copies

- We can compare the optimised assembly to grow `vector<string>`.

Using move construction

```
movsxd  r15, esi
lea     rax, [8*r15]
lea     rdi, [rax + 2*rax]
call    operator new(unsigned long)@PLT
mov     rbx, rax
test    r15d, r15d
jle     .LBB0_1
mov     r15d, ebp
cmp     ebp, 1
jne     .LBB0_12
xor     eax, eax
jmp     .LBB0_5
.LBB0_1:
mov     rdi, r14
call    operator delete(void*)@PLT
jmp     .LBB0_2
.LBB0_12:
mov     ecx, r15d
and     ecx, -2
xor     edx, edx
xorps   xmm0, xmm0
xor     eax, eax
.LBB0_13: # =>This Inner Loop Header:
               Depth=1
mov     rsi, qword ptr [r14 + rdx + 16]
mov     qword ptr [rbx + rdx + 16], rsi
movups  xmm1, xmmword ptr [r14 + rdx]
movups  xmmword ptr [rbx + rdx], xmm1
movups  xmmword ptr [r14 + rdx], xmm0
mov     qword ptr [r14 + rdx + 16], 0
mov     rsi, qword ptr [r14 + rdx + 40]
mov     qword ptr [rbx + rdx + 40], rsi
movups  xmm1, xmmword ptr [r14 + rdx + 24]
movups  xmmword ptr [rbx + rdx + 24], xmm1
movups  xmmword ptr [r14 + rdx + 24], xmm0
mov     qword ptr [r14 + rdx + 40], 0
add     rax, 2
add     rdx, 48
cmp     rcx, rax
jne     .LBB0_13
```

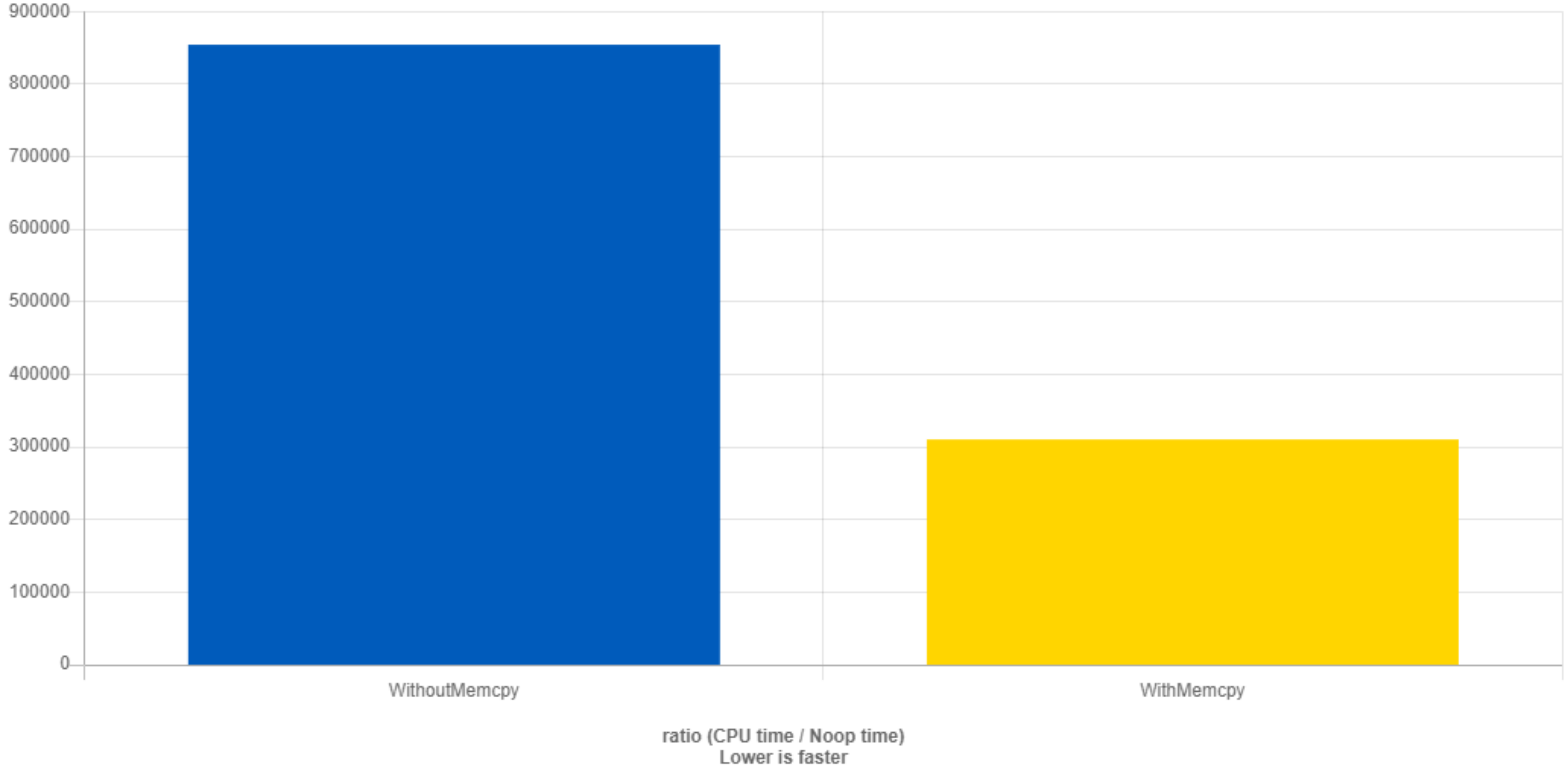
```
.LBB0_5:
test    r15b, 1
je      .LBB0_7
shl     rax, 3
lea     rax, [rax + 2*rax]
mov     rcx, qword ptr [r14 + rax + 16]
mov     qword ptr [rbx + rax + 16], rcx
movups  xmm0, xmmword ptr [r14 + rax]
movups  xmmword ptr [rbx + rax], xmm0
xorps   xmm0, xmm0
movups  xmmword ptr [r14 + rax], xmm0
mov     qword ptr [r14 + rax + 16], 0
.LBB0_7:
mov     rdi, r14
call    operator delete(void*)@PLT
test    ebp, ebp
jle     .LBB0_2
shl     r15, 3
lea     r14, [r15 + 2*r15]
xor     r15d, r15d
jmp     .LBB0_9
.LBB0_11: #   in Loop: Header=BB0_9
                       Depth=1
add     r15, 24
cmp     r14, r15
je      .LBB0_2
.LBB0_9: # =>This Inner Loop Header:
               Depth=1
test    byte ptr [rbx + r15], 1
je      .LBB0_11
mov     rdi, qword ptr [rbx + r15 + 16]
call    operator delete(void*)@PLT
jmp     .LBB0_11
```

Using byte copying

```
movsxd  rax, esi
shl     rax, 3
lea     r14, [rax + 2*rax]
mov     rdi, r14
call    operator new(unsigned long)@PLT
mov     r15, rax
mov     rdi, rax
mov     rsi, rbx
mov     rdx, r14
call    memcpy@PLT
mov     rdi, rbx
call    operator delete(void*)@PLT
```

Growing a vector with byte copies

- Applying trivial relocation optimisations to `vector<string>` gives a factor-2.8 speed-up versus the C++20 Standard Library version, according to a test on quick-bench.com (with optimisation).



Bloomberg
Engineering

Can we really just copy the bytes?

Introducing ***trivially copyable***

Bloomberg

Engineering

Trivially copyable

The C++ Standard defines the term ***trivially copyable type*** as follows:

6.8.1 General [basic.types.general]

- Arithmetic types ([basic.fundamental]), enumeration types, pointer types, pointer-to-member types ([basic.compound]), `std::nullptr_t`, and cv-qualified versions of these types are collectively called *scalar types.* Scalar types, trivially copyable class types ([class.prop]), arrays of such types, and cv-qualified versions of these types are collectively called ***trivially copyable types***.

11.2 Properties of classes [class.prop]

A ***trivially copyable class*** is a class:

- (1.1) that has at least one eligible copy constructor, move constructor, copy assignment operator, or move assignment operator ([special], [class.copy.ctor], [class.copy.assign]),
- (1.2) where each eligible copy constructor, move constructor, copy assignment operator, and move assignment operator is trivial, and
- (1.3) that has a trivial, non-deleted destructor ([class.dtor]).

[*Note 1*: In particular, a trivially copyable or trivial class does not have virtual functions or virtual base classes. — *end note*]

Trivially copyable

The C++ Standard defines the term ***trivial*** for such functions as follows (slide 1/2):

11.4.5.2 Default constructors [class.default.ctor]

- A default constructor is *trivial* if it is not user-provided and if:
- (3.1) its class has no virtual functions ([class.virtual]) and no virtual base classes ([class.mi]), and
- (3.2) no non-static data member of its class has a default member initializer ([class.mem]), and
- (3.3) all the direct base classes of its class have trivial default constructors, and
- (3.4) for all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor.

11.4.5.3 Copy/move constructors [class.copy.ctor]

- A copy/move constructor for class `X` is trivial if it is not user-provided and if:
- (11.1) class `X` has no virtual functions ([class.virtual]) and no virtual base classes ([class.mi]), and
- (11.2) the constructor selected to copy/move each direct base class subobject is trivial, and
- (11.3) for each non-static data member of `X` that is of class type (or array thereof), the constructor selected to copy/move that member is trivial;

Trivially copyable

The C++ Standard defines the term ***trivial*** for such functions as follows (slide 2/2):

11.4.6 Copy/move assignment operator [class.copy.assign]

- A copy/move assignment operator for class `X` is trivial if it is not user-provided and if:
- (9.1) class `X` has no virtual functions ([class.virtual]) and no virtual base classes ([class.mi]), and
- (9.2) the assignment operator selected to copy/move each direct base class subobject is trivial, and
- (9.3) for each non-static data member of `X` that is of class type (or array thereof), the assignment operator selected to copy/move that member is trivial;

11.4.7 Destructors [class.dtor]

- A destructor is trivial if it is not user-provided and if:
- (8.1) the destructor is not virtual,
- (8.2) all of the direct base classes of its class have trivial destructors, and
- (8.3) for all of the non-static data members of its class that are of class type (or array thereof), each such class has a trivial destructor.

Trivially copyable

In high-level terms, a good way to think about this is that, if you have any of the following, then your class is unlikely to be *trivially copyable:*

- Your own constructor(s)
- Your own destructor
- Your own assignment operator(s)
- Any `virtual` function(s) or base class(es)
- Any members or bases that are not *trivially copyable*

So, generally speaking, only the most simple types tend to be trivially copyable.

Trivially copyable

Example of a type that is *trivially copyable* and, therefore, a `vector` would use an optimised implementation:

```
struct MyClass {
        int data1;
        int data2;
        double calculate();
    };
```

Trivially copyable

Examples of types that are not *trivially copyable*:

```
std::unique_ptr
std::shared_ptr
std::string
std::pair<int, int>
```

Bloomberg

Engineering

Trivially copyable

Most current implementations of vector will use `memcpy` as an optimization for *trivially copyable* types.

What do current libraries do?

Introducing ***trivially relocatable***

Facebook Folly (open source)

Bloomberg BDE (open source)

Others (such as Qt)

Common themes

Bloomberg

Engineering

Trivially relocatable

- The term *trivially relocatable* is not defined in the Standard.

- For the purposes of this presentation, we will use the term *trivially relocatable* to describe a type that we can relocate using `memcpy` (given the proviso that we do not subsequently call the destructor on the relocated-from object).

  - A good mental model is to consider, after a relocation operation, that the source object is no more. It has ceased to be. Bereft of life, it rests in peace. It is an ex-object.

Trivially relocatable

Trivial relocation	Relocation using move constructor

// allocate destination memory
dest =
  ::operator new(sizeof(Type));
 	
// copy bytes
memcpy(dest, source,
       sizeof(Type));
 	
// deallocate source
::operator delete(source);
	// allocate destination memory
	dest =
	  ::operator new(sizeof(Type));
	
	// move construct
	::new(dest)
	      Type(std::move(*source));
	
	// destruct source
	source->~Type();
	
	// deallocate source
	::operator delete(source);

Facebook Folly (open source)

- Folly’s `fbvector` class supports `memcpy` for relocations.
- If your type can be relocated using `memcpy`, you need to indicate this fact by partially specialising `IsRelocatable<>`.

```
// at global namespace level
namespace folly {
    struct IsRelocatable<Widget> : boost::true_type {};
}
```

- This must be done after your definition of `Widget` but before you make use of `fbvector<Widget>`.

Bloomberg BDE (open source)

- Bloomberg BDE’s `vector` implementation also supports `memcpy` for relocations.

- If your type can be relocated using `memcpy`, you need to indicate this, which can be done with either a nested trait syntax or a standard trait-like partial specialization.

Bloomberg BDE (open source)

```
class Widget {
    // ...
    // TRAITS
    BSLMF_NESTED_TRAIT_DECLARATION(Widget,
                                   BloombergLP::bslmf::IsBitwiseMoveable);
        // 'Widget' is trivially relocatable.
    // ...
};
```

```
// TYPE TRAITS
namespace bslmf {

template <>
struct IsBitwiseMoveable<Widget> : bsl::true_type
{
    // 'Widget' is trivially relocatable.
};

}  // close namespace bslmf
```

Others

- Many other libraries, lacking language support, adopt similar approaches.

- In Qt, for example, the syntax uses a macro.

```
Q_DECLARE_TYPEINFO( Widget, Q_RELOCATABLE_TYPE );
```

Common themes

- Every single non-*trivially copyable* type that we wish to optimise using `memcpy` must be individually flagged.
- Flagging Standard Library types results in code portability issues (e.g., std::string in libc++ vs. libstdc++).
- The elephant in the room: Both libraries rely on compilers allowing what is, technically, undefined behaviour.

Note: `std::string` can be trivially relocated in libc++ but not in libstdc++,
which uses self-references in its short string optimization.

Relying on undefined behaviour

The caveat with using `memcpy`:

> If the type is not an ***implicit-lifetime*** type, then it is, technically, undefined behaviour to access any non-static members or call any non-static functions on the copied object.
> (C++ Standard, section [basic.life])

The good news:

> No current compilers track this, so libraries can “get away with it”, but there is no guarantee that a future compiler will not decide to optimise away that access and break our code.

Note: All *trivially copyable* types are, by definition, *implicit-lifetime* types.

Is trivial relocation worth doing?

- The vast majority of types are not *trivially copyable*, and those that are tend to be very small and very simple.

- The vast majority of types can be trivially relocated.

- The only non-trivially relocatable types tend to be complex structures that store (directly or indirectly) pointers to themselves or to their own members.

- Thus, adding trivial relocatability to the language would allow std::vector to use `memcpy` in almost all cases.

Is trivial relocation worth doing?

All of the following Standard Library types, though not *trivially copyable,* may be, depending on the library implementation, trivially relocatable:

```
std::unique_ptr
std::shared_ptr
std::string
std::pair<int, int>
```

Note: `std::string` can be trivially relocated in libc++ but not in libstdc++, which uses self-references in its short string optimization.

Adding trivial relocation to the C++ Standard

First attempt: 2014, N4034, Pablo Halpern

Bloomberg

Engineering

First attempt, 2014, Pablo Halpern

N4158: Destructive Move

https://wg21.link/n4158

First attempt, 2014, Pablo Halpern

- This originally started out as paper N4034 https://wg21.link/n4034.

- It was based on (or at least inspired by) the BDE library approach.

- Note that this proposal also considers the case of non-trivial relocations, but that is out of scope for this presentation.

First attempt, 2014, Pablo Halpern

- New Standard Library type traits were proposed.

```
is_trivially_destructive_movable
is_nothrow_destructive_movable
```

- `is_trivially_destructive_movable` defaults to true for types that are both
  - *trivially move constructible.*
  - *trivially destructible.*

- `is_nothrow_destructive_movable` defaults to true if calling `uninitialized_destructive_move` on a type is `noexcept`.

First attempt, 2014, Pablo Halpern

- A new low-level Standard Library function was proposed.

```
template<class T>
uninitialized_trivial_destructive_move(T* from, T* to);
```

- This function is equivalent to
  - — running `memcpy(to, from, sizeof(T))`.
  - — starting the lifetime of `*to`.
  - — ending the lifetime of `*from`.

- This function requires the trait `is_trivially_destructive_movable<T>` to be true.

First attempt, 2014, Pablo Halpern

- New Standard Library functions were proposed.

```
uninitialized_destructive_move
uninitialized_destructive_move_n
```

- These functions default to calling the move constructor and destructor if `is_trivially_destructive_movable` is false, otherwise they call `uninitialized_trivial_destructive_move`.

- Standard Library container implementations can profit by using these methods.

First attempt, 2014, Pablo Halpern

- So what happens if a type, say, `Widget`, can be relocated using `memcpy`?

- You would specialise the `is_trivially_destructive_movable` trait as follows:

```
template <> struct
is_trivially_destructive_movable<Widget> : std::true_type
```

- As a result of this, the function `uninitialized_destructive_move` uses `uninitialized_trivial_destructive_move` rather than construction and destruction
(as does `uninitialized_destructive_move_n`).

First attempt, 2014, Pablo Halpern

- This paper did not progress as it would have required a core language proposal to change the lifetime model and allow something other than a constructor to start the lifetime of an object.

- The WG21 discussion of the lifetime issues raised by this paper did inspire another subsequent paper N4393, “Noop Constructors and Destructors” https://wg21.link/n4393.

- N4393 proposed special constructor and destructor syntax to begin and end the lifetime of an object.

Adding trivial relocation to the C++ Standard

First attempt:	2014, N4034, Pablo Halpern
Second attempt:	2016, P0023, Denis Bider

Second attempt, 2016, Denis Bider

P0023: Relocator: Efficiently moving objects

https://wg21.link/p0023

Bloomberg

Engineering

Second attempt, 2016, Denis Bider

- A relocation constructor, somewhat akin to move constructors, was proposed.

```
class A {
   >>A(A&);    // relocator
};
```

Second attempt, 2016, Denis Bider

- This was the very first proposal to include rules whereby the compiler can deduce a type’s trivial relocatability.

> If the definition of a class X does not explicitly declare a relocator, a non-explicit one is implicitly declared as defaulted, if and only if class X satisfies the following criterion for each other special member:
>
> - X does not have a user-declared (*special member*), or the user-declared (*special member*) is defaulted at first declaration.

Second attempt, 2016, Denis Bider

- Two new type traits were proposed.

```
template struct is_relocatable;
template struct is_trivially_relocatable;
```

- These were defined as follows:

> The value of `is_relocatable::value` is true if T has either a user-defined relocator, or a defaulted relocator that is not defined as deleted.
>
> The value of `is_trivially_relocatable::value` is true if T has a trivial relocator. A trivial relocator is one that is defaulted, not deleted, and calls only other trivial relocators. It is equivalent to `memcpy`.

Second attempt, 2016, Denis Bider

- So what happens if a type, say, `Widget`, can be relocated using `memcpy`?

- You would default the relocator using the following syntax:

```
class A {
   >>A(A&) = default;    // relocator
};
```

- Library functions can then, if they wish, test this using `is_trivially_relocatable` and optimise accordingly.

Second attempt, 2016, Denis Bider

- Note that this proposal also looks at the case of non-trivial relocations, but that is out of scope for this presentation.

- For unrelated reasons, this proposal did not progress beyond the initial (revision 0) version.

Bloomberg

Engineering

Adding trivial relocation to the C++ Standard

First attempt:	2014, N4034, Pablo Halpern
Second attempt:	2016, P0023, Denis Bider
Third attempt:	2020, P1029, Niall Douglas

Third attempt, 2020, Niall Douglas

P1029: move = bitcopies

https://wg21.link/p1029

Third attempt, 2020, Niall Douglas

- This proposal was partly motivated by a desire to optimise lightweight exceptions.

- For more details, see the paper “Zero-overhead deterministic exceptions: Throwing values” by Herb Sutter https://wg21.link/p0709

Bloomberg

Engineering

Third attempt, 2020, Niall Douglas

- This proposal suggests a mechanism to specify that the move constructor can be performed by means of a `memcpy`.

```
class A {
    A(A &&) = bitcopies;
};
```

- This causes the compiler to perform all move constructions using as-if `memcpy` (i.e., the compiler is permitted to elide the copy if it is able to do so).

Third attempt, 2020, Niall Douglas

A type trait was proposed

```
template is_move_constructor_bitcopying;
```

> If a type `T`’s move constructor has `= bitcopies` compatible semantics (which includes trivial copyability), the trait `std::is_move_constructor_bitcopying<T>` shall be true.

which enables libraries to optimise based on trivial relocatability.

Third attempt, 2020, Niall Douglas

- This proposal also includes a mechanism to delegate the decision-making to the compiler.

```
class A {
   A(A &&) = bitcopies(auto);
};
```

Third attempt, 2020, Niall Douglas

- An `= bitcopies` move requires two `memcpy` operations (although the compiler may choose to elide one or both of these).
- Such a move is defined to be equivalent to the following:

```
// Copy bytes of src to dest
memcpy(dest, src, sizeof(Type));

// Copy bytes of constexpr default constructed
// instance to src
static constexpr Type default_constructed{};
memcpy(src, &default_constructed, sizeof(Type));
```

Third attempt, 2020, Niall Douglas

- There are a number of limitations on using move = bitcopies.
  - All bases and members must be either *trivially copyable* or have an `= bitcopies` move constructor.
  - There must be no virtual inheritance.
  - The type itself, as well as all bases and members, must have a `constexpr` default constructor.

Third attempt, 2020, Niall Douglas

- So, not all trivially relocatable types can be given an `= bitcopies` move constructor!
  - This excludes, for example, `std::list,` which is permitted to allocate on construction.
  - This also excludes, for example, anything that writes debug output to a log file on construction.

Bloomberg

Engineering

Third attempt, 2020, Niall Douglas

- For various unrelated reasons, this proposal didn’t progress beyond the initial paper.

Bloomberg
Engineering

Adding trivial relocation to the C++ Standard

First attempt:	2014, N4034, Pablo Halpern
Second attempt:	2016, P0023, Denis Bider
Third attempt:	2020, P1029, Niall Douglas
Fourth attempt:	2018-present, P1144, Arthur O’Dwyer
Fifth attempt:	2023-present, P2786, Alisdair Meredith & Mungo Gill

Fourth and fifth attempts, 2018-present

P1144: Object relocation in terms of move plus destroy

Arthur O’Dwyer

https://wg21.link/p1144r6

P2786: Trivial relocatability options

Alisdair Meredith & Mungo Gill

https://wg21.link/p2786

Note: As of revision 7, the title of P1144 has been changed to std::is_trivially_relocatable.

Fourth and fifth attempts, 2018-present

- P1144 has been under development since 2018.

- P2786 was first introduced during the WG21 2023 Issaquah meeting.

- Unlike the previous papers, these are still under consideration for possible inclusion in the C++ Standard.

- As both proposals are very similar, we will discuss them together and then talk about the differences.

Fourth and fifth attempts, 2018-present

- Both proposals focus almost entirely on the trivially relocatable case.
  - Trivial relocation is less complicated than non-trivial relocation.
  - Trivial relocation provides the greatest opportunities for optimisation compared to non-trivial relocations.
- Both proposals agree that the object lifetime model will need to be addressed to avoid reliance on technically undefined behaviour, involving changes to the abstract machine.

Bloomberg

Engineering

Fourth and fifth attempts, 2018-present

- Neither proposal requires or relies upon any changes to the existing Standard Library containers and algorithms.

- Both proposals have reference implementations (compiler and Standard Library) either completed or in progress.
  - The P1144 reference implementation is publicly available on https://godbolt.org, e.g., see https://godbolt.org/z/1MzfsPGxd.

Bloomberg

Engineering

Fourth and fifth attempts, 2018-present

- Both proposals agree that trivially copyable types are implicitly trivially relocatable.

- Both proposals agree that, after relocating from an object, the destructor must not be called for that object (now bereft of life); to do so leads to undefined behaviour.

  - Relocating to or from an automatic variable is generally a bad idea, unless you really know what you are doing.

- If a type is explicitly marked as trivially relocatable, but for that type move+destroy is not equivalent to `memcpy`, then greater care is required as resulting behaviour may not be what you intended.

Bloomberg

Engineering

Fourth and fifth attempts, 2018-present

- Both proposals provide type traits to enable library implementors to determine trivial relocatability.

P1144	P2786

template< class T > struct is_relocatable;
template< class T > struct is_nothrow_relocatable;
template< class T > struct is_trivially_relocatable;
	template< class T > struct is_trivially_relocatable;

Fourth and fifth attempts, 2018-present

- Both proposals provide a syntax to flag classes as trivially relocatable.

P1144	P2786
`struct`	`struct C`
`[[trivially_relocatable(true)]]`	`trivially_relocatable(true)`
`C {`	`{`
`    C(C&&);`	`    C(C&&);`
`    ~C();`	`    ~C();`
`};`	`};`
`static_assert(`	`static_assert(`
`is_trivially_relocatable_v<C>);`	`is_trivially_relocatable_v<C>);`

Fourth and fifth attempts, 2018-present

- Both proposals provide relocation functions although, as we will show in a subsequent slide, they behave very differently.

P1144

```cpp
template<class T>
T *relocate_at(T* source, T* dest);

template<class T>
T relocate(
    T* source);
```

P2786

```cpp
template<class T>
requires
(is_trivially_relocatable_v<T> &&
                 !is_const_v<T>)
void trivially_relocate(
    T* begin,
    T* end,
    T* new_location) noexcept;
```

Fourth and fifth attempts, 2018-present

- Both also provide convenience functions.

P1144

```
template<class InputIterator,
         class NoThrowFwdIterator>
NoThrowFwdIterator
uninitialized_relocate(
    InputIterator first,
    InputIterator last,
    NoThrowFwdIterator result);

template<class InputIt, class Size,
         class NoThrowFwdIt>
pair<InputIt, NoThrowFwdIt>
uninitialized_relocate_n(
    InputIt first, Size n,
    NoThrowFwdIt result);
```

P2786

```
template<class T>
requires
((is_trivially_relocatable_v<T> &&
 !is_const_v<T>) ||
is_nothrow_move_constructible_v<T>)
T* relocate(
    T* begin, T* end,
    T* new_location)
```

NOTE: P1144’s `relocate` and P2786’s `relocate` are very different functions. They just happen to have the same name in both proposals.

Fourth and fifth attempts, 2018-present

- Both proposals provide automatic compiler detection of trivially relocatable types.

P1144

A object type T is a ***trivially relocatable*** type if it is:
— a trivially copyable type, or
— an array of trivially relocatable type, or
— a (possibly cv-qualified) class type declared with a [[trivially_relocatable]] attribute with value true, or
— a (possibly cv-qualified) class type which:
  — has no user-provided move constructors or move assignment operators,
  — has no user-provided copy constructors or copy assignment operators,
  — has no user-provided destructors,
  — has no virtual member functions,
  — has no virtual base classes,
  — all of whose members are either of reference type or of trivially relocatable type, and
  — all of whose base classes are trivially relocatable.

Fourth and fifth attempts, 2018-present

- Both proposals provide automatic compiler detection of trivially relocatable types.

P2786 (This definition is currently being revised.)

A trivially relocatable class is a class that:
— has no base classes that are not of trivially relocatable type,
— has no non-static non-reference data members whose type is not a trivially relocatable type,
— has no virtual base classes,
— has no user-provided or deleted destructors,
— either has no trivially_relocatable predicate, or has a trivially_relocatable predicate that evaluates to true,
— and either
    — has a move constructor that is neither user-provided nor deleted, or
    — has no move constructor and has a copy constructor that is neither user provided nor deleted.

Fourth and fifth attempts, 2018-2023

- So, if they have so much in common, why are there two proposals under consideration?

- Why are they different?

- (We will ignore any cosmetic stuff like function names and the keyword vs. attribute question.)

Key differences: A difference in tone

- P1144 is more focused on providing methods for higher-level language users.
  - There are more utility functions.
  - Iterator-based interfaces are provided.
  - There is less implementation detail.
- P2768 is more focused on providing a low-level interface for library implementors.
  - A key focus is on the implications for the *abstract machine.*
  - Only one low-level interface and one optional utility function are provided.

Key differences: Utility functions vs. a low-level interface

- In P1144, the functions `relocate_at` (and `relocate`) will use `memcpy` for trivially relocatable types but will fall back to using move construction and destruction otherwise.

- In P2768, `trivially_relocate` will use `memcpy` for trivially relocatable types and will fail to compile otherwise.

Key differences: Interface vs. semantics

- P1144 supports relocation based on the public interface of a type.
  - It is explicitly stated that relocation is equivalent to move+destroy.
  - A type must have a public constructor and destructor.
- P2768 is instead based on the semantics of a type.
  - Relocation is a primitive operation in the memory/object model.
  - Constructors and destructors are not required to be public.
  - Assignment operators have no bearing on the matter.

Bloomberg

Engineering

Key differences: Examples

- Types considered trivially relocatable under P2768 but not P1144 include
  - polymorphic types.
  - everything in `pmr` (i.e., types following scoped allocator model).
  - `const` objects.
  - (some) types with `const` data members.
- Types considered trivially relocatable under P1144 but not P2786 include
  - objects with data members from third-party libraries where those types are not marked as trivially relocatable.

Key differences: Ill-formed code

- P1144 is more dangerous but gives developers greater freedom.
  - Marking a type trivially relocatable where that doesn't make sense is ill formed but is not required to generate any diagnostics and will result in undefined behaviour.
- P2768 is safer but more restrictive for developers.
  - Marking a type trivially relocatable will result in a compile-time error where any members or base classes are not trivially relocatable or where there is virtual inheritance.

Note: In the interests of openness I should point out that the author of P1144 disagrees with this opinion of the relative safety of these proposals.

Key differences: What about assignment and `std::swap`?

- P1144 includes an assumption that, for a trivially relocatable type
  - `memcpy` can be used in place of move-constructor relocation.
  - `memcpy` can be used in place of assignment-operator relocation.
  - `memcpy` can be used for swapping.
- P2768 assumes only the first of these.
- Note: The author of P1144 subsequently discussed the generic swap question in blog posts at https://quuxplusone.github.io/blog/2023/02/24/trivial-swap-x-prize/.

Key differences: What about assignment and `std::swap`?

Is relocation using the move constructor equivalent to relocation using the assignment operator?

Move constructor	Assignment operator

`// Destruct the destination.`
`destination->~Type();`
`// move construct`
`new(destination)`
`      Type(std::move(*source));`
	`// assign`
	`*destination = *source;`

Key differences: What about assignment and `std::swap`?

Is relocation using the move constructor equivalent to relocation using the assignment operator?

- For some types, this is generally NOT the case, such as
  - types with `const` and/or reference members (such types are not assignable).
  - types with non-propagating allocators, such as `std::pmr::string`, unless it can be guaranteed that the source and target objects have the same allocator.

Key differences: What about assignment and `std::swap`?

Is relocation using the move constructor equivalent to relocation using the assignment operator?

- For `std::pmr::vector`, we can safely assume that all members of that vector have been constructed using the same allocator.

- Therefore, for P2786-style trivially relocatable types,
  - it is perfectly safe to use `memcpy` to move elements around within a `std::pmr::vector` or similarly allocator-aware container.
  - we cannot assume this is safe in any other situation.

Fourth and fifth attempts – Next steps

- The papers' authors will work together to reconcile their papers, where possible.

- Commonalities and irreconcilable differences will be re-presented to the WG21 committee for further guidance.

- This process will not be quick, but we need to be confident we are doing the right thing before changing the language Standard.

Conclusion

- Enabling containers to use `memcpy` would be a valuable optimisation.

- Current libraries have workarounds, but they are not perfect and cannot be perfect without language support.

- Over the last 9 years, three previous proposals and two current, ongoing proposals indicate the need for adding support for trivial relocatability into the language.

- Perhaps some combination of the two current attempts will make it into C++26.

Bloomberg

Engineering

Thank you! Questions?

https://www.TechAtBloomberg.com/cplusplus

We are hiring: bloomberg.com/engineering

Bloomberg Engineering

TechAtBloomberg.com