Designing a Slimmer Vector of Variants

CHRISTOPHER FRETZ

Cppcon
The C++ Conference

2024
September 15 - 20

Designing a Slimmer Vector of Variants

CppCon 2024
September 18, 2024

Chris Fretz
Senior C++ Engineer

TechAtBloomberg.com

Bloomberg Engineering

Overview for this Talk

- This talk details my experience in creating a novel solution to an observed problem with memory usage of `std::vector<std::variant<...>>`

- The talk starts with the motivating use case, considers several candidate designs, refining as we go, and then presents interesting implications of the approach, benchmarks, and lessons learned

- I implemented the data structure mostly for the fun of it, as well as to get more familiar with C++20 features

- The final presented design in this presentation is a simplified version of the actual data structure, but it should illustrate the point

Motivation

- Heterogeneous containers like `std::vector<std::variant<...>>` are an extremely natural way to represent data across many different paradigms

- Naive usage of `std::vector<std::variant<...>>` can result in *comically* bad memory utilization if the types are of disparate sizes

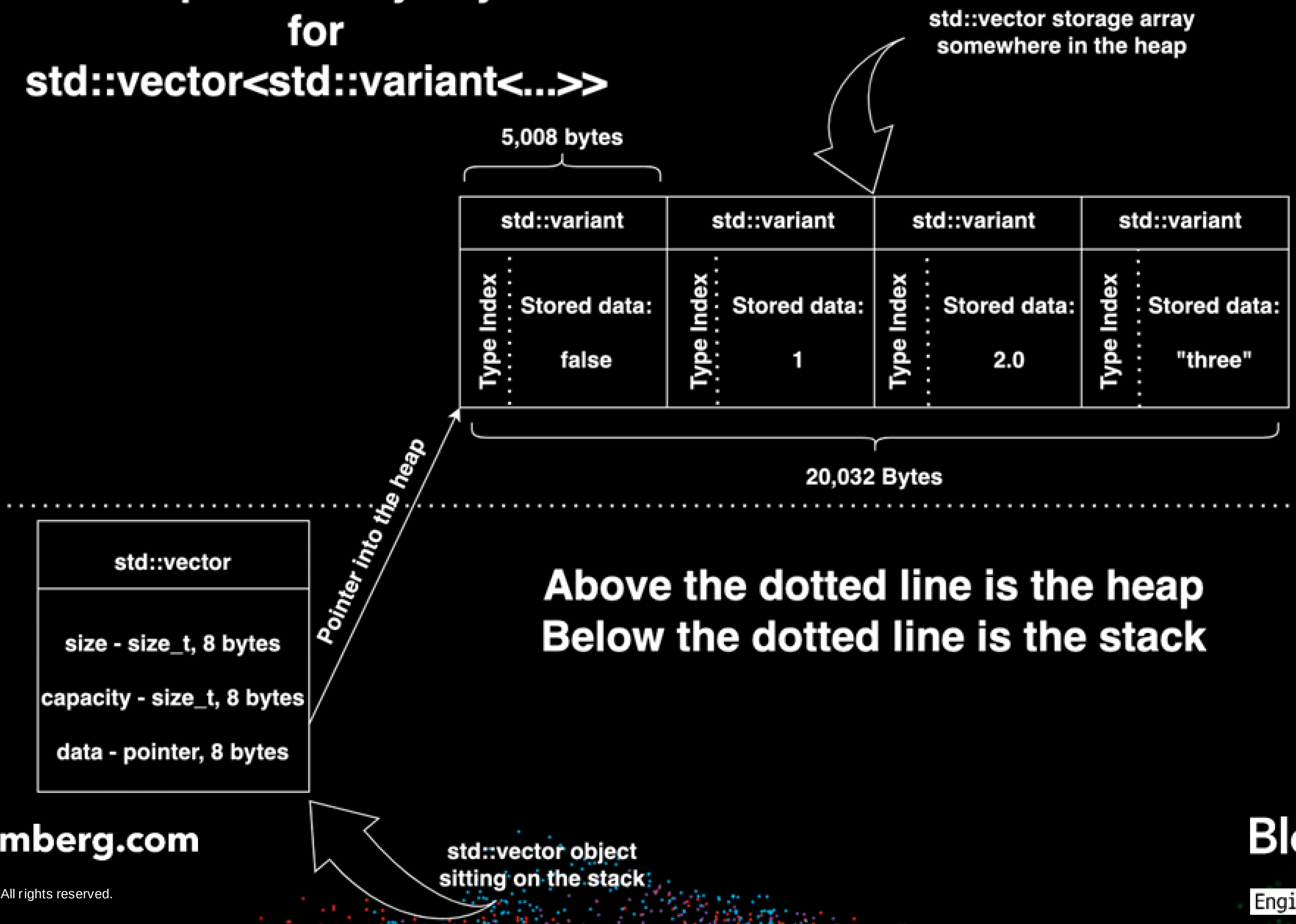
- While fine for trivial examples and slide code, it can lead to significant bloat in real-world code

- Original motivating use-case was modeling a time series of market events as a vector of variants, where one of the types in the variant happened to be 10x larger than the rest.

Trivia: How Much Memory is Used by this Vector?

```
#include <vector>
#include <string>
#include <variant>

struct big_data {
    char data[5000];
};

using combo_type = std::variant<bool, int, double, std::string, big_data>;

int main() {
    std::vector<combo_type> vec {false, 1, 2.0, "three"};
}
```

Example Memory Layout
for
std::vector<std::variant<...>>



Above the dotted line is the heap
Below the dotted line is the stack

Analysis of Memory Usage

- `std::variant<...>` stores the wrapped types inside of the class
- And so, it must pessimistically be at least as large as the largest T
- `std::vector<T>` assumes all elements are the same size
- Which brings us to our failure state, which requires ~20KB to store a single `bool`, `int`, `double`, and `std::string`
- Assuming a `bool` is 1 byte, an `int` is 4 bytes, a `double` is 8 bytes, and a `std::string` is 24 bytes, the minimum conceivable memory usage is 37 bytes, meaning this represents a bloat factor of 540x!

Why is `std::vector<std::variant<...>>` so wasteful?

- `std::vector` and `std::variant` are generically composable containers that don't cooperate with each other

- `std::variant` is specified such that it cannot allocate

- If you assume all data is the same size and have to store all types in-situ, this is necessarily the outcome

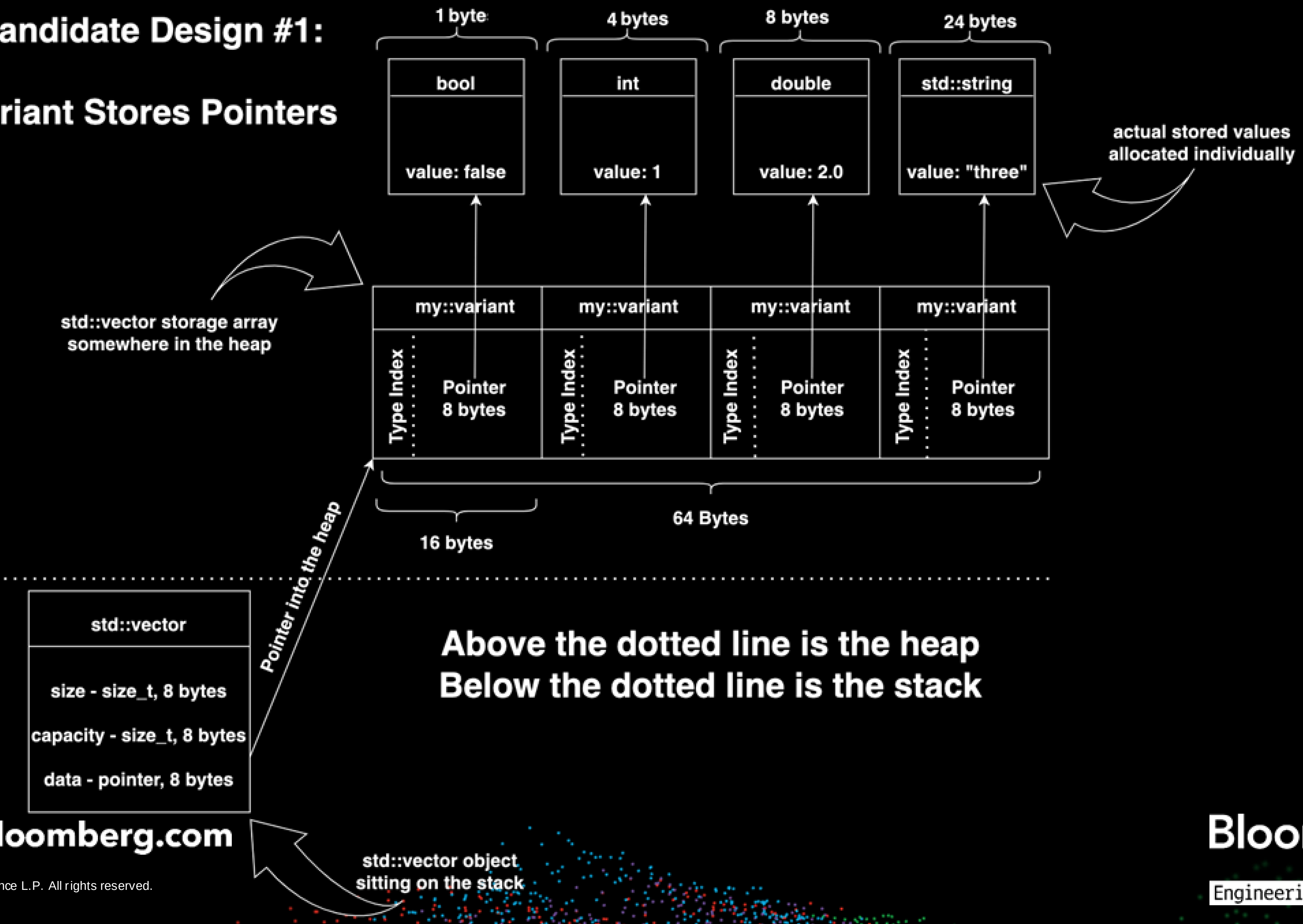
- Can we do better?

Candidate Design #1

- Say that we relaxed the limitation that `std::variant` can't allocate, where would that get us?

Candidate Design #1:

Variant Stores Pointers



Above the dotted line is the heap
Below the dotted line is the stack

Candidate Design #1

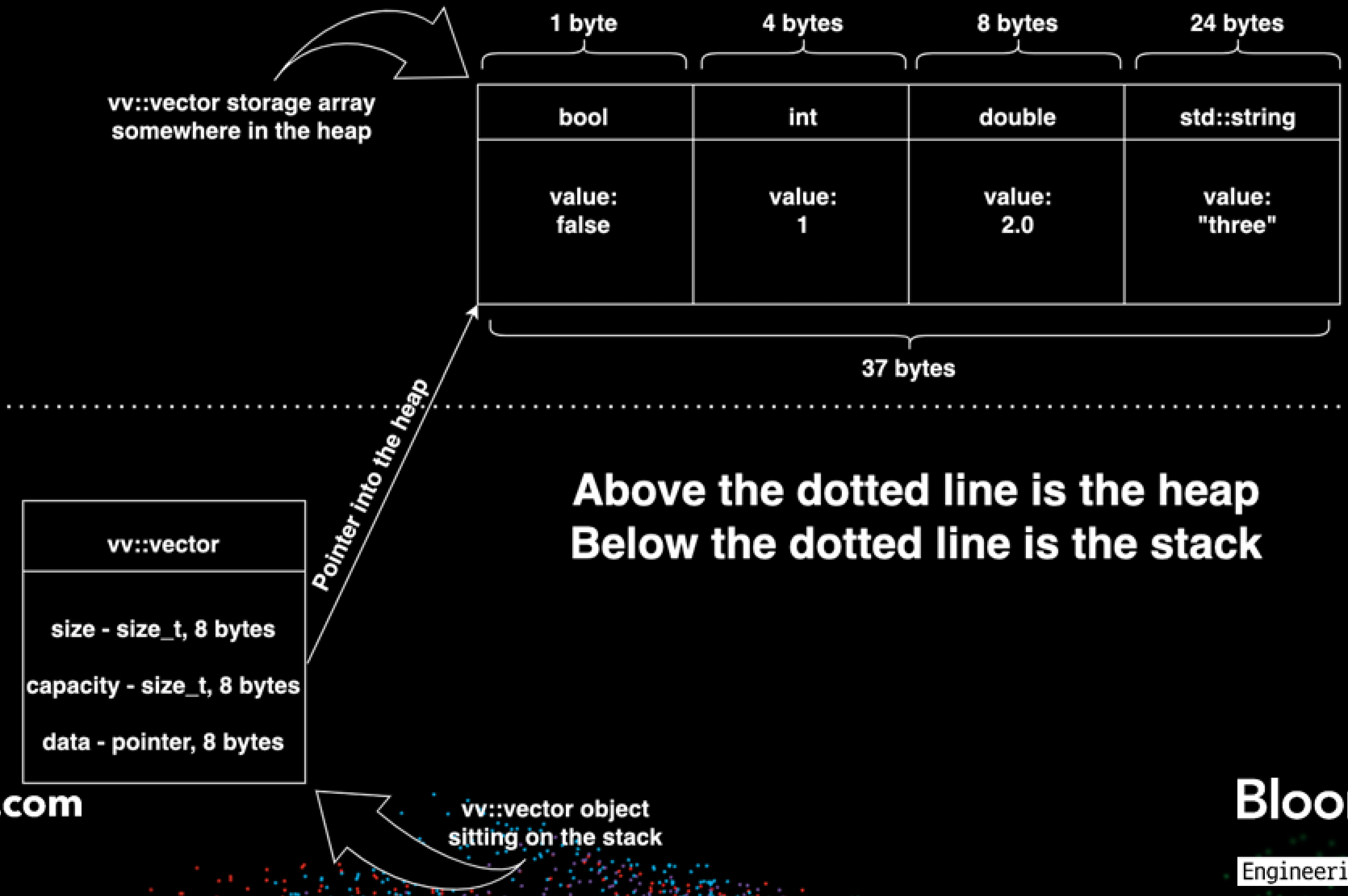
- New design uses only 101 bytes! We're now down to just 2.75x memory bloat
- However, *every* ***single*** `push_back()` now needs to allocate from the heap
- We've saved a lot of memory, and our implementation remains simple, but we've also fragmented the heap and *demolished* the performance characteristics of the original data structure
- Can we do better?

Candidate Design #2

- Performance overhead of allocating for every variant is way too expensive
  - Could try to mitigate this with a `std::pmr` memory pool.
  - Pointers themselves are still a source of significant memory bloat
  - Also exposes us to bloat/performance issues around pool fragmentation
- What if we instead wrote our own vector class that was actually aware of the different types it is storing?

Candidate Design #2:

Vector Stores Multiple Types



Above the dotted line is the heap
Below the dotted line is the stack

TechAtBloomberg.com

Bloomberg
Engineering

Candidate Design #2

- Design uses exactly 37 bytes, which was our theoretical minimum!
- That was easy!
- And people say “C++ is hard!” 🤭

Thank you!

https://techatbloomberg.com/cplusplus

https://www.bloomberg.com/careers

TechAtBloomberg.com

Bloomberg Engineering

Not So Fast!

😱 🤣 😭

TechAtBloomberg.com

Bloomberg
Engineering

Some Problems with Candidate Design #2

- How do we actually get things back out of this vector?

  - Everything is stored in one array, and each element is potentially a different size. How do we know where element N is?

  - Beyond this, each element is also potentially a different type! How do we know what type element N is?

Candidate Design #3

- Same as Design #2, but let's also add some additional storage to keep track of types and offsets

- We can use `int`s to store both the offsets and the types

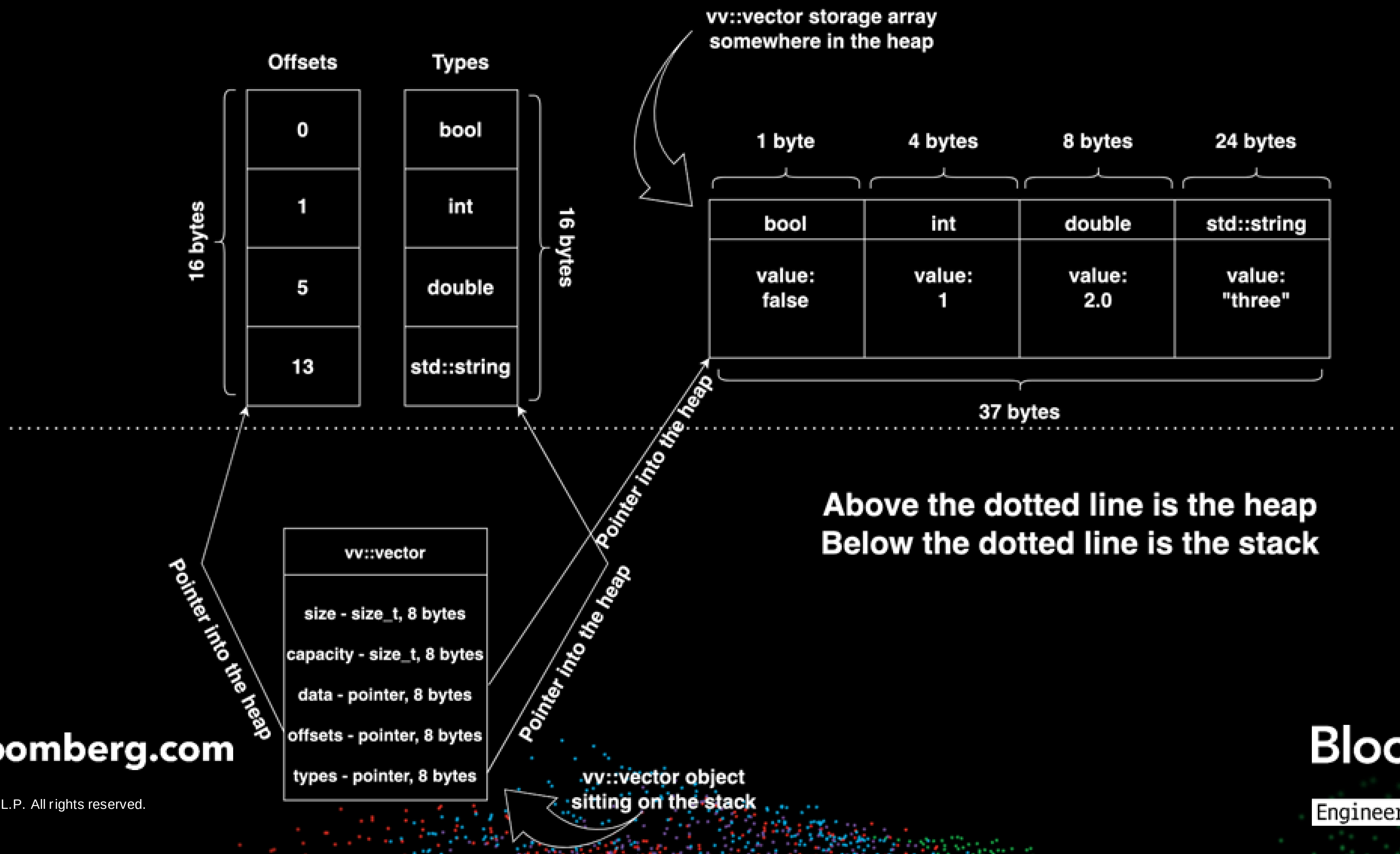
  — Offset can be stored as bytes from the base of the storage array

  — Type can be stored as the index in the variant's type list

  — 0 for `bool`, 1 for `int`, and so on

Candidate Design #3:

Vector Stores Multiple Types...
but this time, with Metadata!



Above the dotted line is the heap
Below the dotted line is the stack

TechAtBloomberg.com

Bloomberg
Engineering

Candidate Design #3

- Updated design now uses 69 bytes, so we're now both more memory efficient than Design #1 *and* significantly faster at the same time

- We can now correctly discern between different types, and we also know the offsets where things are located, so we're good to go, right?

- Let's write a quick test program to check… and just to be fun and quirky, let's try running it on Solaris

Design #3 Test Driver

```
#include <varvec.h>
#include <string>

using vector = vv::vector<bool, int, double, std::string>;

int main() {
    vector vec {false, 1, 2.0, "three"};
    for (auto elem : vec) {
        std::visit(elem, [] (auto val) { std::cout << val << std::endl; } );
    }
}
```

Design #3 Test Driver Results

```
sundev9:working $ ./a.out
0
Bus Error (core dumped)
```

The Problem of Memory Alignment 😱

- Welcome to the world of the compiler, or more specifically, the world that the compiler *hides* from you

- Most code never has to think about it, because both the language and compiler work together to abstract it away, but there are strict requirements imposed by the hardware on what kinds of addresses can legally be used by different machine types

- Let’s take a concrete example

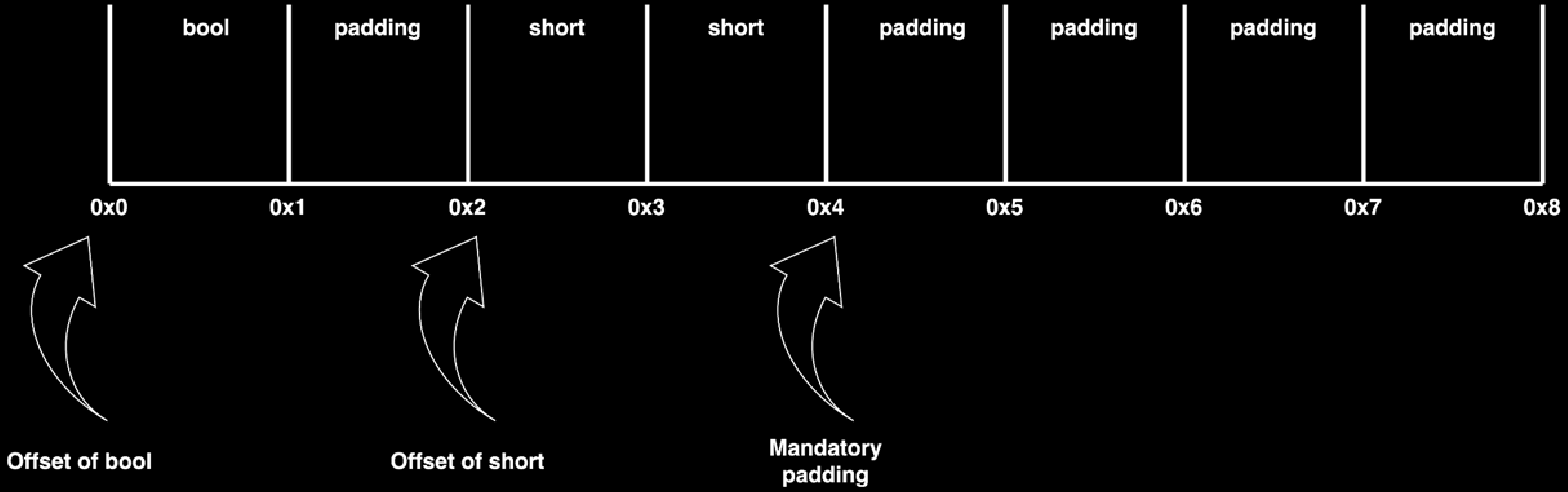
Memory Alignment Example

```
struct example {
    bool flag;
    short state;
    long counter;
};
```

```
static_assert(
    sizeof(example)== 16
);
```

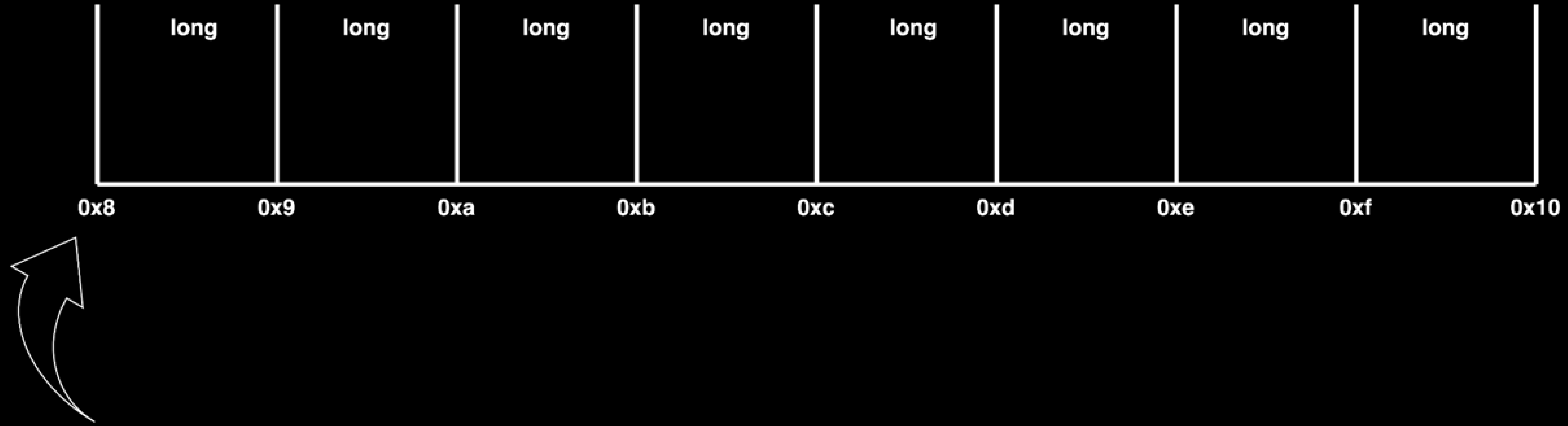
Exercise in Understanding Memory Alignment:

Example struct layout



Exercise in Understanding Memory Alignment:

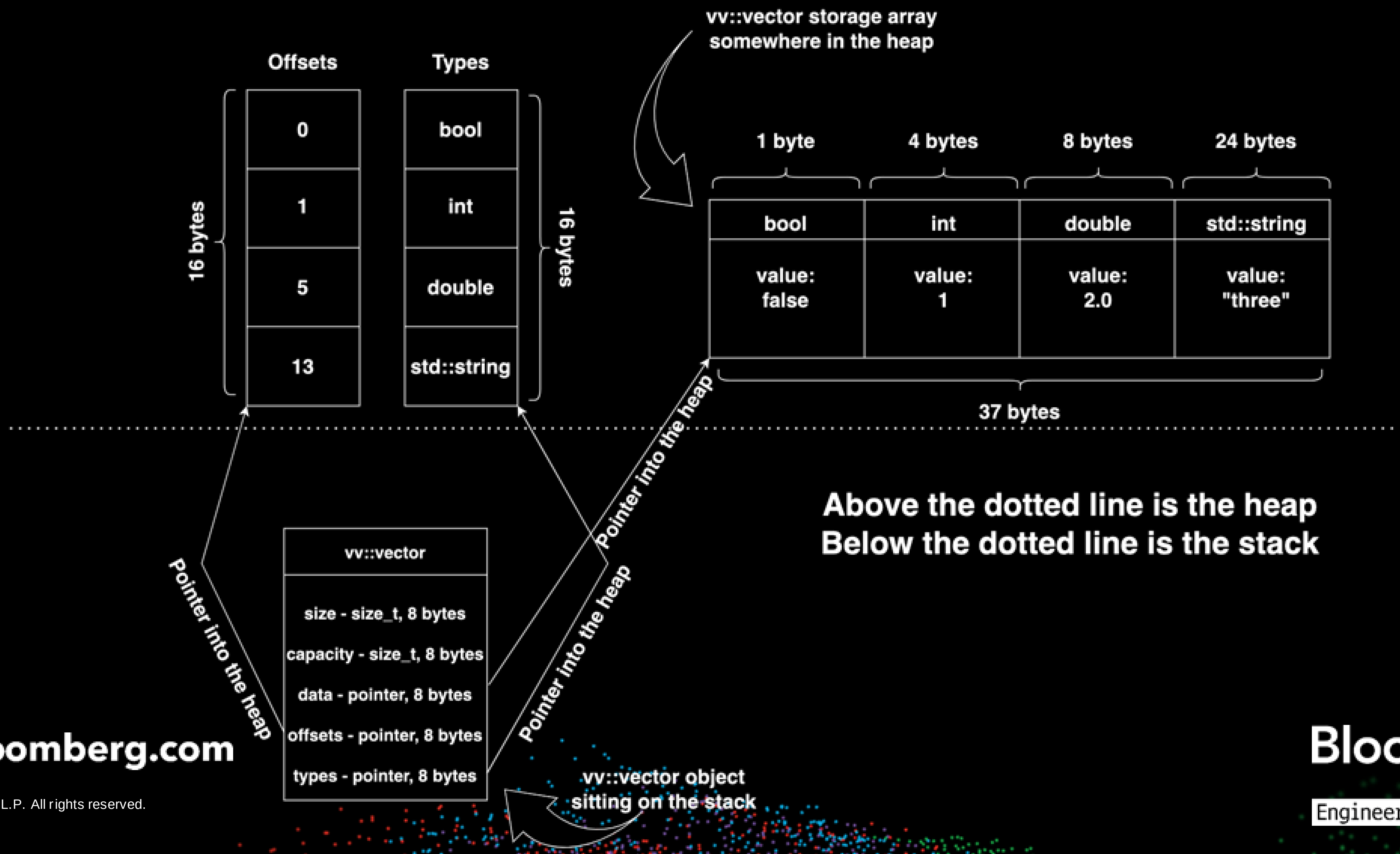
Example struct layout continued



Offset of long

Candidate Design #3:

Vector Stores Multiple Types...
but this time, with Metadata!



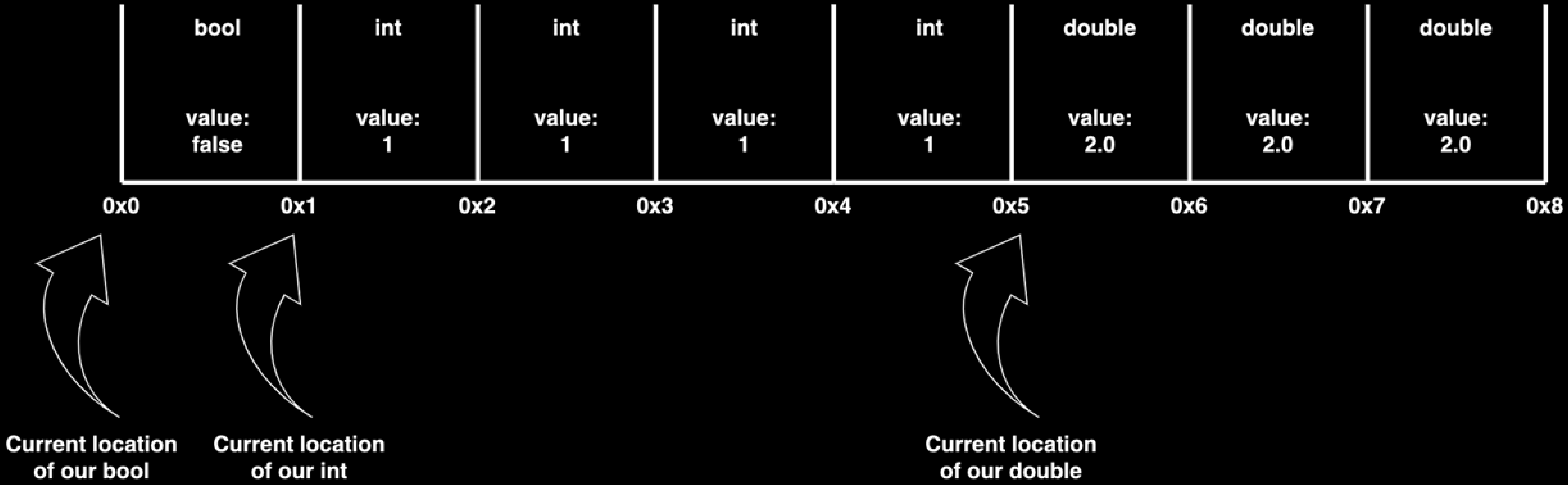
Above the dotted line is the heap
Below the dotted line is the stack

TechAtBloomberg.com

Bloomberg
Engineering

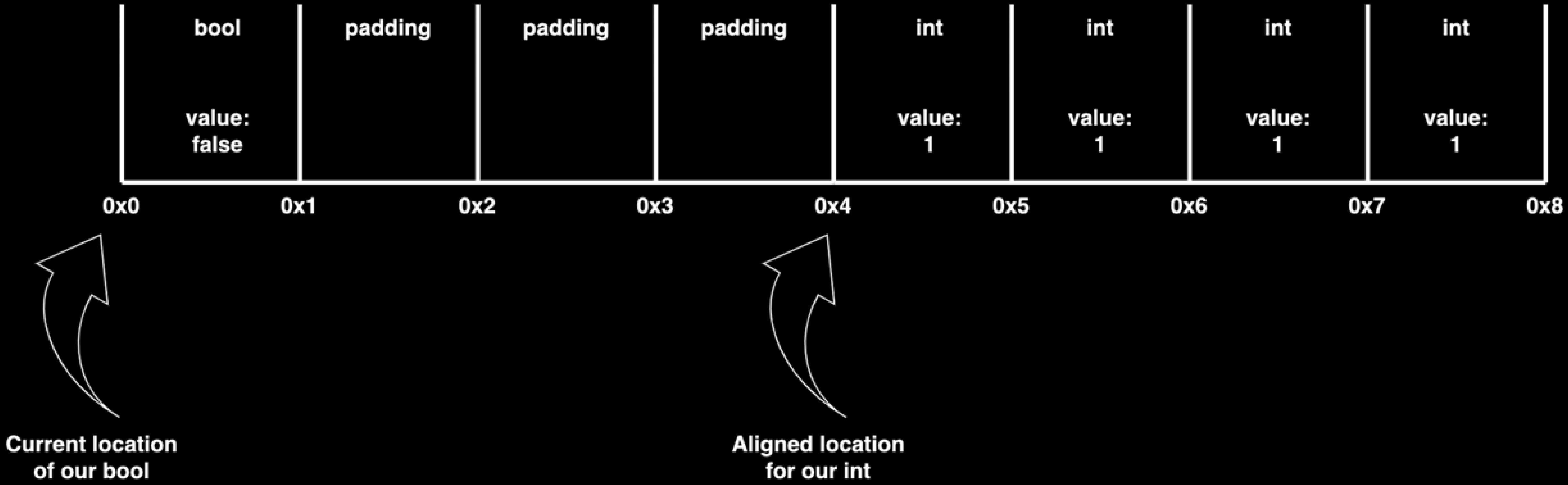
Exercise in Understanding Memory Alignment:

Our current storage layout



Exercise in Understanding Memory Alignment:

Corrected storage layout



TechAtBloomberg.com

Bloomberg Engineering

The Problem of Memory Alignment 😱

- Misaligned accesses are always UB, but the specific behavior depends on the platform.
- x86 handles this “automatically”.
  - — Previous code likely would succeed.
- SPARC traps with a memory bus exception.
- Undefined behavior sanitizer can be used to validate alignment on x86.

Candidate Design #4

- Same as Design #3, but let's ensure that our data is properly aligned this time around

- C++ has features that can help us:

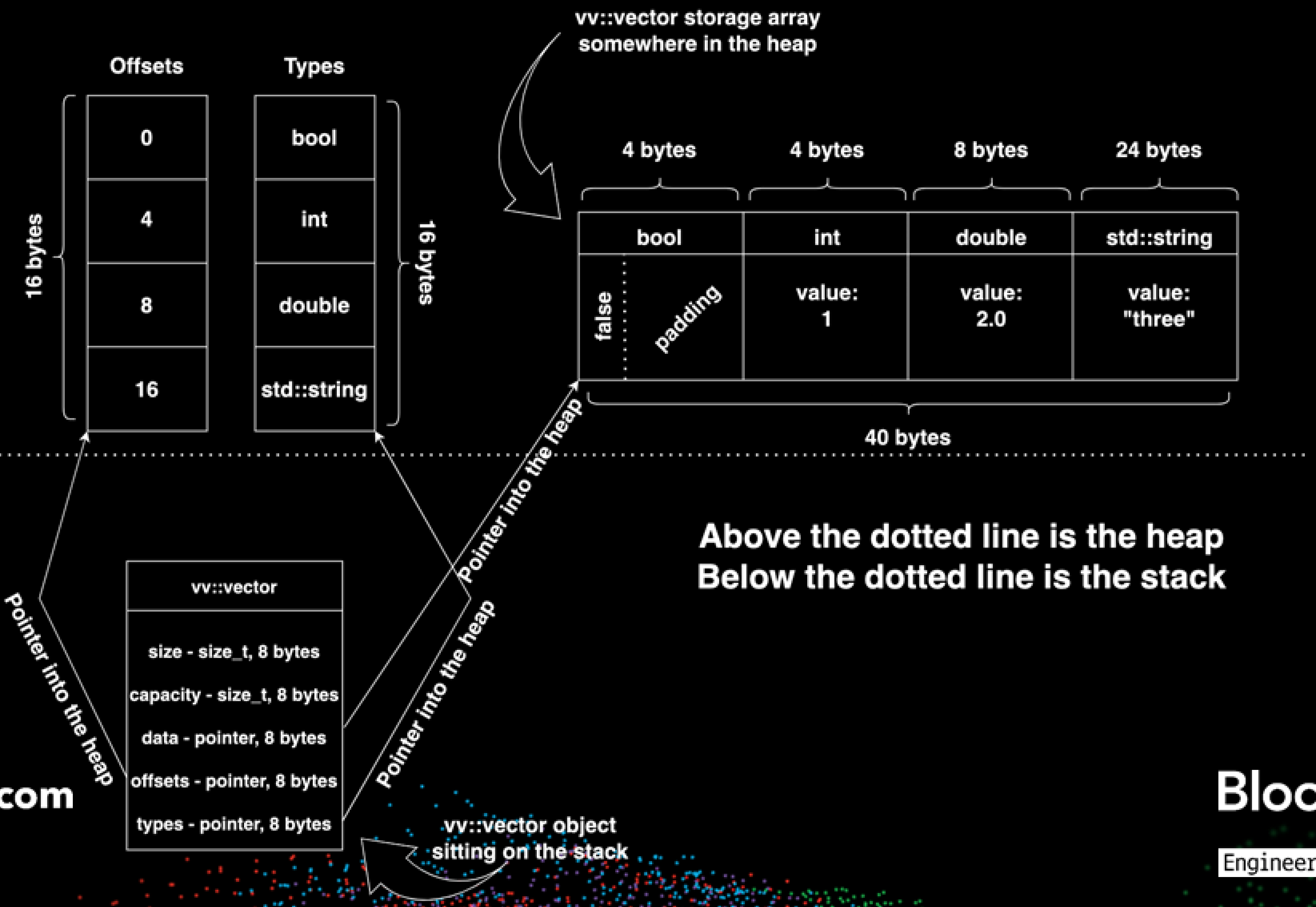
  — `alignof(T)` can be used to return the alignment required by type T

  — `alignas(T)` can be used to force a specific alignment

  — `new(std::align_val_t(N))` can be used to allocate aligned

Candidate Design #4:

Vector Stores Multiple Types...
but this time, with alignment!



Above the dotted line is the heap
Below the dotted line is the stack

Candidate Design #4

- Finally, we have something that actually works! 🎉 🎉

- At this point, our design requires 72 bytes, which is slightly worse than Design #3, but it actually works – and it's also a far cry from the 20KB in the original example

- Can we do better?

Candidate Design #5

- Currently, nearly half of our memory usage (32 bytes) goes into the offset and type storage; it has to be possible to do better than this

- First observation: there will always be a static number of types that our vector is parameterized by
  - It’s unlikely the vector will ever use more than a handful of types
  - If we know how many types we’ll have to store, we can compute the minimum number of bits to uniquely distinguish types, and store the type info in a bitmap, radically decreasing storage requirements

Candidate Design #5

- Second observation: if we can put an upper bound on the size of the vector, we can compute the smallest integer type that can represent the maximum offset
  - It needn’t actually be a hard limit. If we have the ability to “swap out” or “rebuild” our index storage at runtime, we can dynamically select the smallest representation that will suffice

Candidate Design #5

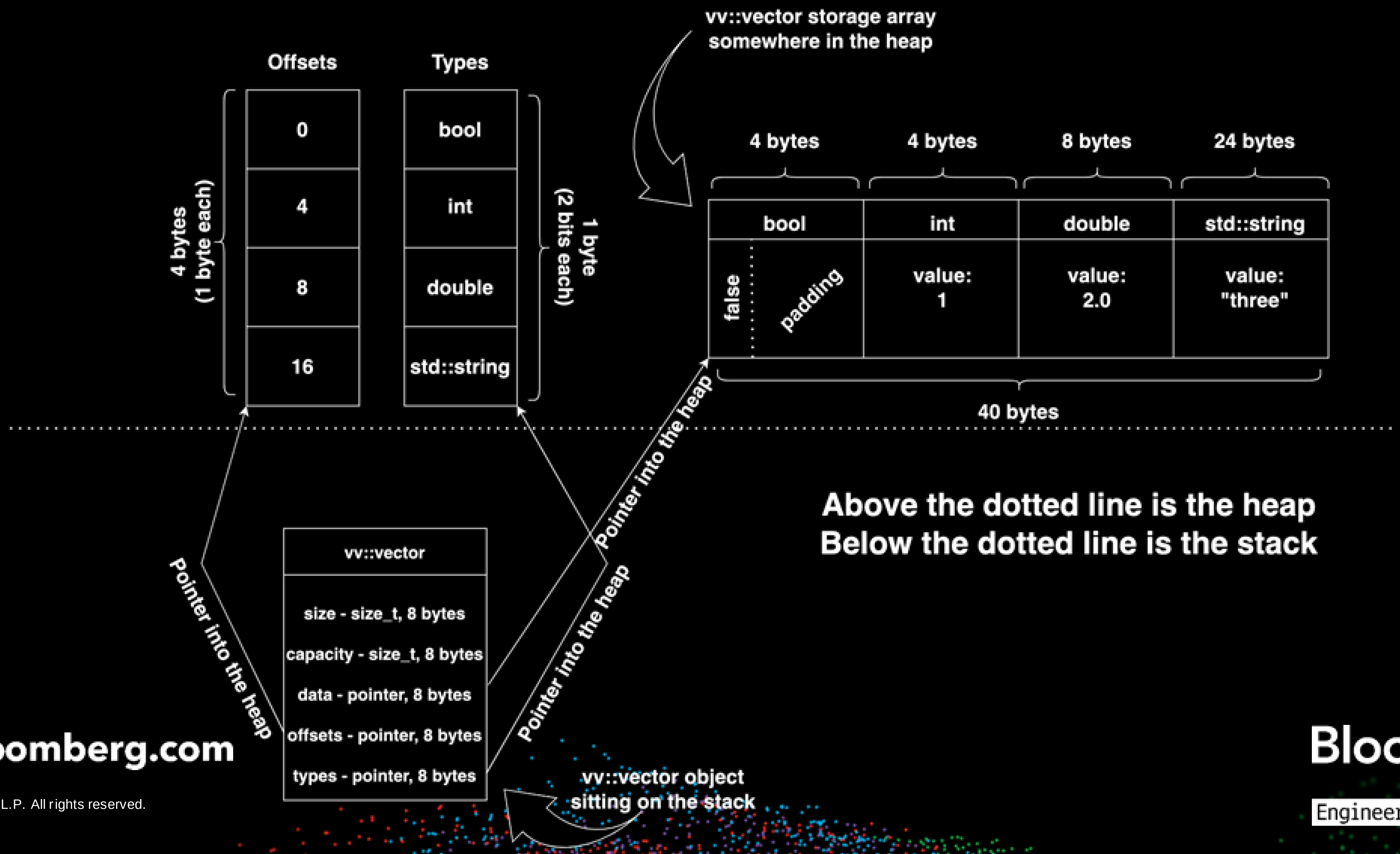
- Replace the current type storage with a bitmap that stores types using the minimum bits possible

- Replace the current offset storage with a virtual interface that will allow us to transparently “rebuild” and “swap out” the offset storage with the smallest usable version at runtime

- Combining both approaches should get us significantly closer to our theoretical minimum

Candidate Design #5:

Compactifying the metadata storage



Above the dotted line is the heap
Below the dotted line is the stack

Candidate Design #5

- This optimization brings us to a grand total of 45 bytes for the full representation, an overhead factor of only about 20% versus the theoretical minimum

- Can we still do better? Perhaps…

A Few Additional Ideas

- If we can handle the dynamic case, a static vector implementation should be a trivial extension
  - Static vector can store contents in-situ in the class
  - C++ 20 `requires` clause can conditionally disable copy/move operations
  - Very natural extensions for `constexpr`

A Few Additional Ideas

- For **absolute** minimum memory usage, we *could:*
  - Store `trivially_copyable` types contiguously, without alignment
  - Copy-in/out and realign through the stack on access
- Probably to the point of diminishing returns

Implications of Design

- We now have a candidate design for a data structure that radically reduces the memory footprint from our original example, while retaining the same functionality
- This works nicely to solve the original problem, but do these changes have any implications for the data structure’s wider API?

The Challenges of Mixed Type Storage

- Many of the member functions in `std::vector` silently assume that all of its elements are of the same size.

- What does `capacity` mean for this data structure?

- What does `reserve` mean for this data structure?

- What about `resize`?

- All of these functions operate on the number of elements, from which the number of bytes can be directly inferred. We cannot do this

The Challenges of Mixed Type Storage

```cpp
#include <string>
#include <varvec.h>
#include <iostream>

int main() {
    vv::vector<bool, int, double, std::string> vec;
    vec.reserve_bytes(64);
    vec.reserve<std::string>(16);
    vec.resize(16, "a test string");
    std::cout << vec.capacity<std::string>() << std::endl;
    std::cout << vec.capacity_bytes() << std::endl;
}

// => 16
// => 384
```

Implications of Design

- Semantic challenges for “number of elements” functions like `capacity`
- Complexity of implementation for functions like `insert/erase`
- Are there any other implications?

The Woes of `operator []`

- While the design presented thus far minimizes memory usage quite nicely, it has unfortunate implications for the subscript operator

- Consider the following code:

TechAtBloomberg.com

Bloomberg Engineering

The Woes of `operator []`

```
#include <vector>
#include <string>
#include <variant>
#include <iostream>

using combo_type = std::variant<bool, int, double, std::string>;

int main() {
    std::vector<combo_type> vec {false, 1, 2.0, "three"};
    vec[3] = "one plus two";
    std::cout << std::get<std::string>(vec[3]) << std::endl;
}

// => one plus two
```

The Woes of `operator []`

- How does the assignment inside of `main` actually work?
  - The subscript operator of `std::vector` returns an lvalue reference to the requested index, of type `std::variant<...>&`
  - The assignment operator of `std::variant` is called, and it does some template magic to ultimately select and call the assignment operator of `std::string`
  - The assignment operator of `std::string` receives the string literal and changes its value, causing the print statement to see the new value

TechAtBloomberg.com

Bloomberg Engineering

The Woes of `operator []`

- What happens if we try the same with our vector?

The Woes of `operator []`

```
#include <string>
#include <variant>
#include <iostream>
#include <varvec.h>

using vector = vv::vector<bool, int, double, std::string>;

int main() {
    vector vec {false, 1, 2.0, "three"};
    vec[3] = "one plus two";
    std::cout << std::get<std::string>(vec[3]) << std::endl;
}

// => three
```

TechAtBloomberg.com

Bloomberg Engineering

The Woes of `operator []`

- The assignment runs fine, and yet the value doesn't update. What gives?
- The issue lies in the first bullet-point of our assignment explanation:
  - "The subscript operator of `std::vector` returns an lvalue reference…"
- `std::vector` is storing a variant, and so it can return a reference to that variant upon access. Our vector, however, is *not* storing an actual variant, and therefore has no reference to give
- Upon access, our vector creates a temporary variant to return, which *does* get assigned to, but it then evaporates at the end of the line

The Woes of `operator []`

- This is extremely unfortunate, but is an inevitable outcome of not actually storing variants inside the vector

- Could theoretically work around this by returning a proxy class, but this seems like an anti-pattern

- To work around this, we need some new APIs:

The Woes of `operator []`

```
#include <string>
#include <variant>
#include <iostream>
#include <varvec.h>

using vector = vv::vector<bool, int, double, std::string>;

int main() {
    vector vec {false, 1, 2.0, "three"};
    vec.get<int>(1) = -1;
    vec.get<std::string>(3) = "one plus two";
    vec.visit(1, [] (auto& val) { std::cout << val << std::endl; });
    std::cout << std::get<std::string>(vec[3]) << std::endl;
}
// => -1
// => one plus two
```

The Woes of `operator []`

- The `.get<T>(idx)` and `.visit<T>(idx, [] (auto&) {})` APIs both model their STL equivalents and allow for in-place mutation

- Note that the subscript operator still works as expected for read-only interaction, like so:

The Woes of `operator []`

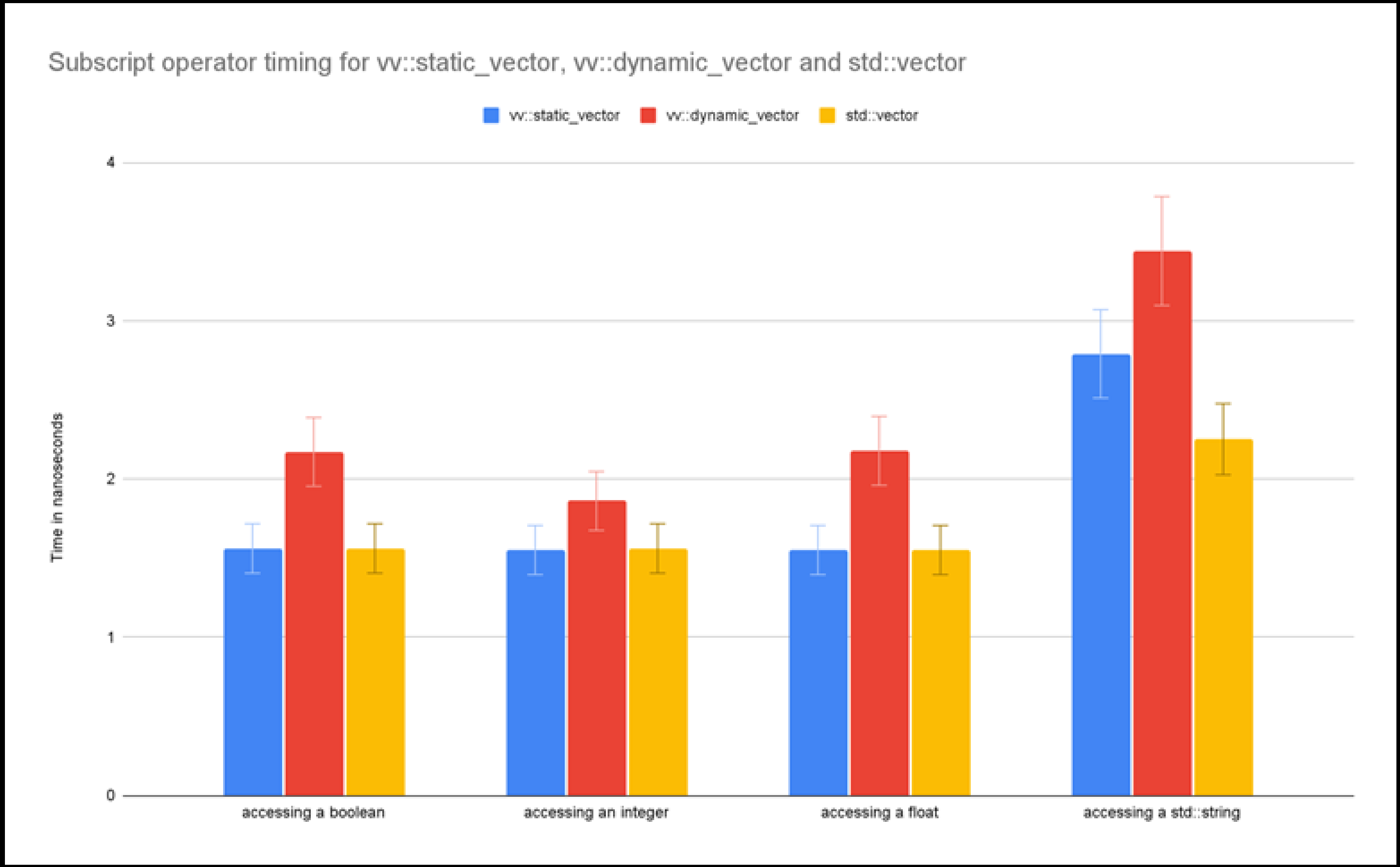
```
#include <string>
#include <variant>
#include <iostream>
#include <varvec.h>

using vector = vv::vector<bool, int, double, std::string>;

int main() {
    vector vec {false, 1, 2.0, "three"};
    std::visit(
        [] (auto&& val) { std::cout << val << std::endl; },
        vec[2]
    );
}

// => 2.0
```

Implications of Design

- Semantic challenges for “number of elements” functions like `capacity`
- Complexity of implementation for functions like `insert/erase`
- Not storing actual variants leads to difficulties for the subscript operator

Benchmarks!

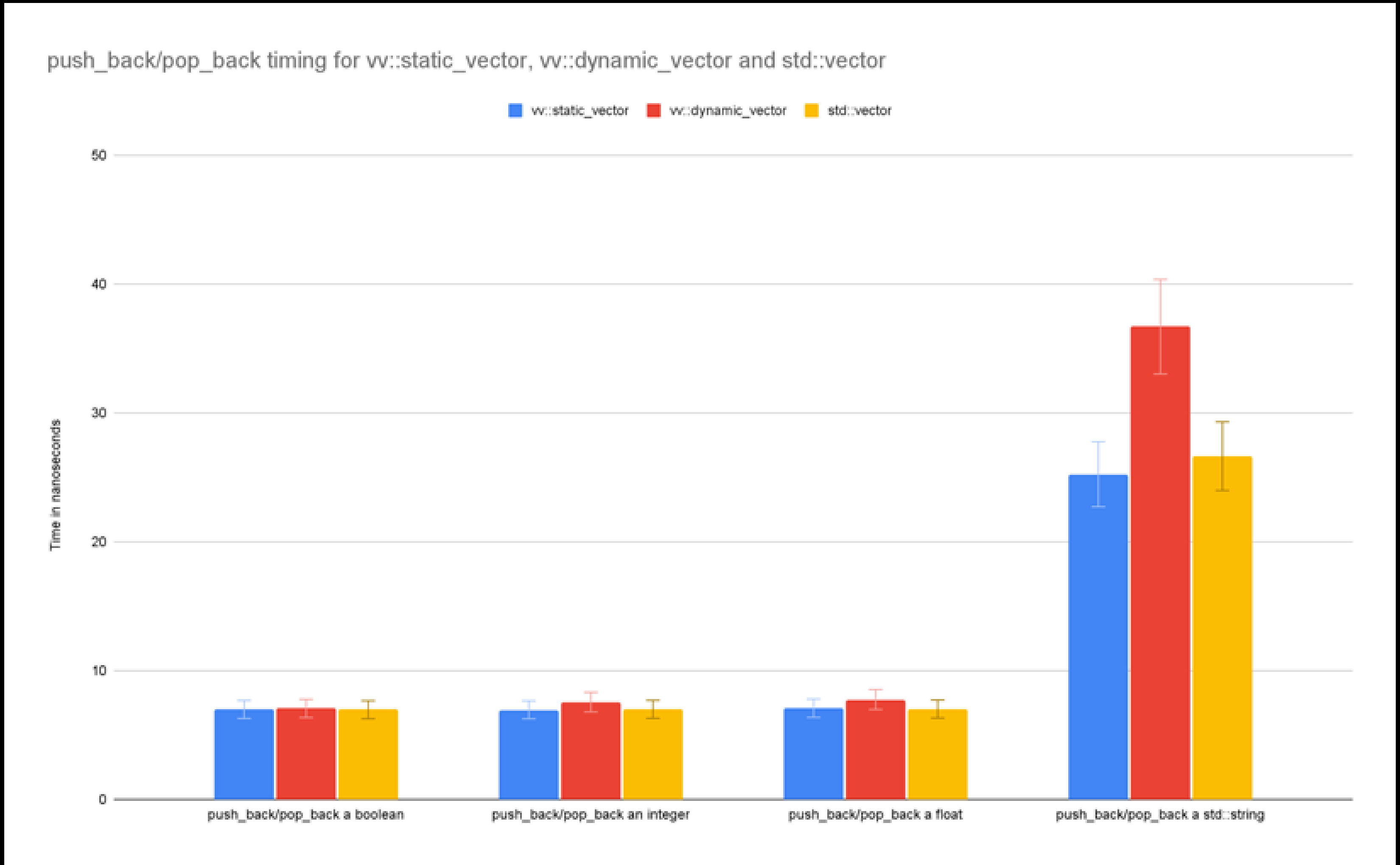
Benchmarks!



TechAtBloomberg.com

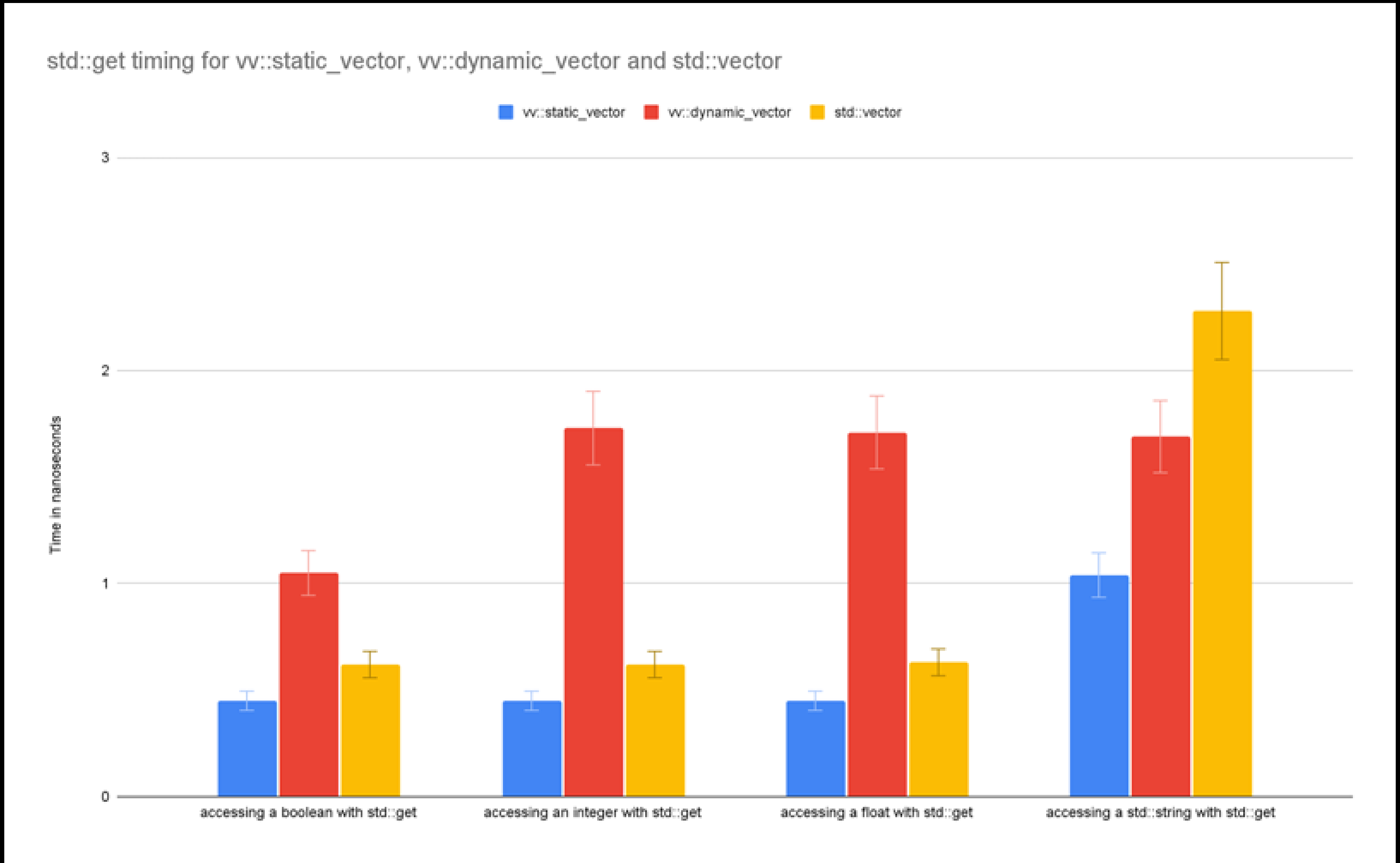
Bloomberg Engineering

Benchmarks!



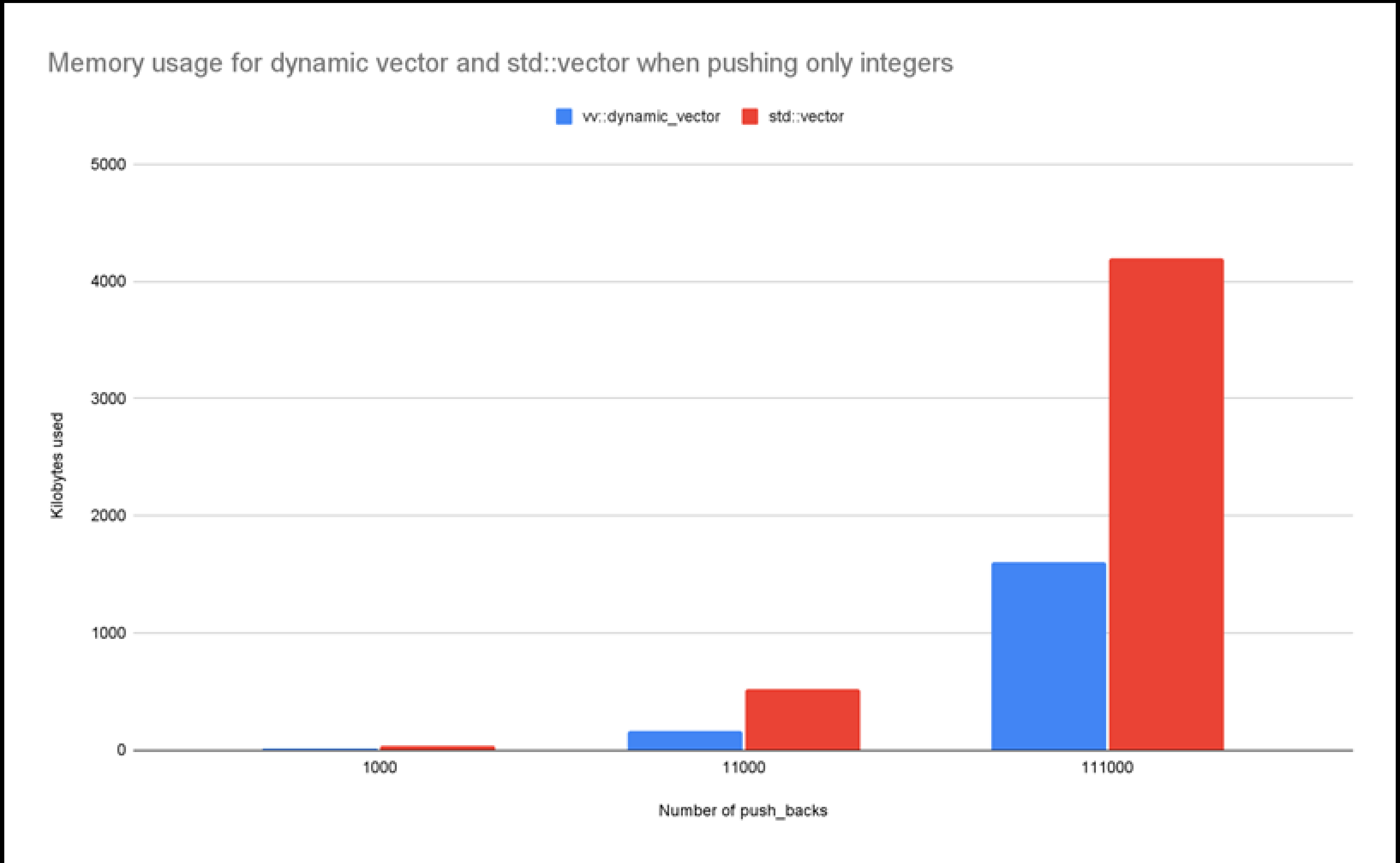
push_back/pop_back timing for vv::static_vector, vv::dynamic_vector and std::vector
vv::static_vector
vv::dynamic_vector
std::vector
Time in nanoseconds
50
40
30
20
10
0
push_back/pop_back a boolean
push_back/pop_back an integer
push_back/pop_back a float
push_back/pop_back a std::string

Benchmarks!



std::get timing for vv::static_vector, vv::dynamic_vector and std::vector
vv::static_vector
vv::dynamic_vector
std::vector
Time in nanoseconds
3
2
1
0
accessing a boolean with std::get
accessing an integer with std::get
accessing a float with std::get
accessing a std::string with std::get

Benchmarks!

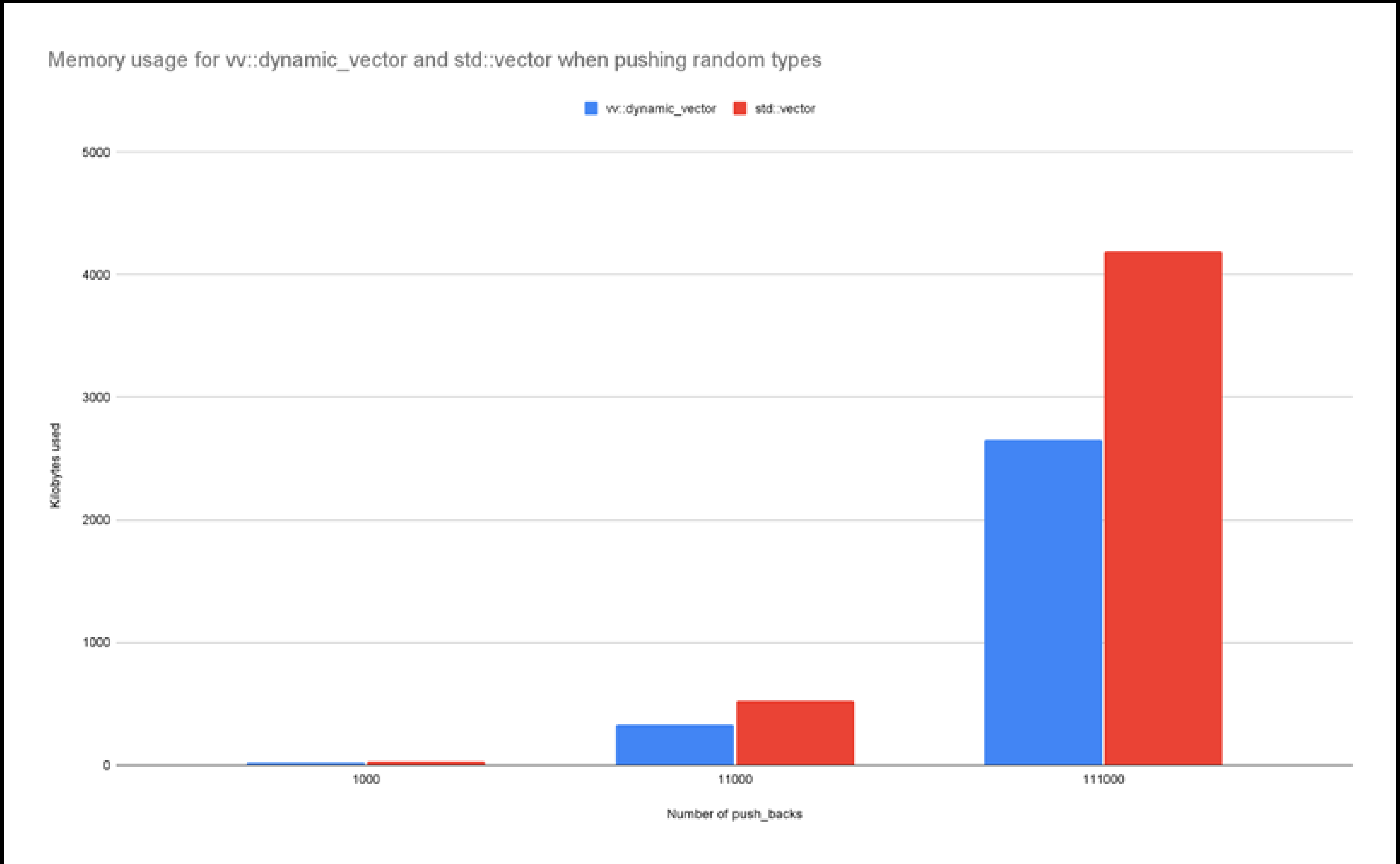


TechAtBloomberg.com

Bloomberg

Engineering

Benchmarks!



TechAtBloomberg.com

Bloomberg
Engineering

Wrapping Up

- Final Design
  - — Retain contiguous storage array, supported by metadata to track offsets and types
  - — Bit-pack all the things
  - — Deal with mixed alignment gracefully
  - — Provide alternative APIs for the subscript operator to work around fundamental design implications

Wrapping Up

- The data structure presented in this talk is a simplified version of the actual data structure created, and leaves many things unhandled, such as:
  - — Static vs. dynamically resizing vectors
  - — Handling types with throwing move constructors
  - — Dynamic growth strategy
  - — Exception safety guarantees

Thank you!

https://techatbloomberg.com/cplusplus

https://www.bloomberg.com/careers

TechAtBloomberg.com

Bloomberg Engineering