How to Hold a T

CJ Johnson

Welcome to "How to Hold a T"

CJ Johnson

Google NYC

Abseil

WG21

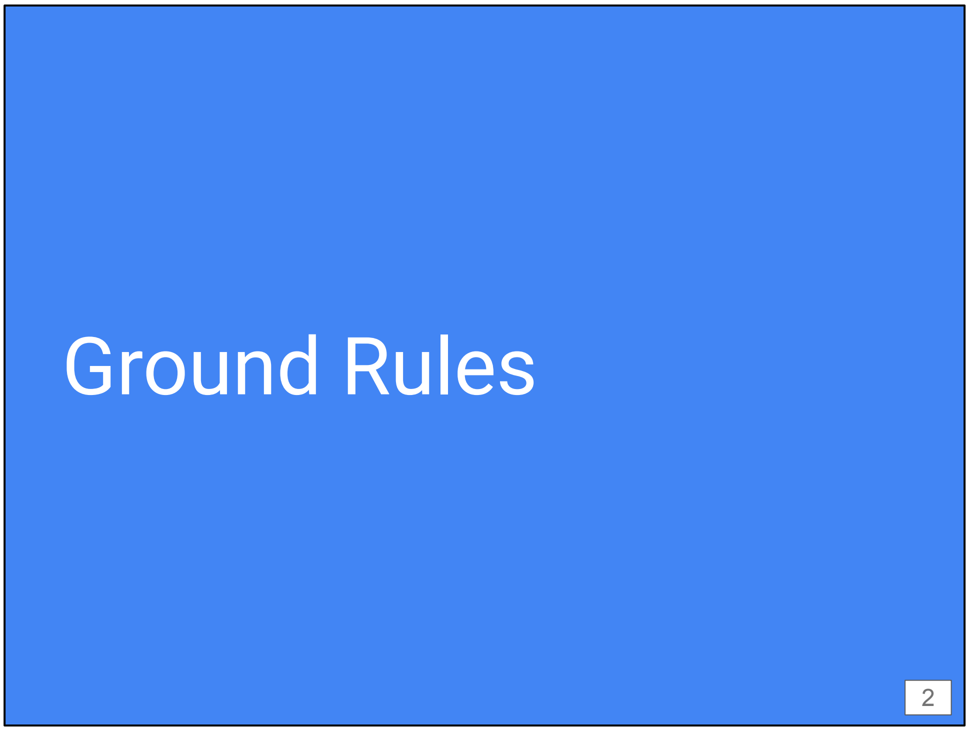
1

My name is CJ Johnson.

I work in Google's NYC office on the Abseil team.

Abseil is an open source utility library, and you can learn more about it at abseil.io.

As part of that, lately I've been participating in the ISO C++ standards meetings.



Ground Rules

2

And just a few quick ground rules.

First, please hold your questions until the end.

Every slide has a number at the bottom right, so if you aren't sure about something, just write down the slide number and we can come back to it at the end.

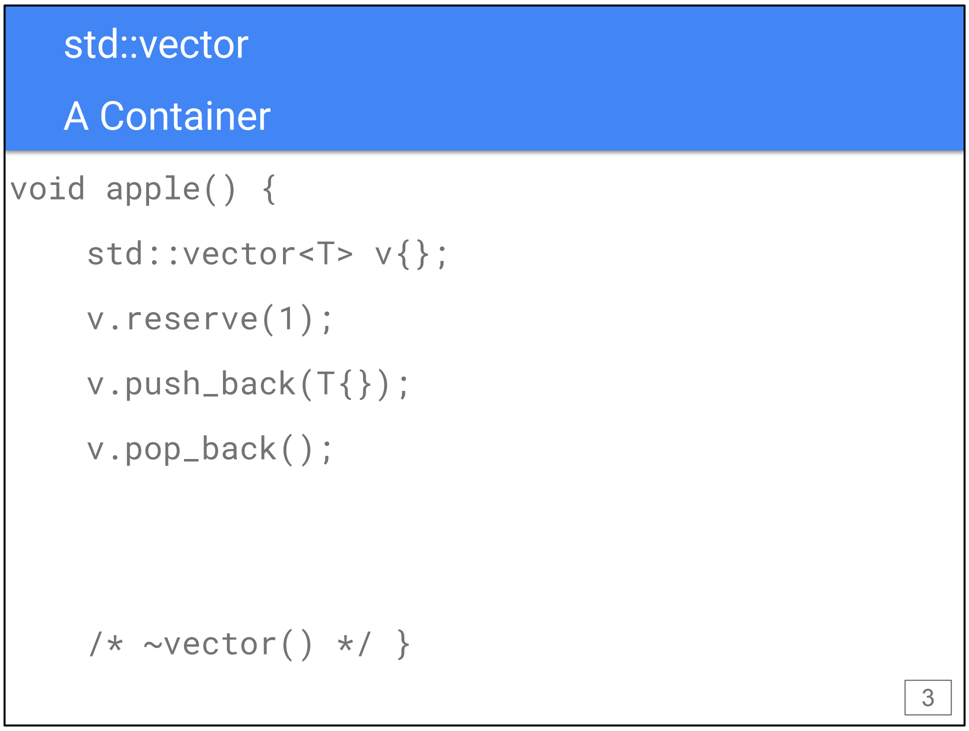
Additionally, this talk will include some half truths and incomplete information.

This is intentional!

It helps keep the talk focused and makes the content more approachable.

Just a fair warning. :)

With that, let's dive into the content….



std::vector

A Container

```
void apple() {
    std::vector<T> v{};
    v.reserve(1);
    v.push_back(T{});
    v.pop_back();



    /* ~vector() */ }
```

3

So let's talk about std::vector.

Specifically, let’s think about how it is able to “contain” objects, since it is a “container”.

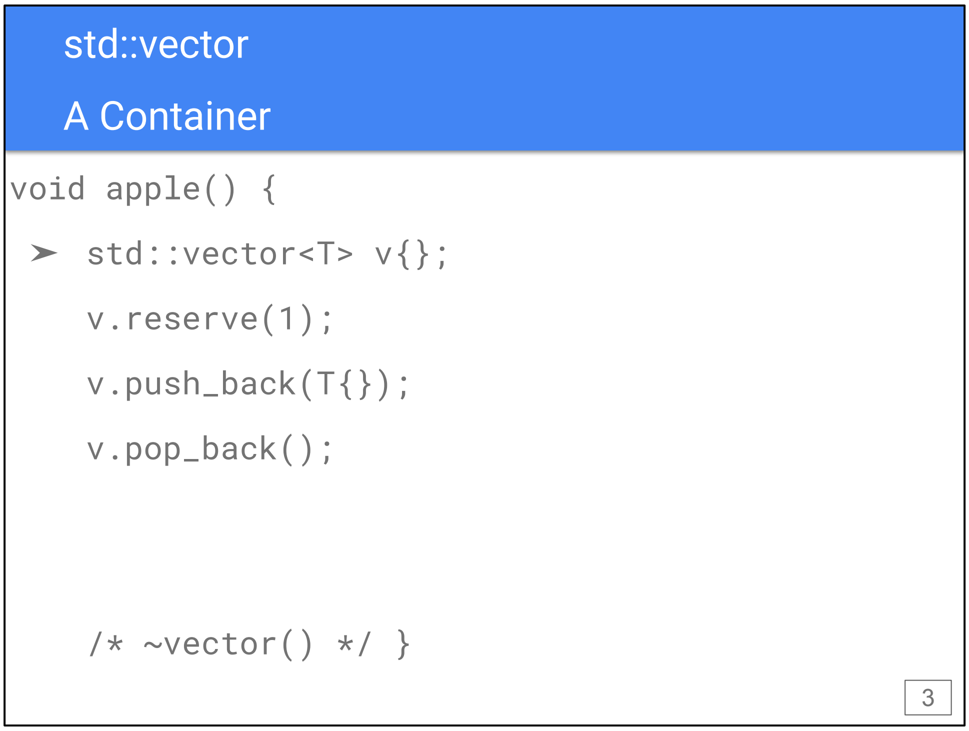
On the slide, we have the function “apple”.

Inside apple, we create a vector of T and mutate it a few times.

For the purposes of this talk, “T” is a placeholder to mean ANY arbitrary type.

Nothing about this talk is specific to templates.

So, let’s try and reason about what’s going on here.



std::vector

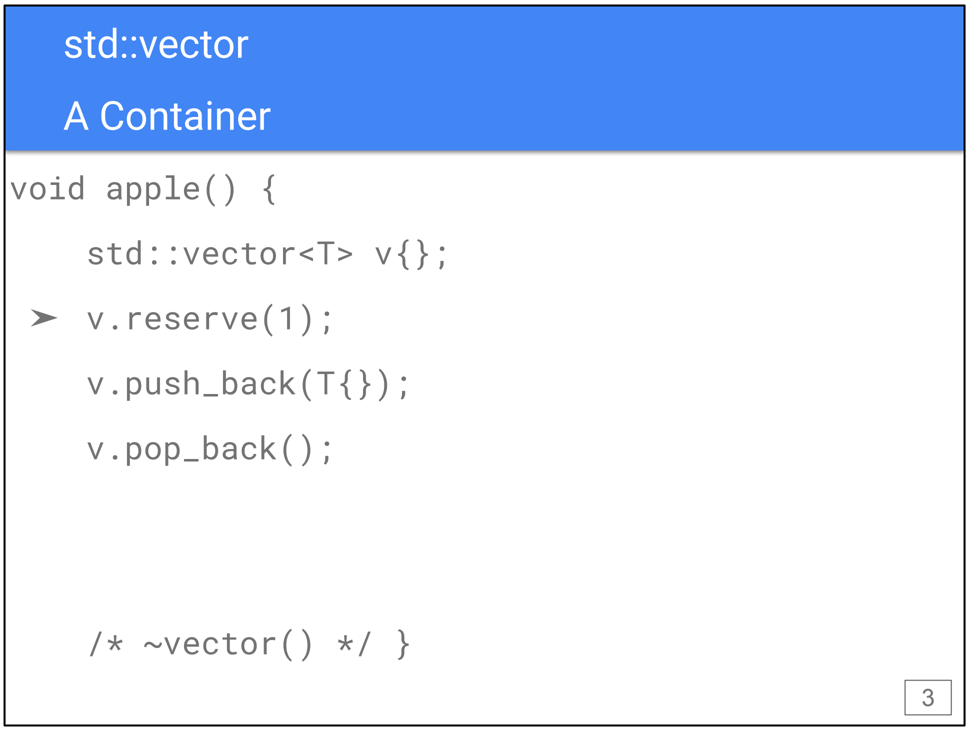
A Container

```
void apple() {
  ➢  std::vector<T> v{};
     v.reserve(1);
     v.push_back(T{});
     v.pop_back();



     /* ~vector() */ }
```

3

Initially, we’re creating a vector.

It’s being default constructed so it starts out empty, meaning it has no values and allocates no memory.



std::vector

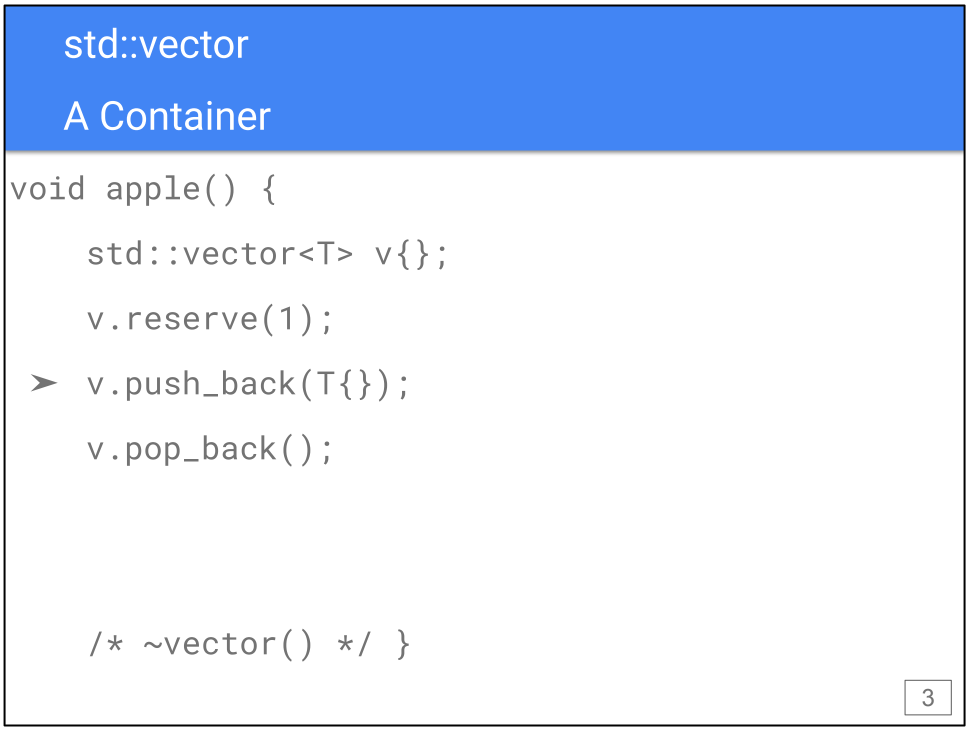
A Container

```
void apple() {
    std::vector<T> v{};
  ➤ v.reserve(1);
    v.push_back(T{});
    v.pop_back();



    /* ~vector() */ }
```

3

Then in the second statement we call reserve with an argument of 1.

This allocates enough memory for 1 instance of T, but it does NOT call T’s constructor.



std::vector

A Container

```
void apple() {
    std::vector<T> v{};
    v.reserve(1);
  ➤ v.push_back(T{});
    v.pop_back();



    /* ~vector() */ }
```

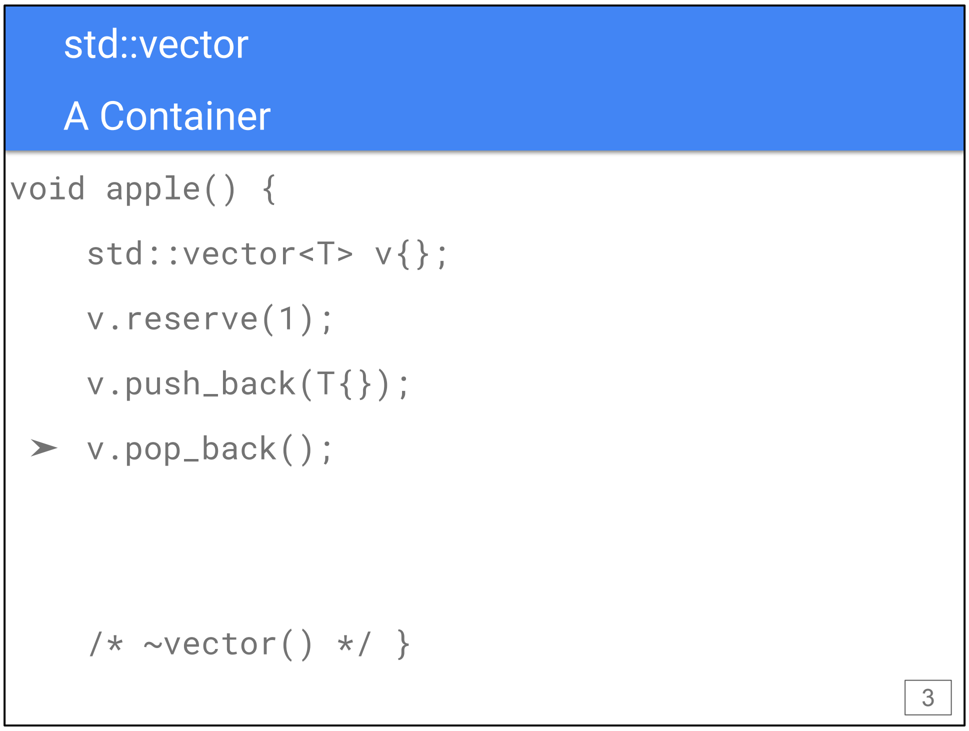
3

Then here’s where things start to get interesting…

push_back is constructing a T inside the memory that was allocated by the previous reserve call.

Because the vector already has enough memory for the new T instance, it does NOT need to reallocate.

This call to push_back ONLY constructs a T, it does NOT allocate memory.



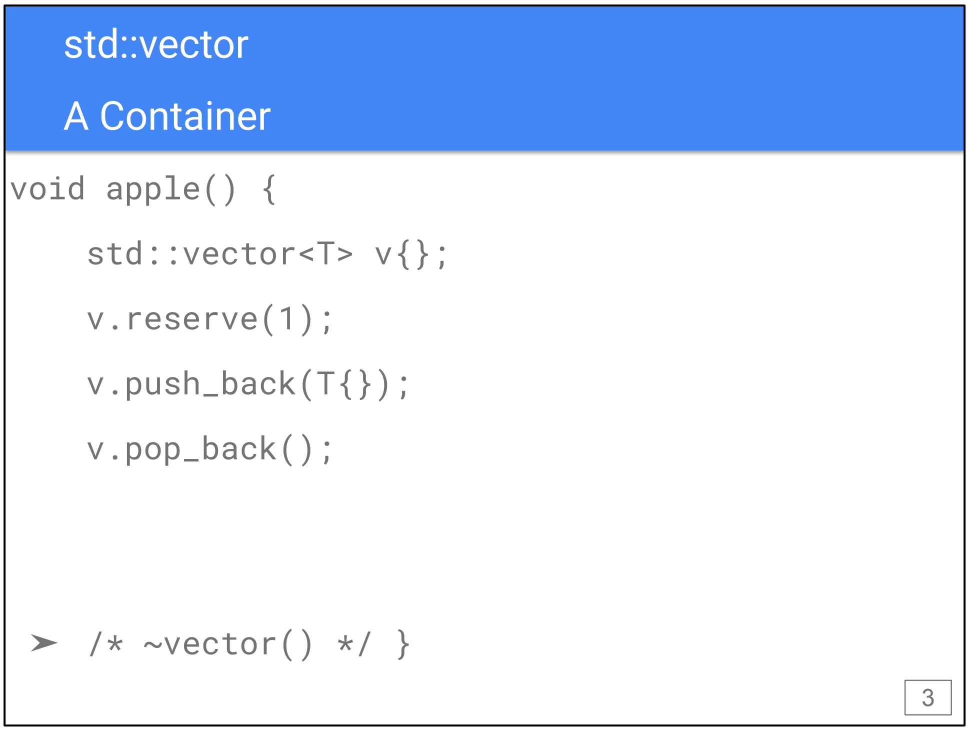
std::vector

A Container

```
void apple() {
    std::vector<T> v{};
    v.reserve(1);
    v.push_back(T{});
  ➤ v.pop_back();



    /* ~vector() */ }
```

3

And similarly, the call to pop_back destroys the T instance, but it does NOT deallocate the memory.

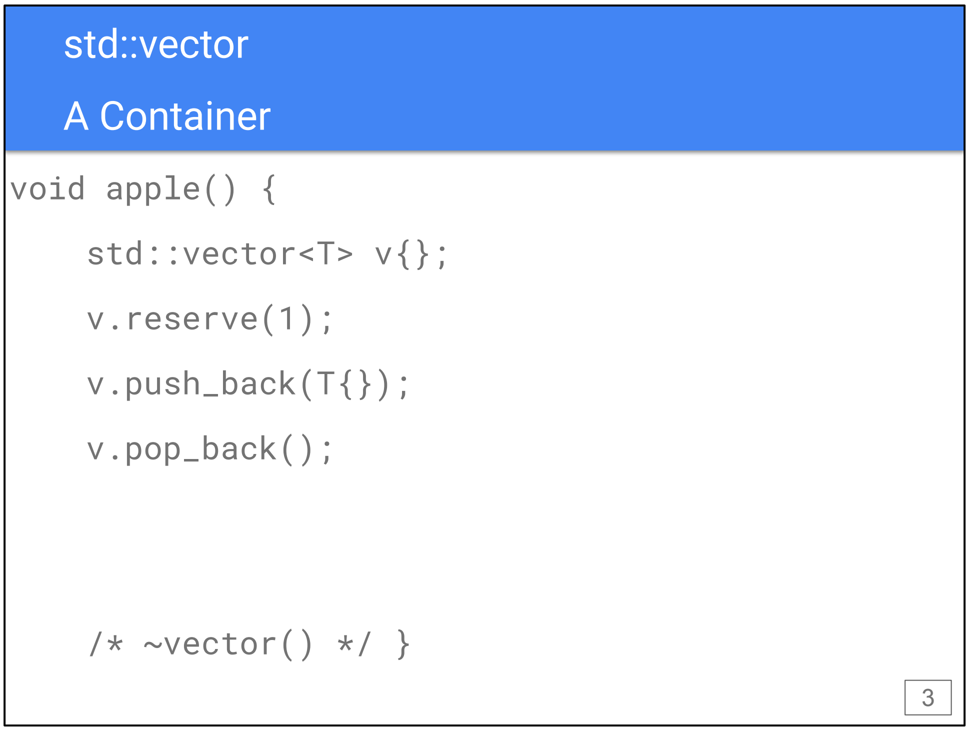

std::vector

A Container

```
void apple() {
    std::vector<T> v{};
    v.reserve(1);
    v.push_back(T{});
    v.pop_back();



 ➤  /* ~vector() */ }
```

3

It’s not until this line, with an implicit call to the vector destructor, that the memory for T is deallocated.



std::vector

A Container

```
void apple() {
    std::vector<T> v{};
    v.reserve(1);
    v.push_back(T{});
    v.pop_back();


    /* ~vector() */ }
```

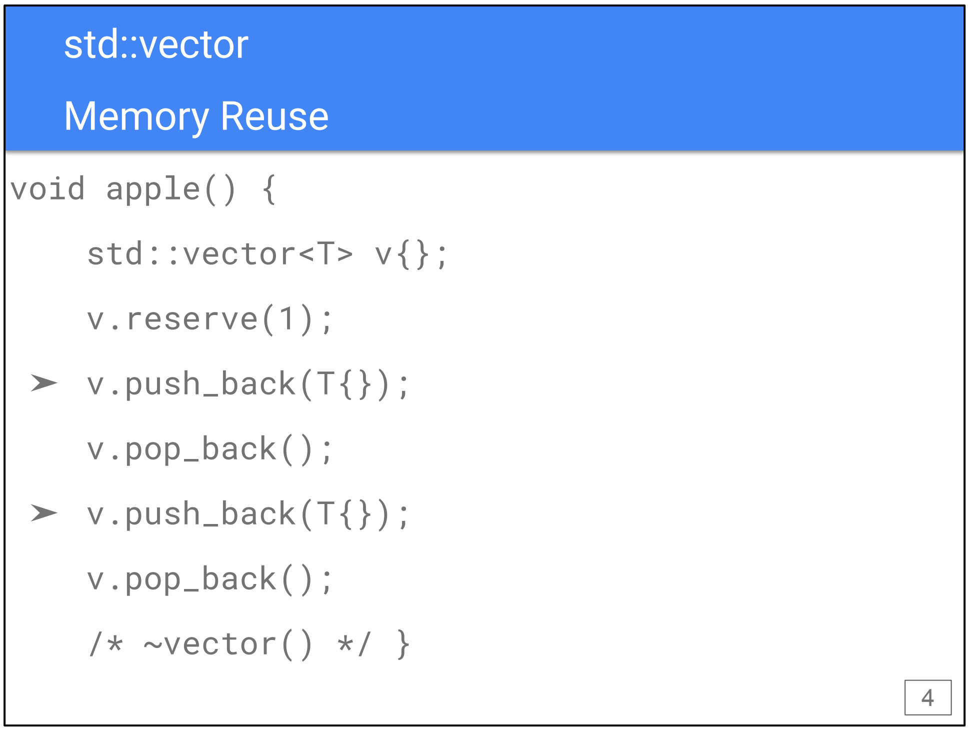
3

So what’s going on here?

We're allocating and deallocating memory.

And we're constructing and destroying a T.

But we're doing those things at different times as separate operations.



std::vector

Memory Reuse

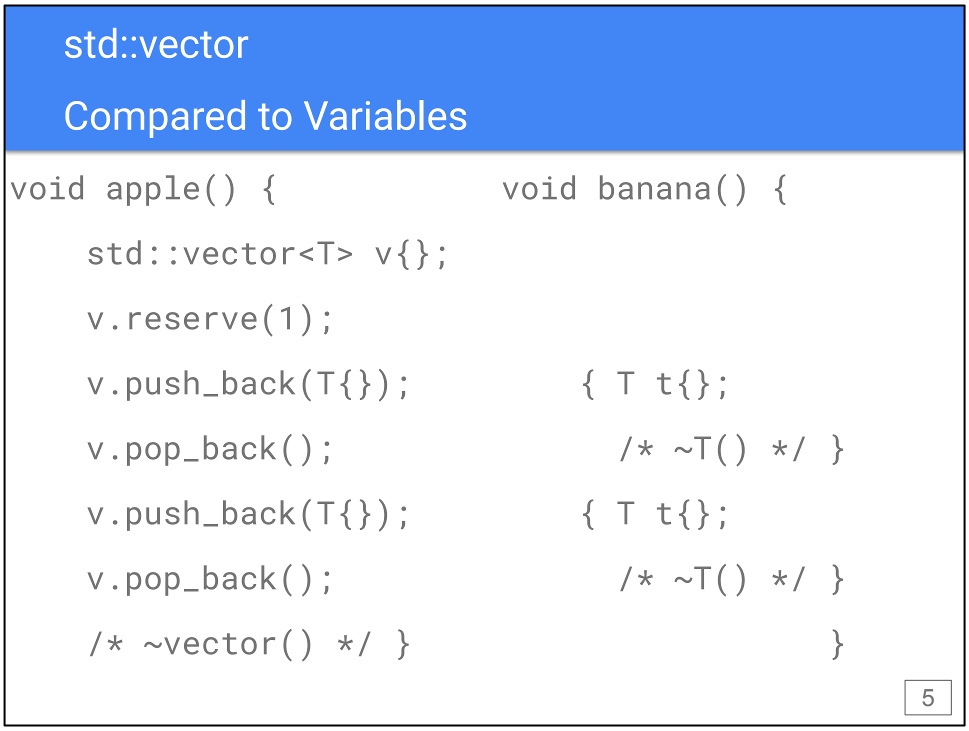
```
void apple() {
    std::vector<T> v{};
    v.reserve(1);
 ➤  v.push_back(T{});
    v.pop_back();
 ➤  v.push_back(T{});
    v.pop_back();
    /* ~vector() */ }
```

4

Using the same memory that we allocated at the top, you can actually construct and destroy MULTIPLE objects.

Of course these objects must exist at different times, you can't have two simultaneously exist at the same address.

But as long as they don't overlap time-wise, you can REUSE the memory that you already have for DIFFERENT instances of T.



std::vector

Compared to Variables

```
void apple() {              void banana() {
    std::vector<T> v{};
    v.reserve(1);
    v.push_back(T{});           { T t{};
    v.pop_back();                 /* ~T() */ }
    v.push_back(T{});           { T t{};
    v.pop_back();                 /* ~T() */ }
    /* ~vector() */ }                        }
```

5

Let’s compare this side by side with a different function.

On the right we have the function banana which behaves very similarly to apple, but it uses local variables in block scopes instead of a container.

Functionally, apple and banana are quite similar.

Sure apple uses heap memory and banana uses stack memory, but both of them are creating and destroying two instances of T, sequentially.

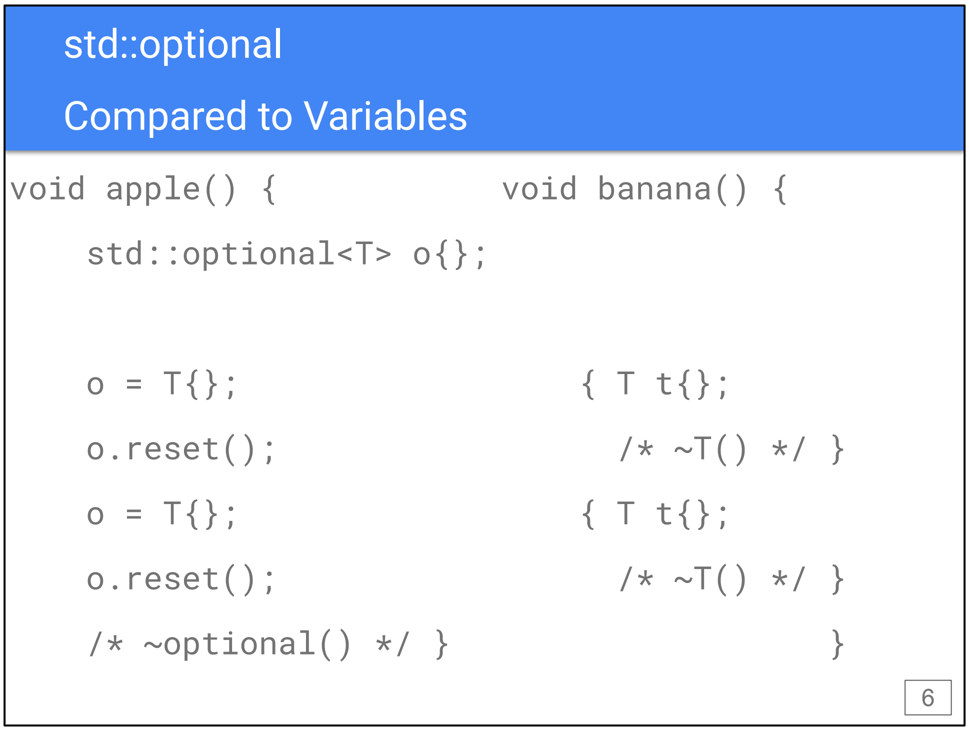
The thing is, apple is re-using its existing memory, while banana is NOT.

Banana, instead, allocates and deallocates stack memory at the same time it constructs and destroys the Ts.

[PAUSE]

So while this comparison roughly holds, there’s a bit too much going on here.

Let’s simplify things.

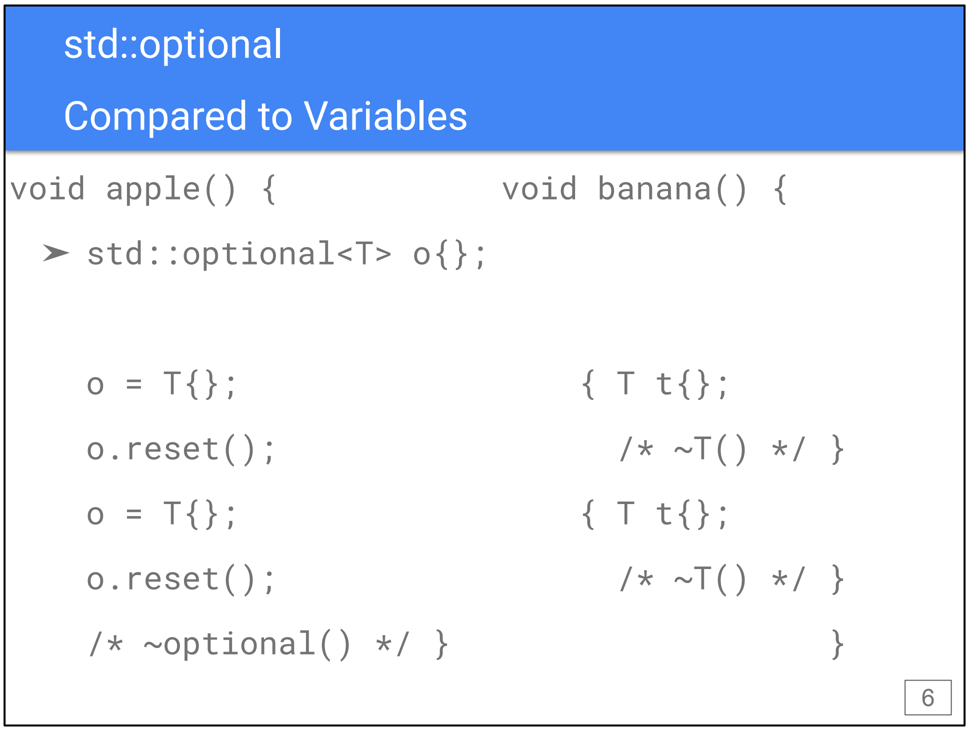


std::optional

Compared to Variables

```
void apple() {              void banana() {
    std::optional<T> o{};

    o = T{};                    { T t{};
    o.reset();                    /* ~T() */ }
    o = T{};                    { T t{};
    o.reset();                    /* ~T() */ }
    /* ~optional() */ }                      }
```

Instead of using std::vector for this behavior, we can use std::optional.



std::optional

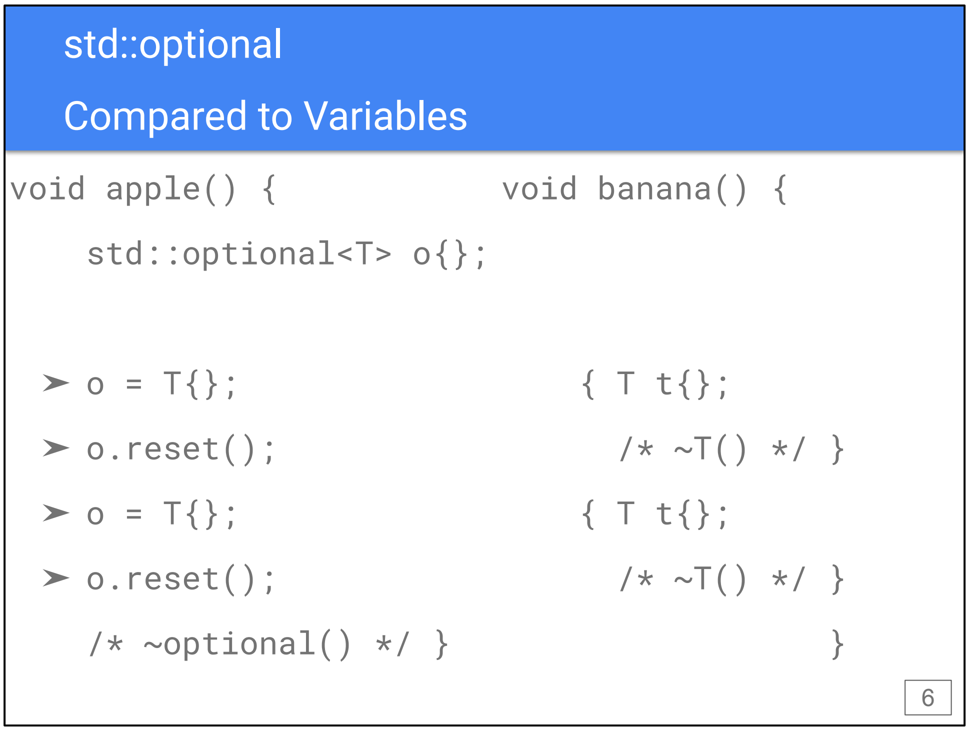
Compared to Variables

```
void apple() {               void banana() {
  ➤ std::optional<T> o{};

    o = T{};                     { T t{};
    o.reset();                     /* ~T() */ }
    o = T{};                     { T t{};
    o.reset();                     /* ~T() */ }
    /* ~optional() */ }                       }
```

6

Much like our call to vector reserve, default constructing a std::optional allocates enough memory for 1 instance of T.

Because std::optional holds its T inside the type, not in a separate allocation, the memory it allocates for T is in fact on the STACK, not on the heap.



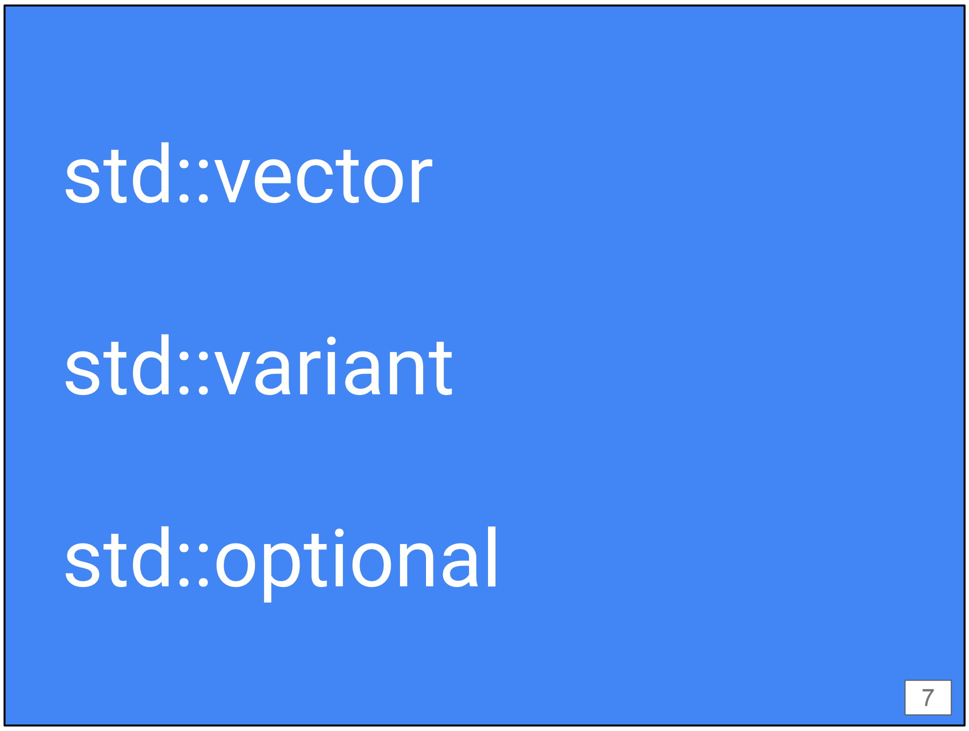
std::optional

Compared to Variables

```
void apple() {               void banana() {
    std::optional<T> o{};

 ➢ o = T{};                      { T t{};
 ➢ o.reset();                      /* ~T() */ }
 ➢ o = T{};                      { T t{};
 ➢ o.reset();                      /* ~T() */ }
    /* ~optional() */ }                         }
```

6

And then similar to push_back and pop_back in vector, assigning a T and then calling reset will construct and destroy a T inside the stack memory that std::optional owns.



std::vector

std::variant

std::optional

7

std::vector, std::variant, std::optional…

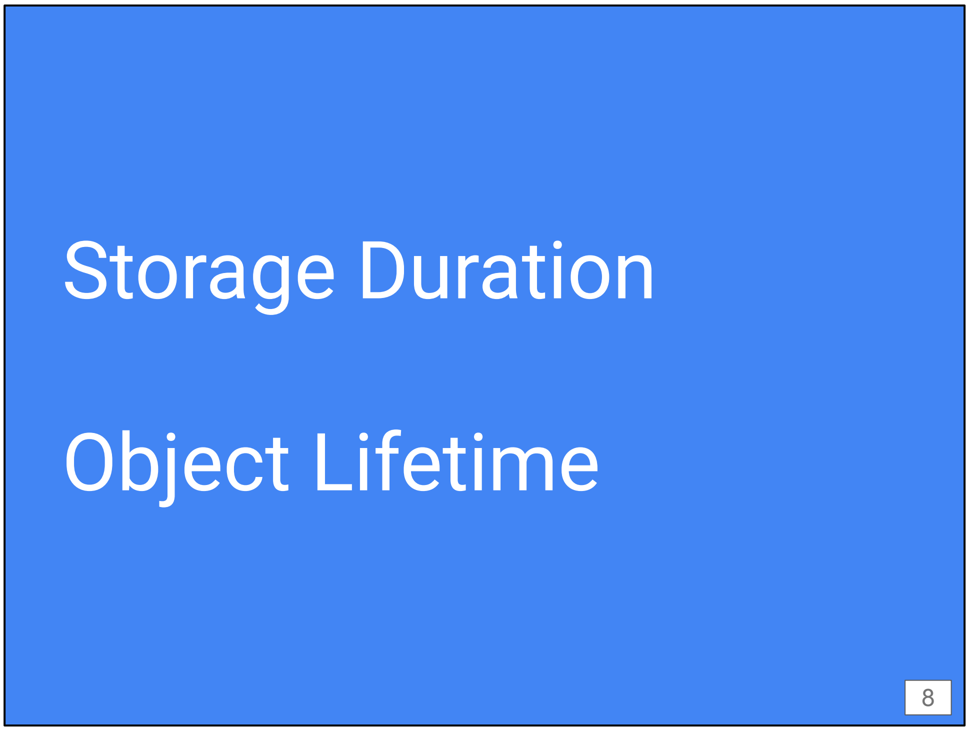
These types can be said to “hold” Ts because they manage memory and elements independently.

The ability to hold Ts is critical to their functionality.

For std::vector, this means growth can be amortized.

For std::optional, this allows for the “empty” state where there is stack memory for a T, but no T exists.

So, how does one implement this behavior?



Storage Duration

Object Lifetime

8

Let’s start by defining some terms: Storage Duration and Object Lifetime.

Object lifetime is the time that an object is alive.

Pretty simple, right?

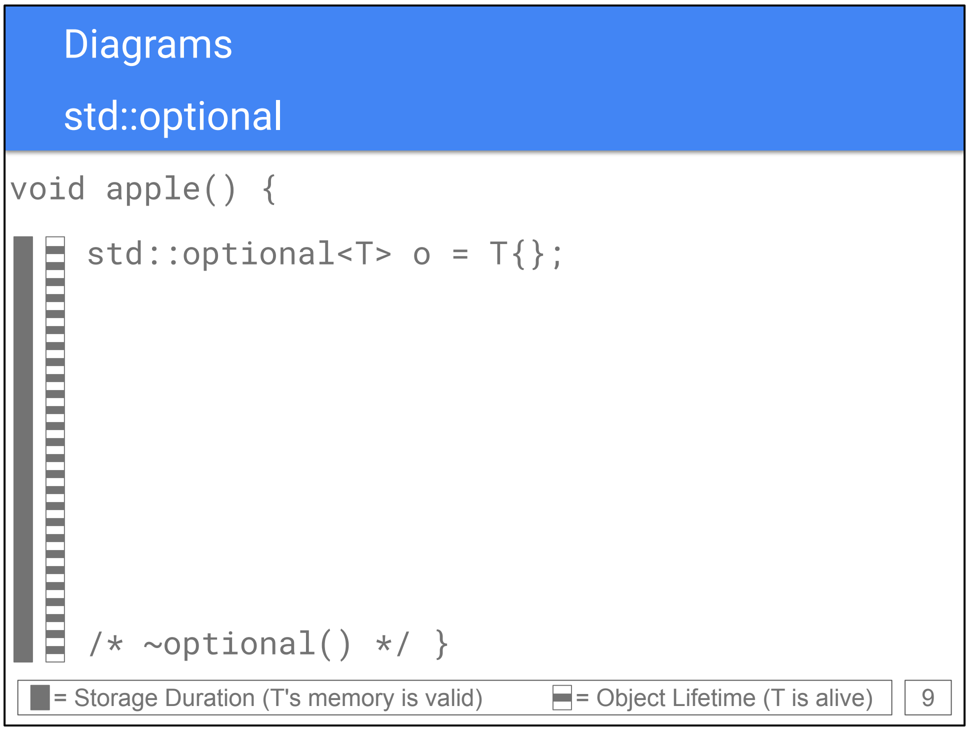
Said another way, the time between an object’s construction and destruction is its object lifetime.

Similarly, storage duration is the time for which the MEMORY for an object is valid.

Or, the time that passes between allocation and deallocation is the storage duration.

[PAUSE]

To really hammer this home, lets diagram these two terms next to some code.



Diagrams

std::optional

```
void apple() {
    std::optional<T> o = T{};







    /* ~optional() */ }
```

■ = Storage Duration (T's memory is valid) ▤ = Object Lifetime (T is alive) 9

Here we have apple again, implemented with std::optional.

To the left of the body, we have two vertical bars.

The left-most bar, the one with a solid background, models the storage duration for T.

From the top of the bar to the bottom of the bar, the memory for T is allocated and valid for use.

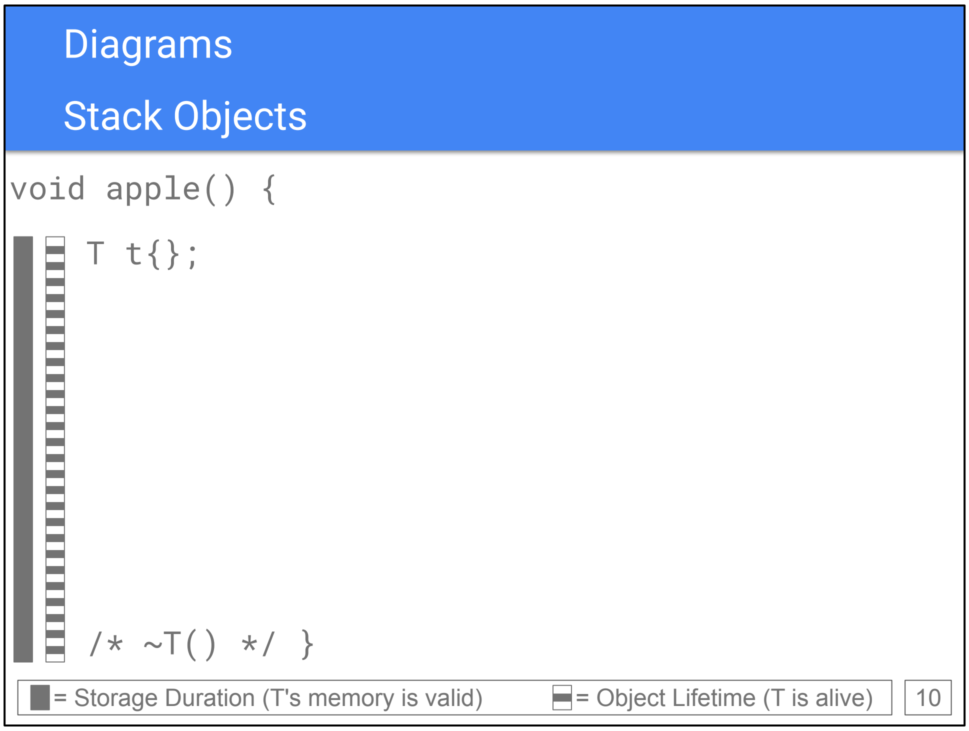
And then the bar to the right of that, the striped bar, we model the object lifetime of T.

If you forget which is which, you can just refer to the key at the bottom of the slide.

So as it happens, since this optional was constructed with a T and was not modified, in this case the storage duration and object lifetime are the same.

The object lifetime begins in the same statement that allocates the memory, and it ends alongside the deallocation.

This is normally how things work in C++.



Diagrams

Stack Objects

```
void apple() {
    T t{};




    /* ~T() */ }
```

■ = Storage Duration (T's memory is valid) ▤ = Object Lifetime (T is alive) 10

By default, the storage duration and object lifetime match one another.

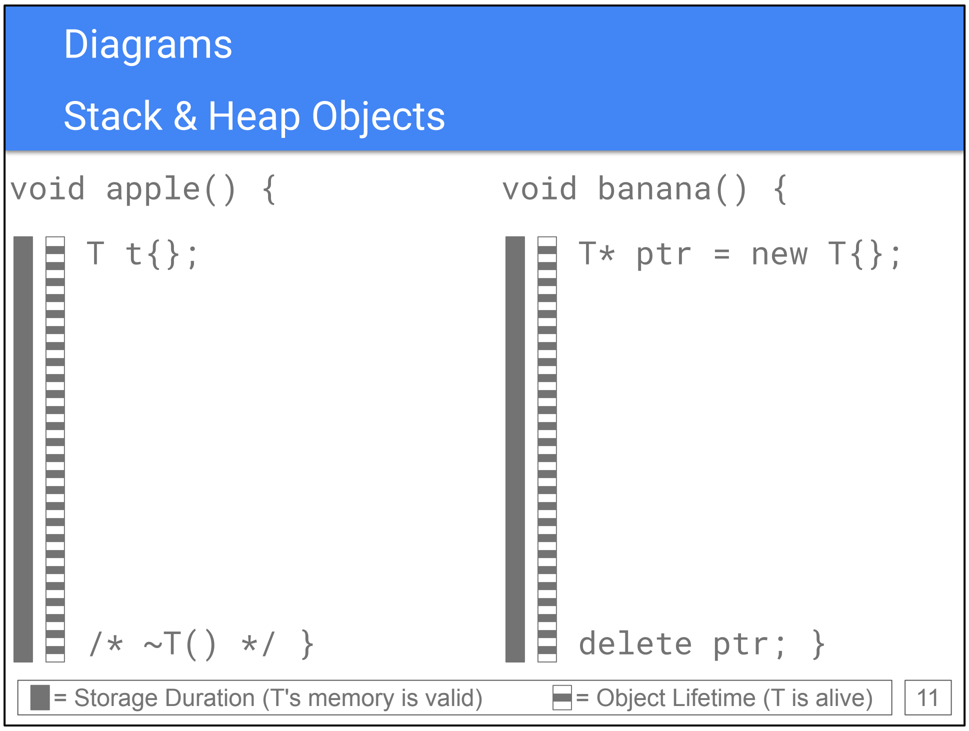
They are “coupled”, you could say.

Here, in apple, we instantiate a local T variable.

Due to the normal C++ rules, at the end of the scope, the T is automatically destroyed.

The stack memory for T is allocated and the T is constructed in the same statement.

And at the end of the function, together, the destructor is called and the memory is deallocated automatically.



Diagrams

Stack & Heap Objects

```
void apple() {                void banana() {
  T t{};                        T* ptr = new T{};

  /* ~T() */ }                  delete ptr; }
```

■ = Storage Duration (T's memory is valid)
▤ = Object Lifetime (T is alive)

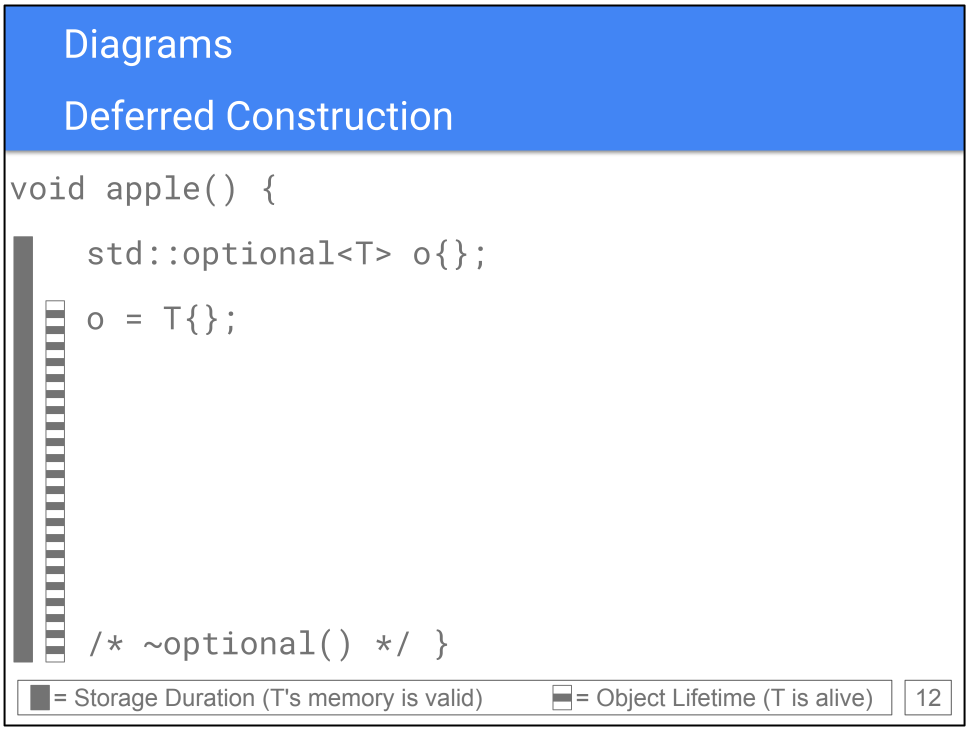
This is ALSO true of heap allocated objects, as is shown here in banana.

When you new-up an instance of T, it allocates memory AND calls T’s constructor.

And then when you delete the pointer, it calls T’s destructor and deallocates the memory.

So to recap, by default, storage duration and object lifetime are coupled together.

[PAUSE]



Diagrams

Deferred Construction

```
void apple() {
    std::optional<T> o{};
    o = T{};

    /* ~optional() */ }
```

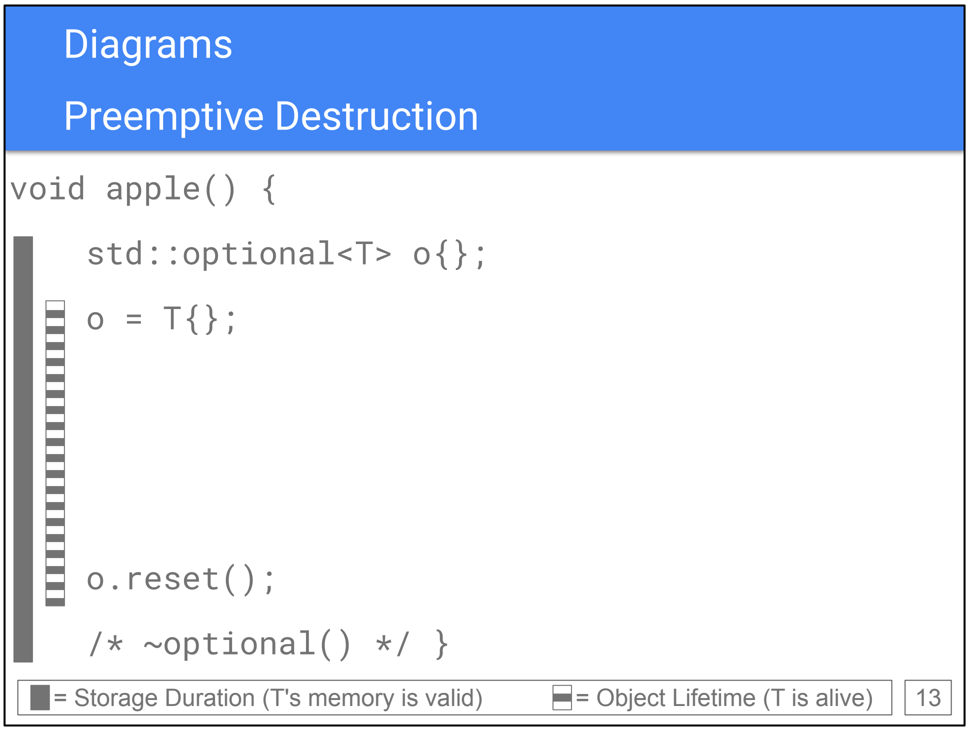
■ = Storage Duration (T's memory is valid) ▤ = Object Lifetime (T is alive) 12

But here we have std::optional again.

Only this time, storage duration and object lifetime are DE-coupled.

See how the vertical bars are different lengths?

That’s because we’re deferring T's constructor until later, after the memory has already been allocated.



Diagrams

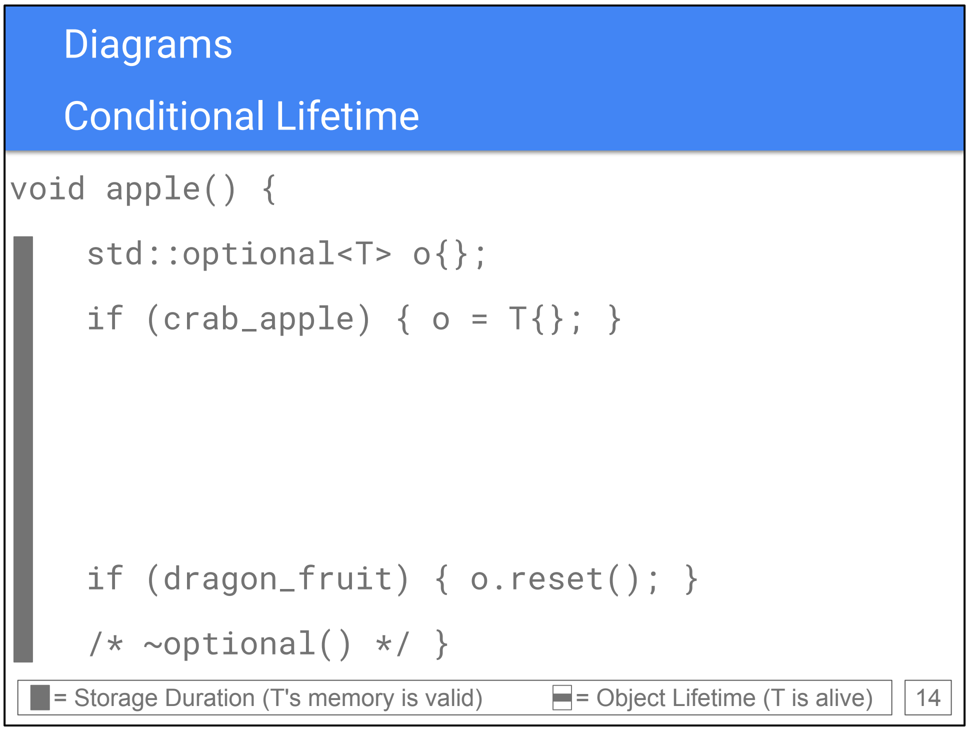
Preemptive Destruction

```
void apple() {
    std::optional<T> o{};
    o = T{};




    o.reset();
    /* ~optional() */ }
```

█ = Storage Duration (T's memory is valid) ▤ = Object Lifetime (T is alive) 13

Similarly, we can shorten the END of the vertical bar, as well.

With a call to reset, we are destroying the T inside the optional, ending its object lifetime IN ADVANCE of the memory being deallocated.



Diagrams

Conditional Lifetime

```
void apple() {
    std::optional<T> o{};
    if (crab_apple) { o = T{}; }




    if (dragon_fruit) { o.reset(); }
    /* ~optional() */ }
```

█ = Storage Duration (T's memory is valid) ▬ = Object Lifetime (T is alive) 14

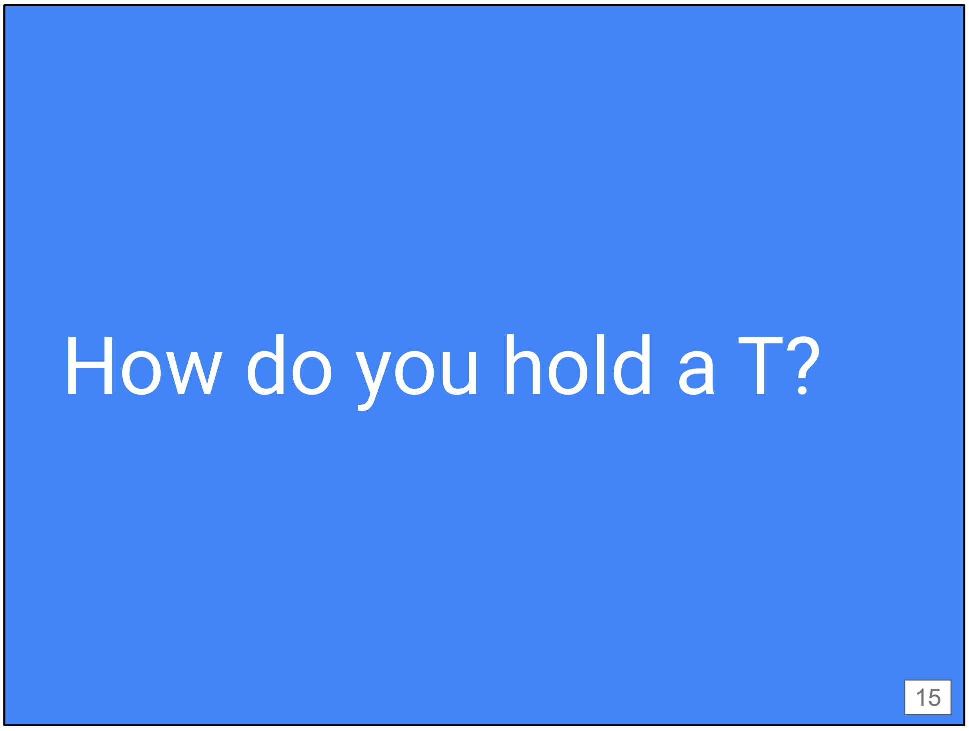
In fact, we can even CONDITIONALLY assign a T into the optional, and CONDITIONALLY reset it, and it works perfectly fine.

I had to remove the object lifetime bar, because there's no way to statically know what the object lifetime of T is, here.

It's fully runtime defined.

All of this is possible because std::optional decouples object lifetime from storage duration.

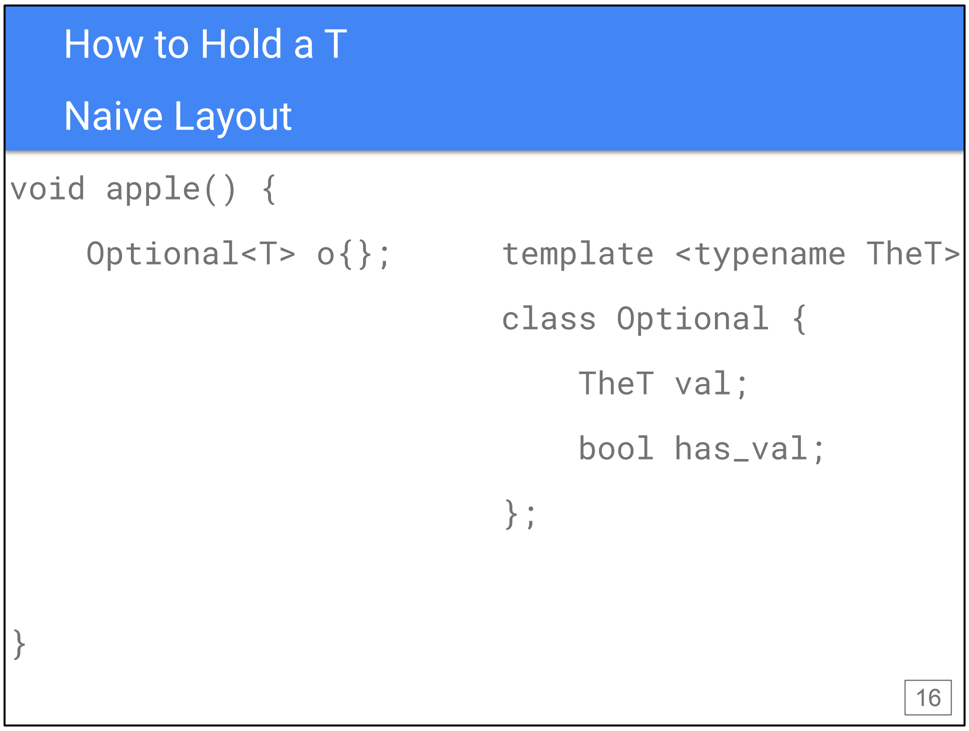
std::optional KNOWS how to hold a T.



How do you hold a T?

So how do you do it?

How do you hold a T?



How to Hold a T

Naive Layout

```
void apple() {
    Optional<T> o{};      template <typename TheT>
                          class Optional {
                              TheT val;
                              bool has_val;
                          };


}
```

Natively, the layout shown here feels like it could work.

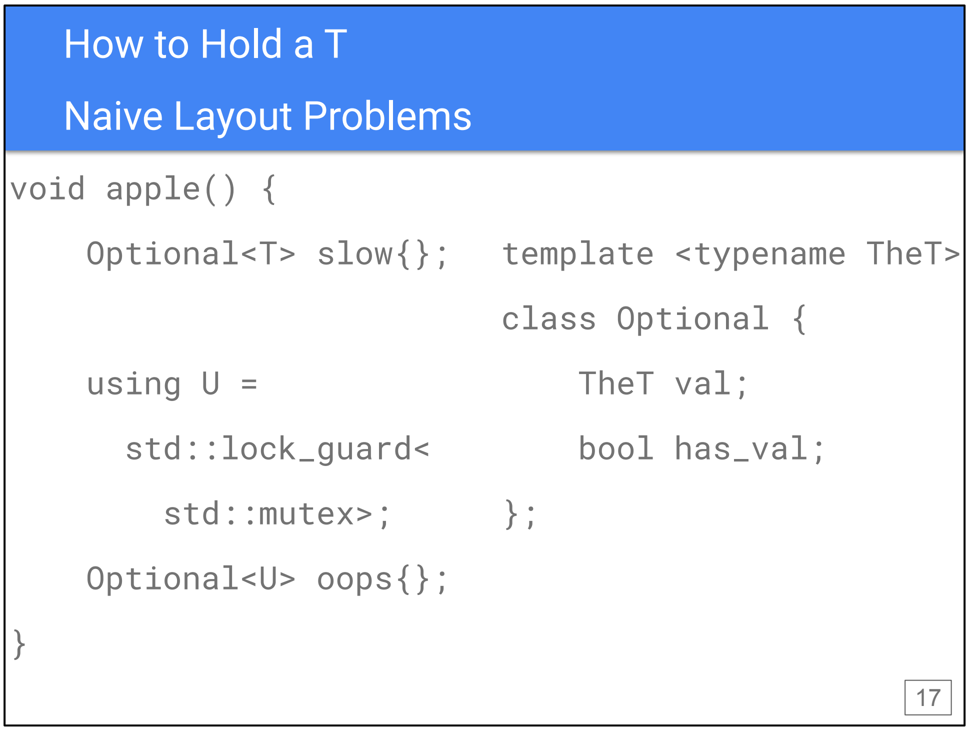
We have a instance of T that exists inside the type and a boolean to track its lifetime.

Obviously there would be much more code involved, this is just showing the data structure, but this has enough to go off of.

We could model the “empty” state as being “if has_val is false, we consider the optional to be empty and ignore the value”

And then we can model the “non-empty” state as being “if has_val is true, we consider the optional to be non-empty and then can read value”

[PAUSE]



How to Hold a T

Naive Layout Problems

```
void apple() {
    Optional<T> slow{};

    using U =
      std::lock_guard<
        std::mutex>;
    Optional<U> oops{};
}
```

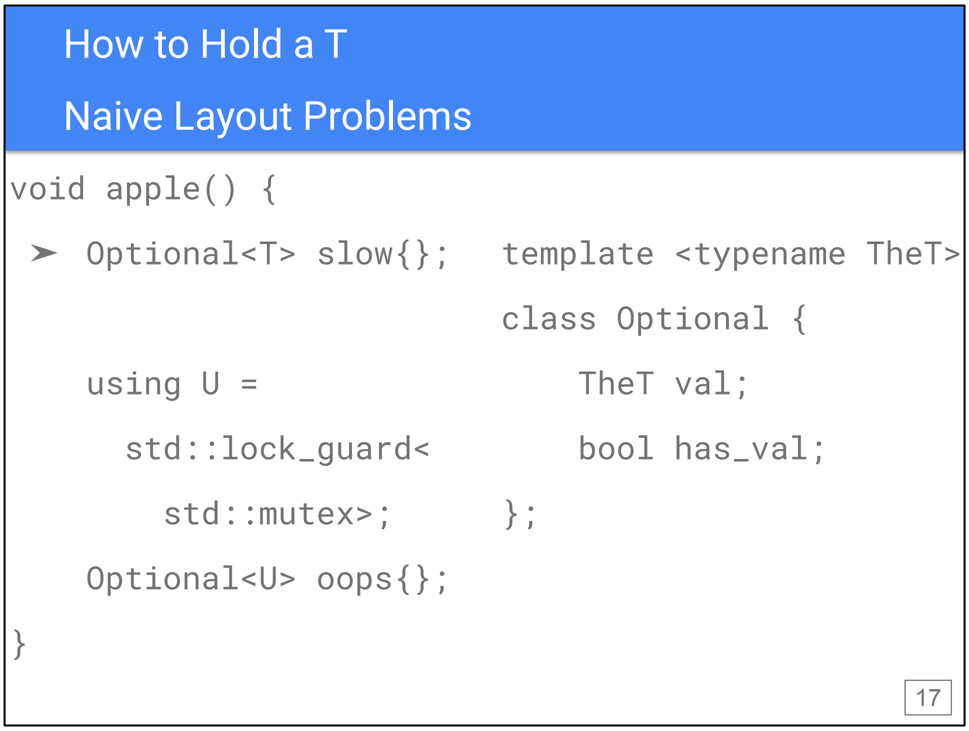
```
template <typename TheT>
class Optional {
    TheT val;
    bool has_val;
};
```

17

Unfortunately, this isn't going to give you what you want.

So first off, using this layout where T is a member of the optional violates "don't pay for what you don't use"

Let's imagine T's default constructor is very expensive to call.



How to Hold a T

Naive Layout Problems

```
void apple() {
 ➤  Optional<T> slow{};   template <typename TheT>
                          class Optional {
    using U =                 TheT val;
      std::lock_guard<        bool has_val;
        std::mutex>;      };
    Optional<U> oops{};
}
```

17

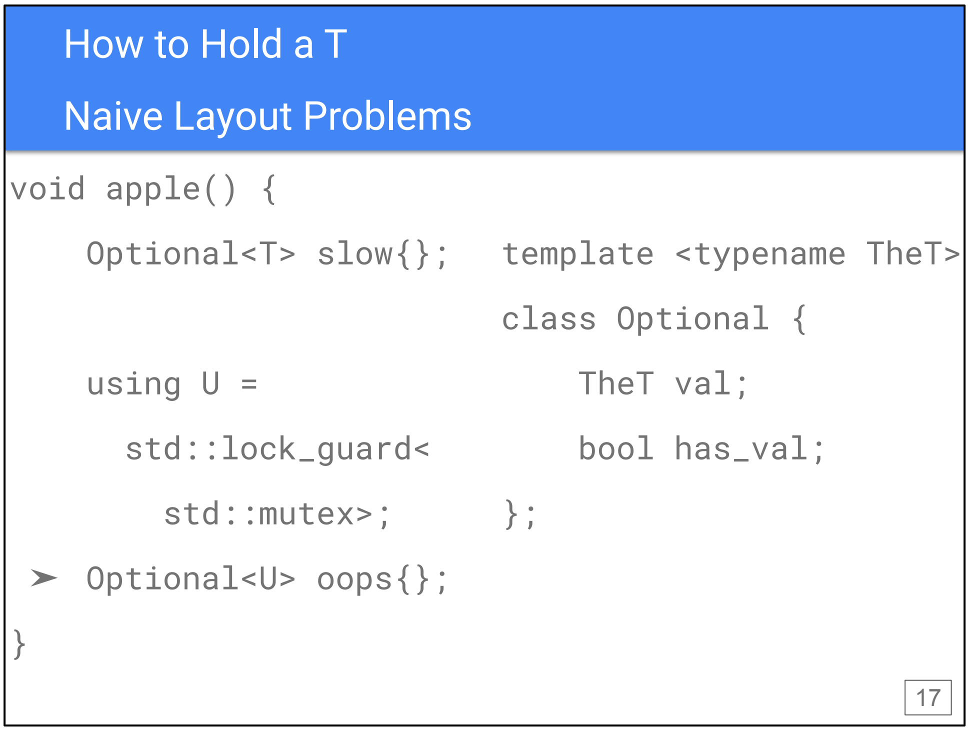
That means that this instance of optional is going to be slow, even though it's empty.

We default constructed the optional meaning it contains no value, but we're STILL paying for the expensive default constructor of T for no reason.

That's not what we want.

But that's fine, we might say that we're willing to pay for the slow constructor anyway.

Even then, this layout is insufficient.



How to Hold a T

Naive Layout Problems

```
void apple() {
    Optional<T> slow{};

    using U =
      std::lock_guard<
        std::mutex>;
 ➤  Optional<U> oops{};
}
```

```
template <typename TheT>
class Optional {
    TheT val;
    bool has_val;
};
```

Because THIS instance of optional won't even compile.

std::lock_guard has no default constructor, meaning with this layout, the default constructor of the optional is ill formed.

Now, am I telling you to make optional lock_guards in your codebase?

No, certainly not.

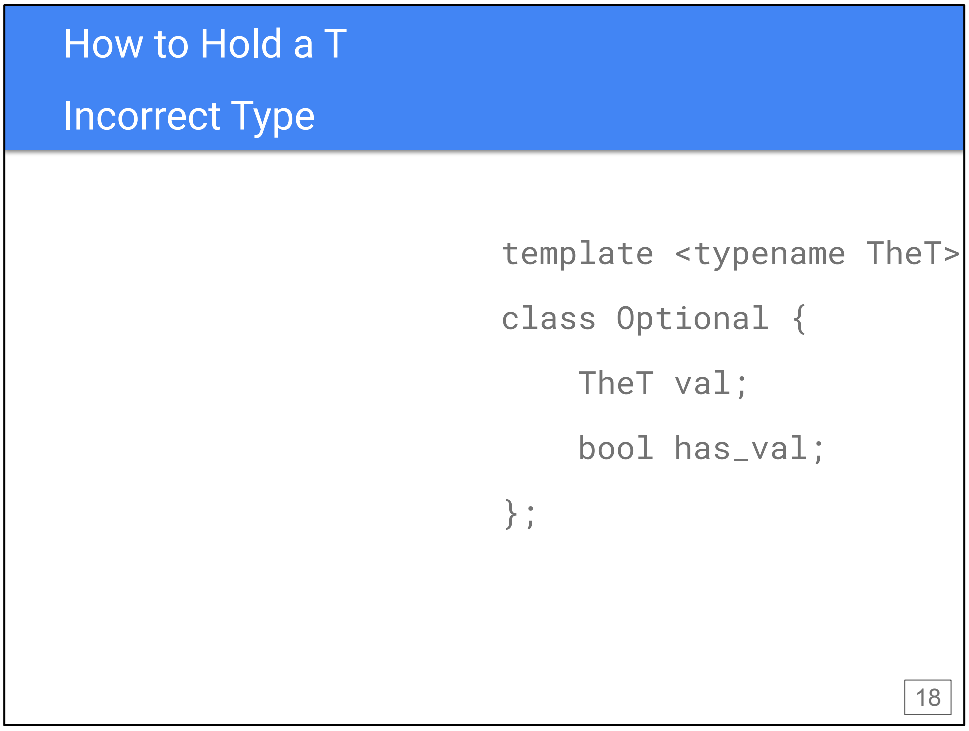
But I'm using it as an example type.

There are types in the world with no default constructor, this is just one example.

We need our implementation of optional to be generic and efficient, and these two example show that it's not living up to that goal.

So then what can we do to make it better?

What's the solution here?



How to Hold a T

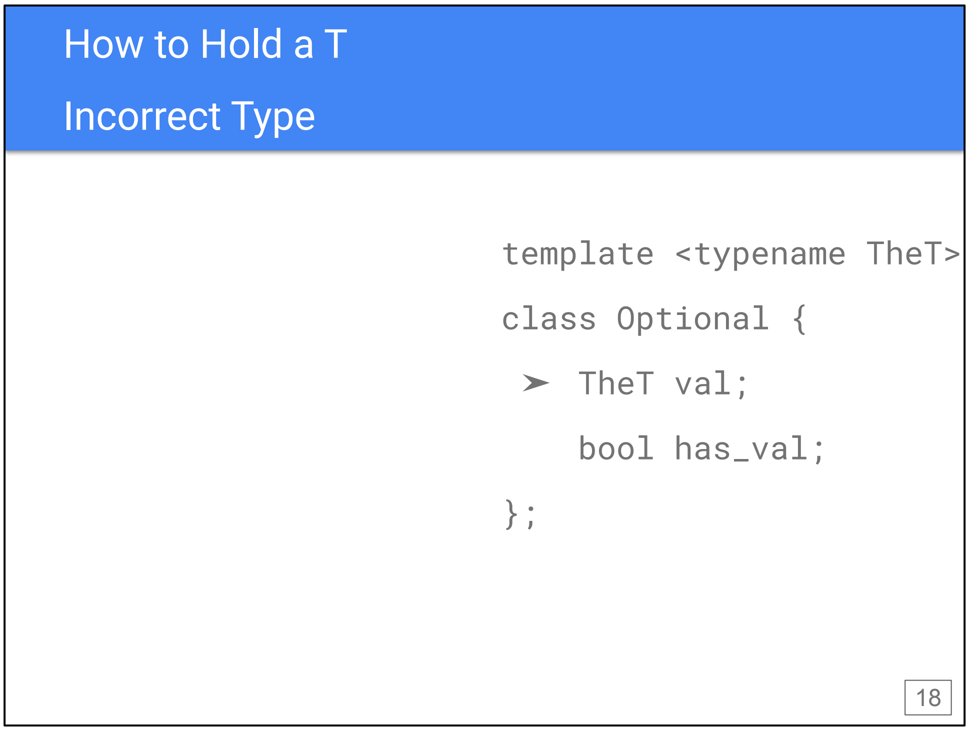
Incorrect Type

```
template <typename TheT>
class Optional {
    TheT val;
    bool has_val;
};
```

18

The reason this implementation fails is that, ...

in the language C++, ...



How to Hold a T

Incorrect Type

```
template <typename TheT>
class Optional {
 ➤  TheT val;
    bool has_val;
};
```

18

T is the WRONG TYPE for holding a T

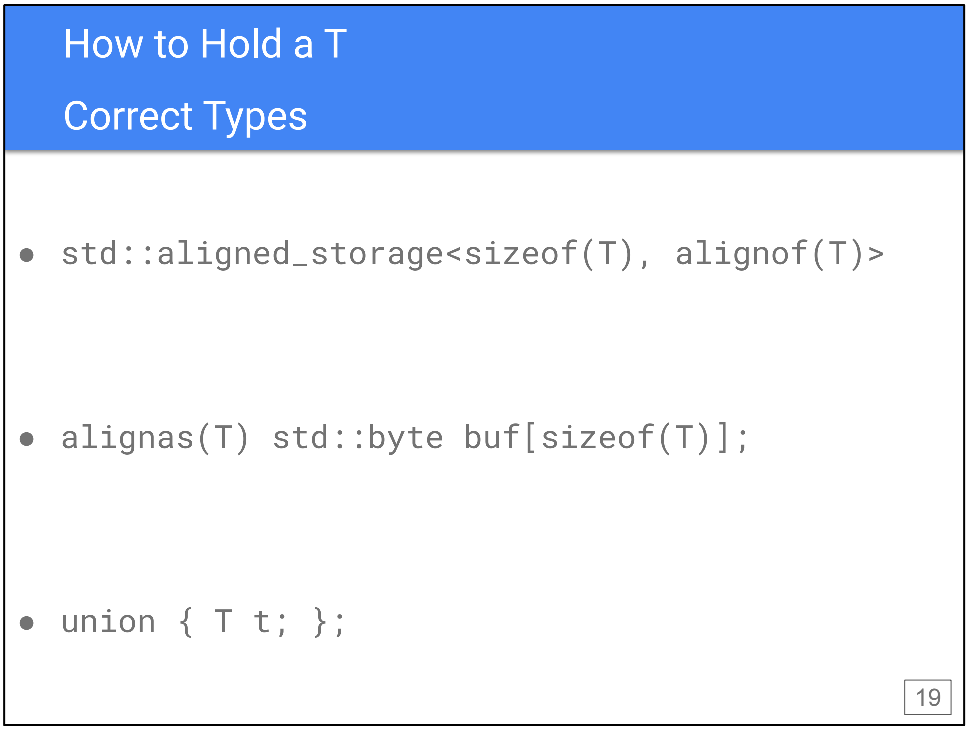
that's right

T is the WRONG TYPE for holding a T

Instead, you need to use some other, unrelated type that is not T but that provides STORAGE for T so that you the programmer can manage the object lifetime yourself.

Pretty wild right?

Let's explore those storage types…



How to Hold a T

Correct Types

- `std::aligned_storage<sizeof(T), alignof(T)>`
- `alignas(T) std::byte buf[sizeof(T)];`
- `union { T t; };`

So here's the set of storage type options we have available to us.

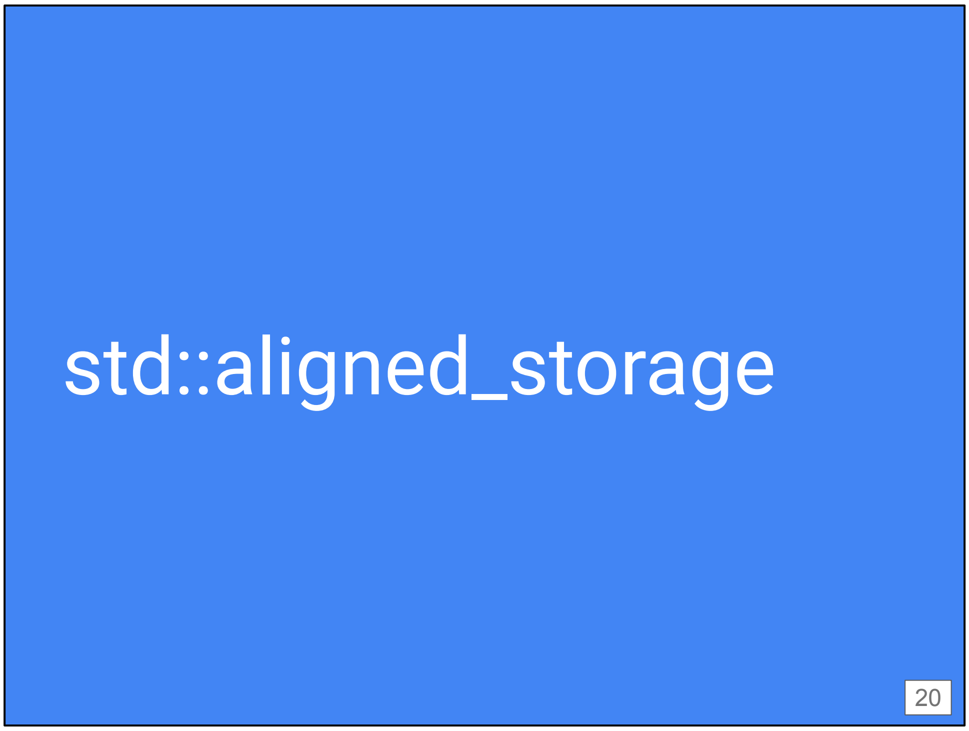
First off there's aligned_storage.

This is actually an entire category of storage types, but aligned_storage is the most common among them.

Following that we have byte arrays.

This is a carryover from C, so you can use std::byte or you can use char or even unsigned char as the underlying type.

And finally we have unions, also a feature from C.



std::aligned_storage

20

Let’s start with aligned_storage.

Now up to this point, this talk has been just information delivery.

This slide changes that.

Just a fair warning, I’m about to include my opinion.

Ready?

Alright!

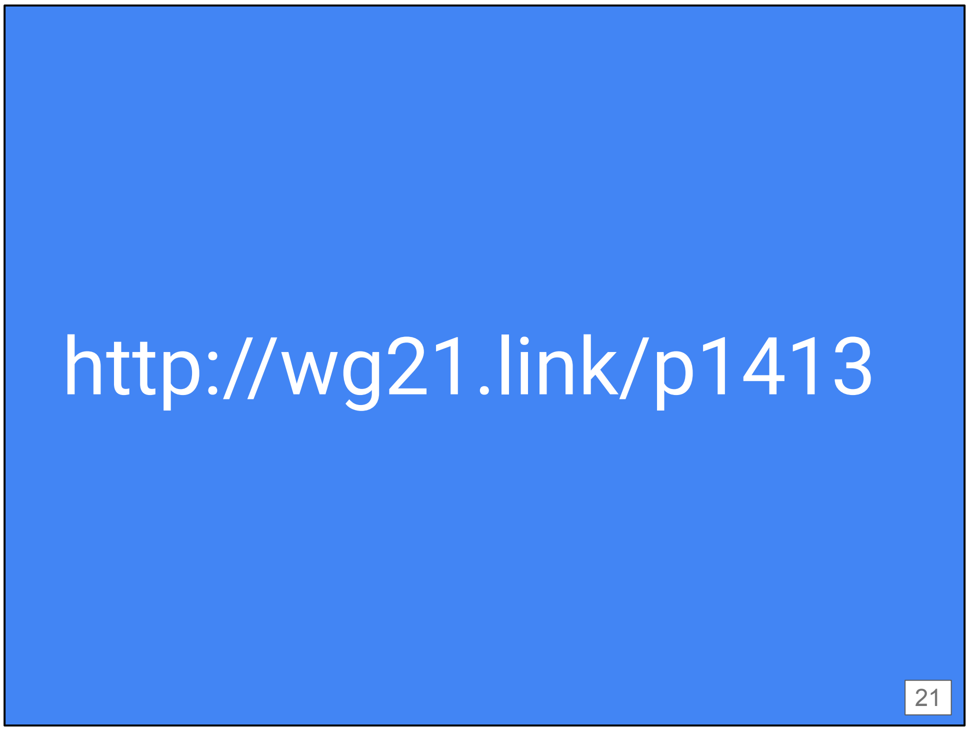
Do not use aligned_storage

at all.

Wow!

That's a strong opinion, right?

Why would I say this?

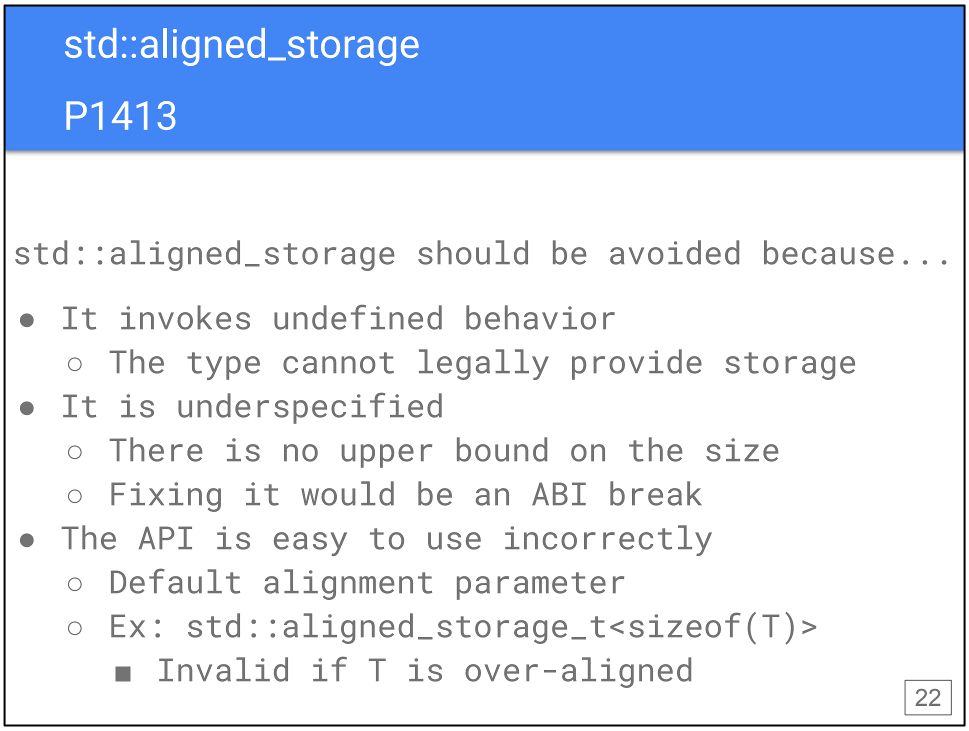


http://wg21.link/p1413

21

For the long winded answer to that question, check out P1413, it’s an ISO C++ standards paper I wrote.

In it I advocate for the complete deprecation of aligned_storage and aligned_union.



std::aligned_storage

P1413

```
std::aligned_storage should be avoided because...

● It invokes undefined behavior
    ○ The type cannot legally provide storage
● It is underspecified
    ○ There is no upper bound on the size
    ○ Fixing it would be an ABI break
● The API is easy to use incorrectly
    ○ Default alignment parameter
    ○ Ex: std::aligned_storage_t<sizeof(T)>
        ■ Invalid if T is over-aligned
```

22

But at a high level, the problems are as follows.

First, it invokes undefined behavior.

Only byte arrays are allowed to provide storage for another type.

Contrary to popular belief, aligned_storage is NOT a typedef for a byte array because that is NOT implementable in the language.

I'll explain why in just a minute.

Instead, it's implemented as a struct type, and so using it invokes UB.

Second, it's underspecified.

The current wording does not set an upper bound for the size of the type, so you could ask it for a type that it of size X and get something much larger.

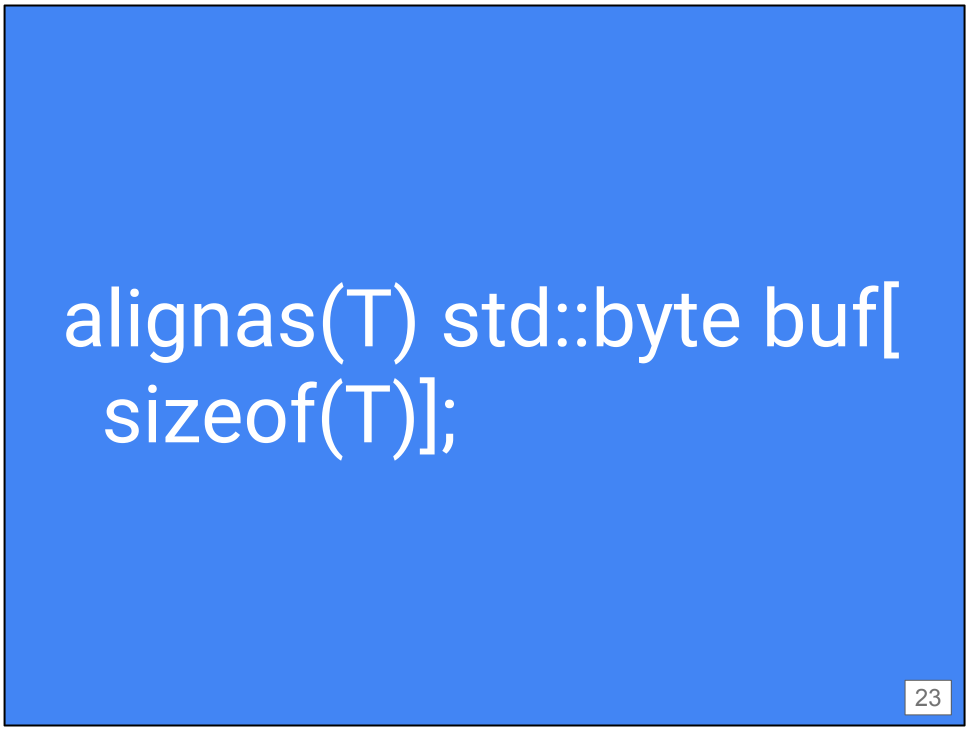
For some implementations, fixing this would be an ABI break, so it's unlikely to change.

And third is that the API is very easy to misuse.

Instead of taking in a type, it takes in size_t template parameters, the second of which has a default value which may or may not be sufficient for your type.

I’ll leave it at that for now, but if you’re curious to learn more, read P1413.

Alright, let’s get back out of opinion mode and back into information delivery mode



alignas(T) std::byte buf[sizeof(T)];

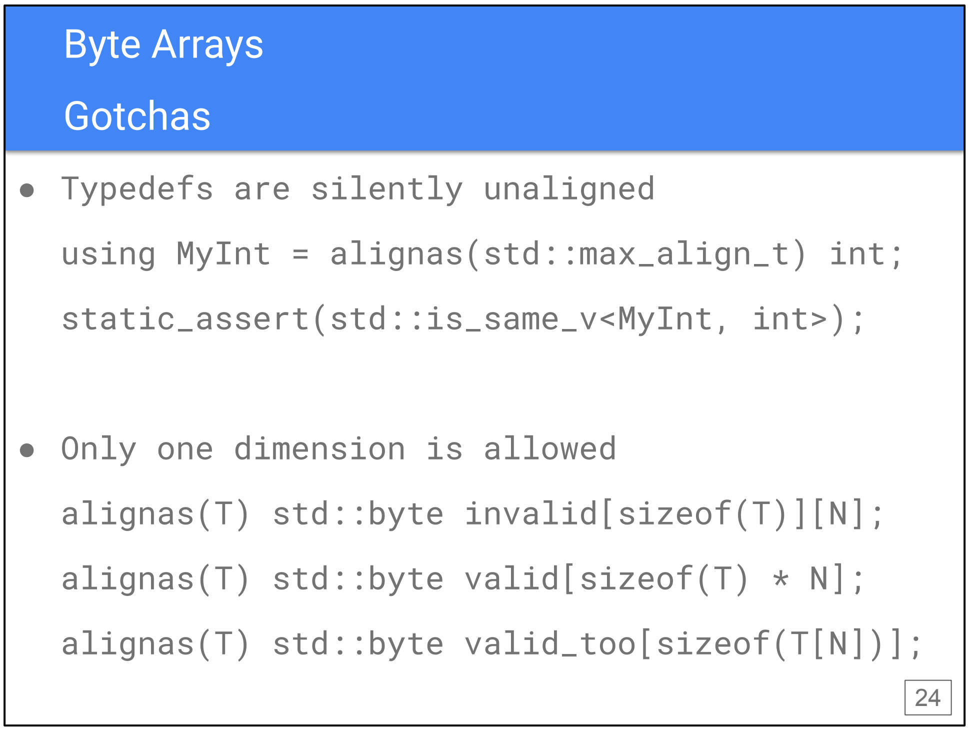
So that we can talk about byte arrays!

Yes, what’s old is new again.

Character buffers are as relevant today as ever.

So for the most part, these will do what you expect them to.

There’s just a couple gotchas I want to highlight.



Byte Arrays

Gotchas

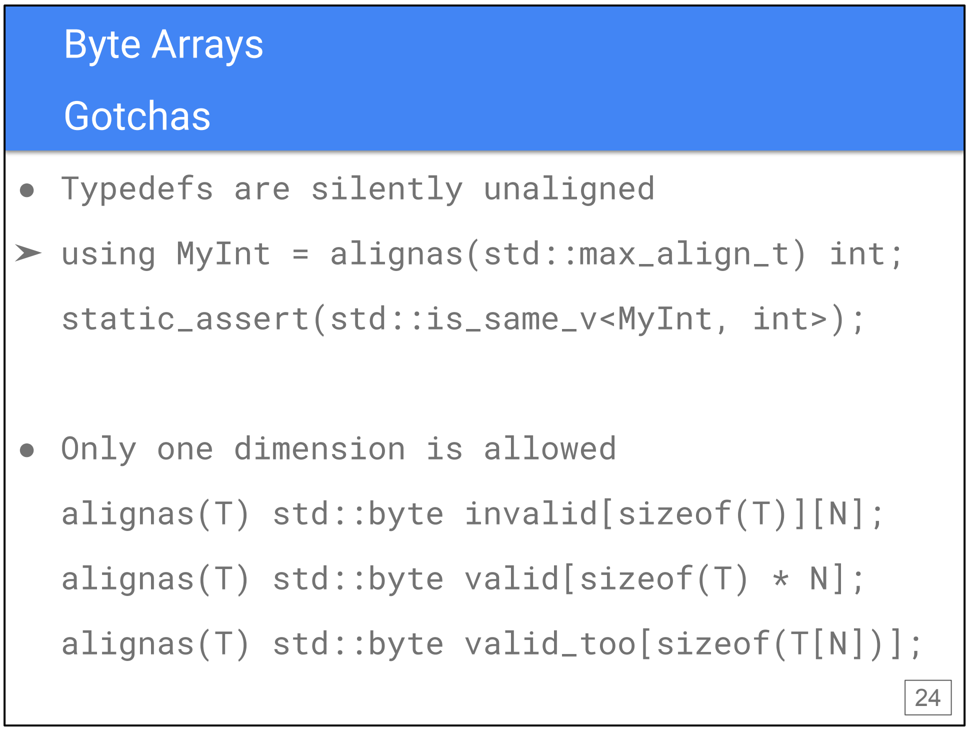
- Typedefs are silently unaligned

```
using MyInt = alignas(std::max_align_t) int;
static_assert(std::is_same_v<MyInt, int>);
```

- Only one dimension is allowed

```
alignas(T) std::byte invalid[sizeof(T)][N];
alignas(T) std::byte valid[sizeof(T) * N];
alignas(T) std::byte valid_too[sizeof(T[N])];
```

First is the interaction between alignment and typedefs.


Byte Arrays

Gotchas

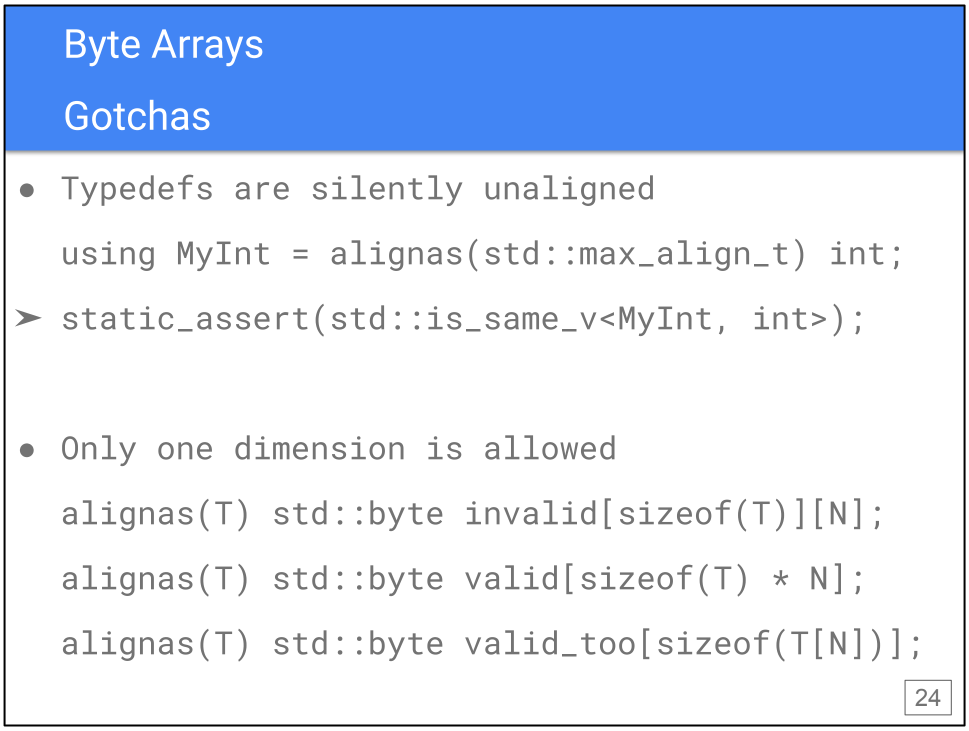
- Typedefs are silently unaligned

➢
```
using MyInt = alignas(std::max_align_t) int;
static_assert(std::is_same_v<MyInt, int>);
```

- Only one dimension is allowed

```
alignas(T) std::byte invalid[sizeof(T)][N];
alignas(T) std::byte valid[sizeof(T) * N];
alignas(T) std::byte valid_too[sizeof(T[N])];
```

In the first bullet there’s a typedef called MyInt which has an alignment parameter specified.



Byte Arrays

Gotchas

- Typedefs are silently unaligned

```
using MyInt = alignas(std::max_align_t) int;
```

➢ `static_assert(std::is_same_v<MyInt, int>);`

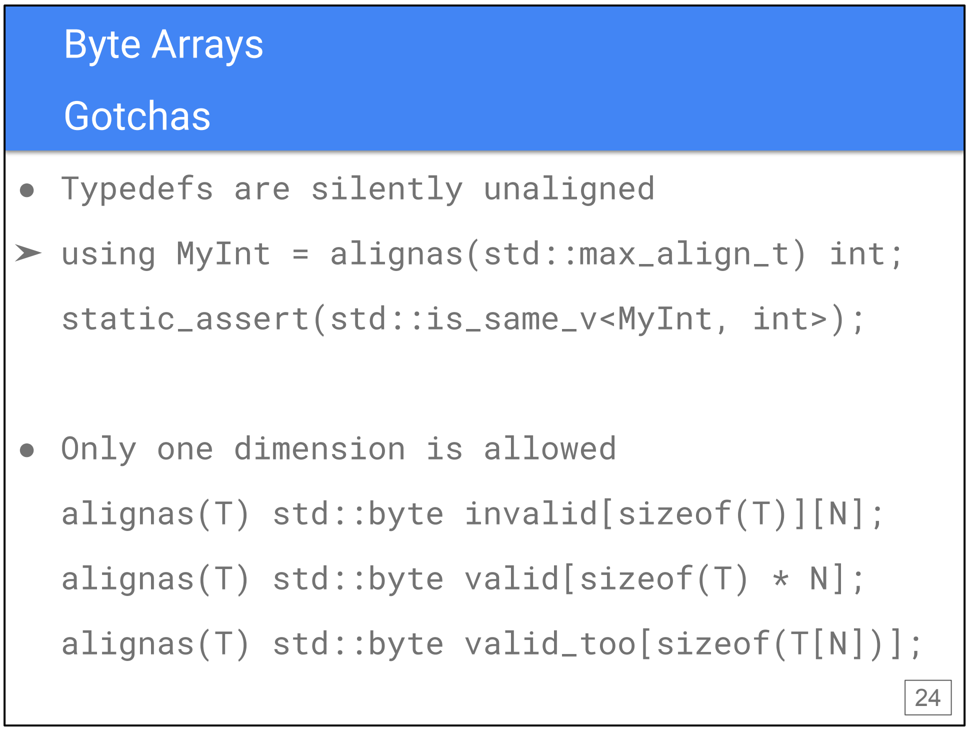
- Only one dimension is allowed

```
alignas(T) std::byte invalid[sizeof(T)][N];
alignas(T) std::byte valid[sizeof(T) * N];
alignas(T) std::byte valid_too[sizeof(T[N])];
```

24

But on the line below that, I'm static asserting that MyInt is the same type as int.

This compiles just fine as it is, because the alignment parameter isn't actually getting applied to MyInt at all.



Byte Arrays

Gotchas

- Typedefs are silently unaligned

```
➢ using MyInt = alignas(std::max_align_t) int;
  static_assert(std::is_same_v<MyInt, int>);
```

- Only one dimension is allowed

```
alignas(T) std::byte invalid[sizeof(T)][N];
alignas(T) std::byte valid[sizeof(T) * N];
alignas(T) std::byte valid_too[sizeof(T[N])];
```

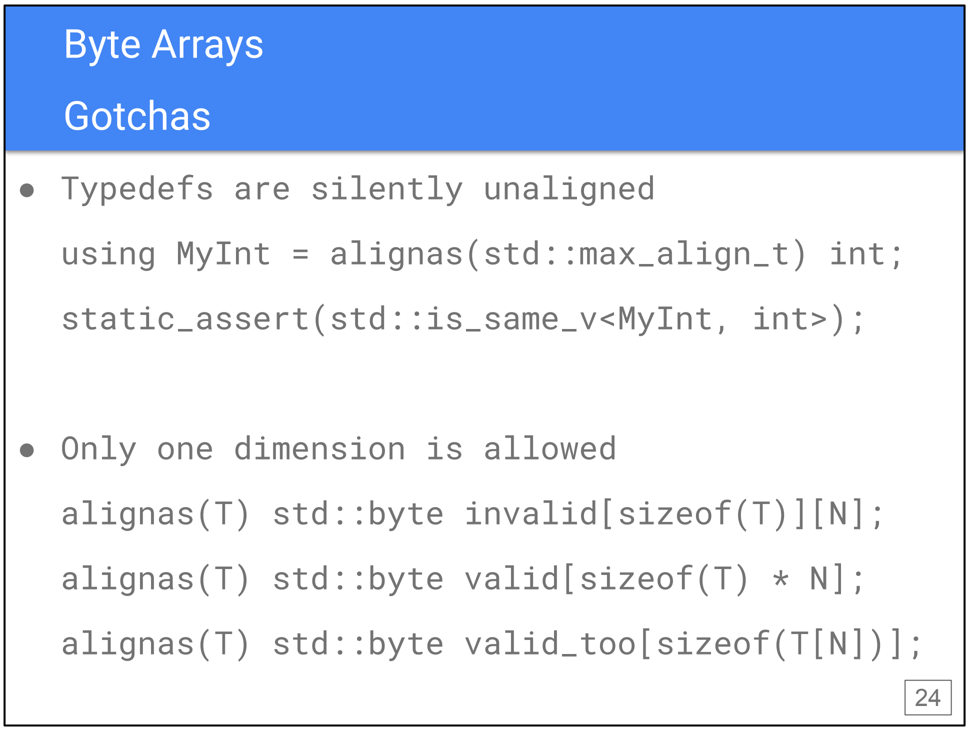
24

For some reason, this syntax is allowed, but the alignment is completely ignored.

So even if you specify it, it actually has no effect on the typedef at all.

THIS is why aligned_storage is implemented as a struct instead of a typedef.

There’s simply NO way to align typedefs like this, even though it appears to work.



Byte Arrays

Gotchas

- Typedefs are silently unaligned

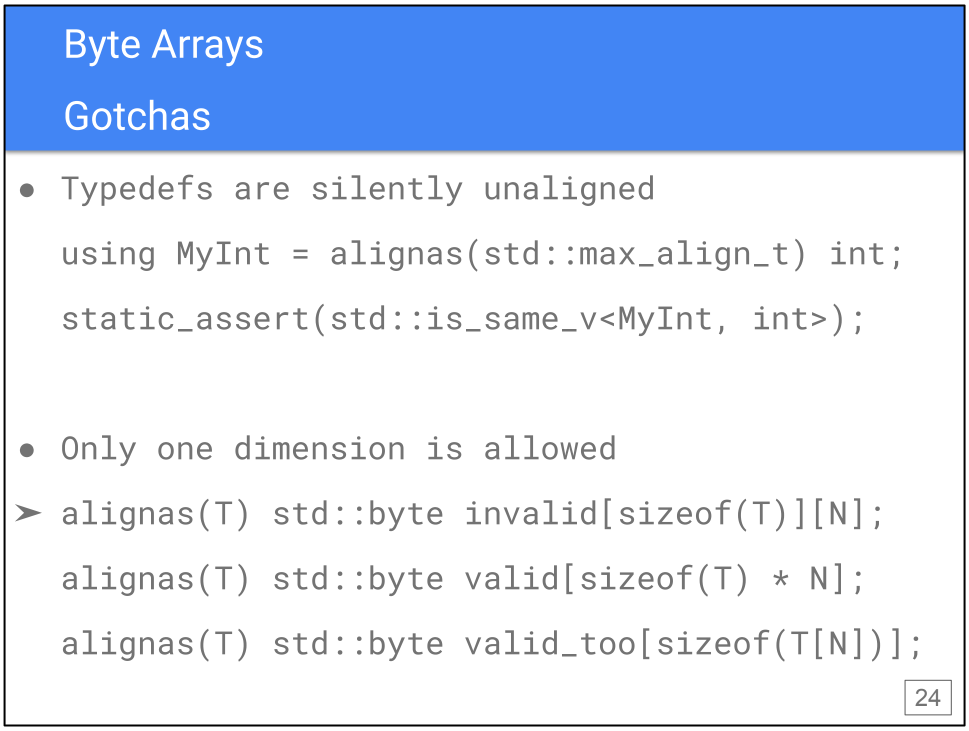
```
using MyInt = alignas(std::max_align_t) int;
static_assert(std::is_same_v<MyInt, int>);
```

- Only one dimension is allowed

```
alignas(T) std::byte invalid[sizeof(T)][N];
alignas(T) std::byte valid[sizeof(T) * N];
alignas(T) std::byte valid_too[sizeof(T[N])];
```

The other problem is around dimensionality.

Only arrays of char, unsigned char and std::byte are legally allowed to provide storage.



Byte Arrays

Gotchas

- Typedefs are silently unaligned

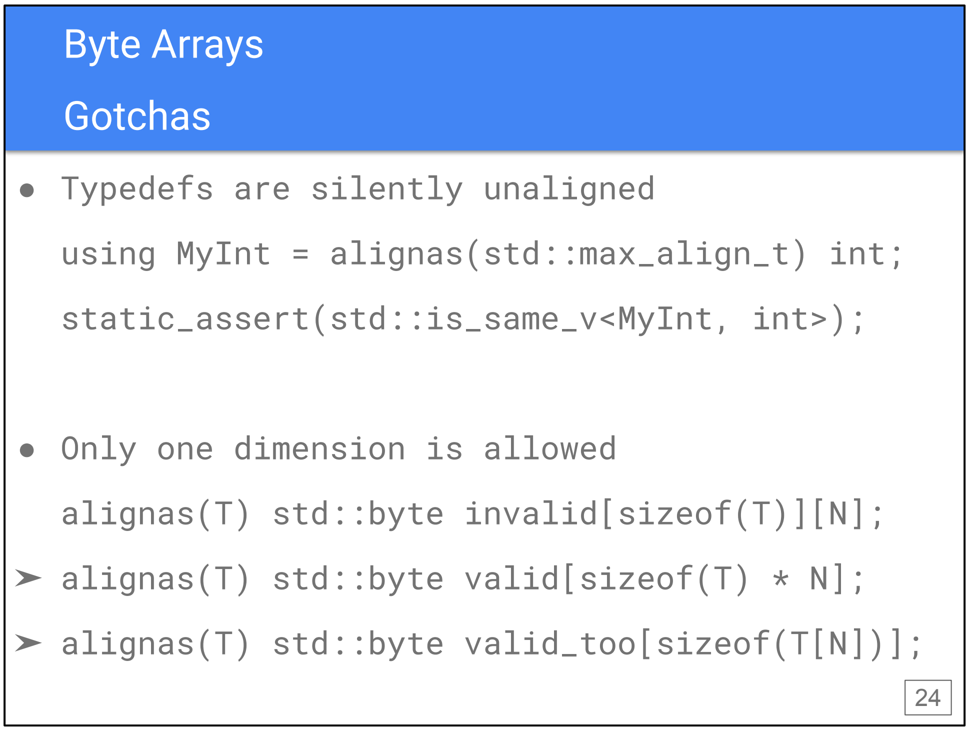
```
using MyInt = alignas(std::max_align_t) int;
static_assert(std::is_same_v<MyInt, int>);
```

- Only one dimension is allowed

```
➢ alignas(T) std::byte invalid[sizeof(T)][N];
  alignas(T) std::byte valid[sizeof(T) * N];
  alignas(T) std::byte valid_too[sizeof(T[N])];
```

Not found in that list are arrays of arrays.

So if you need to hold N contiguous Ts in memory, you can’t do that with an array of storage arrays, such as the one shown here.



Byte Arrays

Gotchas

- Typedefs are silently unaligned

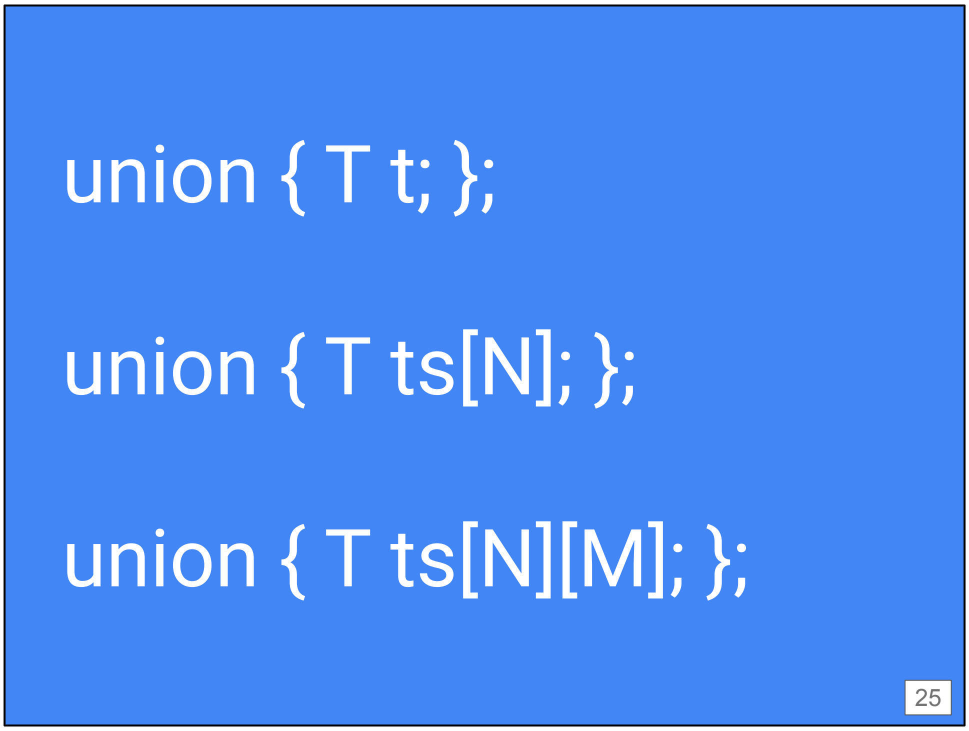
```
using MyInt = alignas(std::max_align_t) int;
static_assert(std::is_same_v<MyInt, int>);
```

- Only one dimension is allowed

```
  alignas(T) std::byte invalid[sizeof(T)][N];
➢ alignas(T) std::byte valid[sizeof(T) * N];
➢ alignas(T) std::byte valid_too[sizeof(T[N])];
```

Instead, you need to compute the entire array's size and pass that into the top level dimension.

“valid” and “valid_too” are both ways of computing the size of the storage type, and you're free to choose whichever one you prefer.



```
union { T t; };

union { T ts[N]; };

union { T ts[N][M]; };
```

25

Finally we have unions!

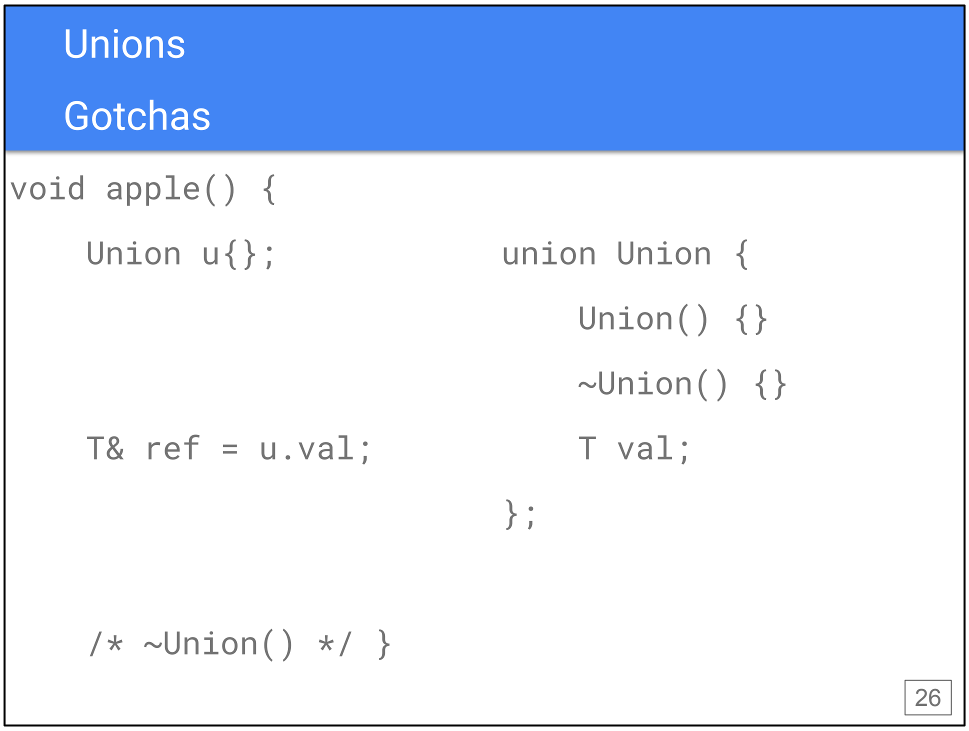
These have a lot of advantages over the other storage types.

First off, unions are aligned and sized correctly by default.

You don’t need to specify alignment or size anywhere, because it just does the right thing.

Additionally, if you need multiple contiguous Ts, you can use normal array syntax.

As long as the array is a top-level, non-static member of the union, that union is sufficient for holding N Ts, giving every element independent object lifetimes.



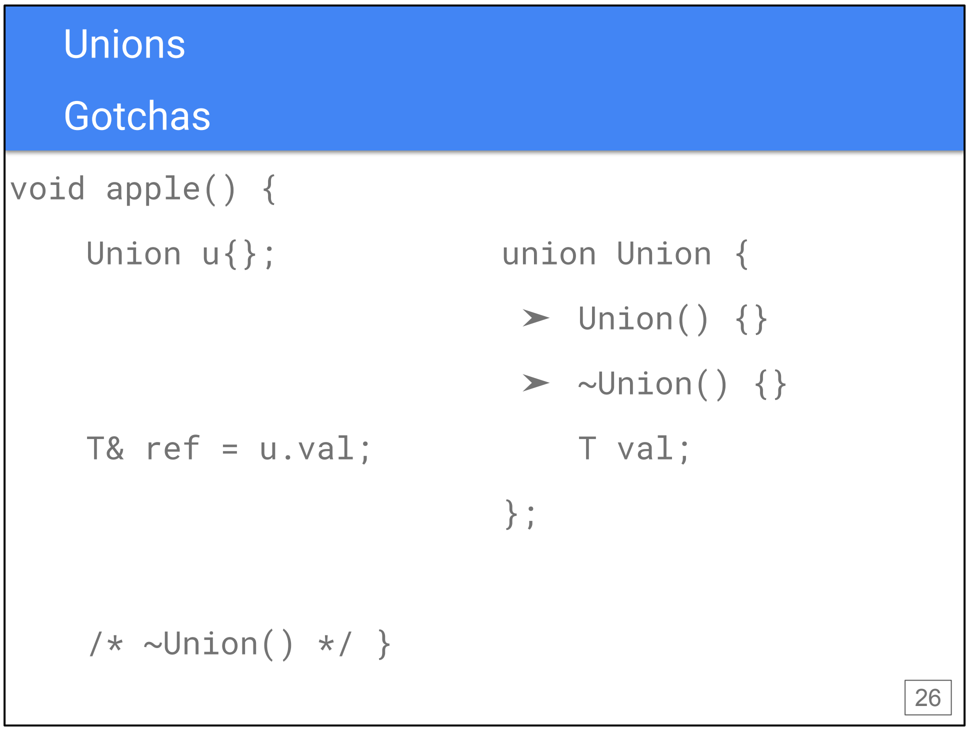
Unions

Gotchas

```
void apple() {
    Union u{};               union Union {
                                 Union() {}
                                 ~Union() {}
    T& ref = u.val;              T val;
                             };


    /* ~Union() */ }
```

26

There’s just one gotcha that I think is worth mentioning.



Unions

Gotchas

```
void apple() {
    Union u{};                union Union {
                               ➢ Union() {}
                               ➢ ~Union() {}
    T& ref = u.val;                T val;
                              };


    /* ~Union() */ }
```

26

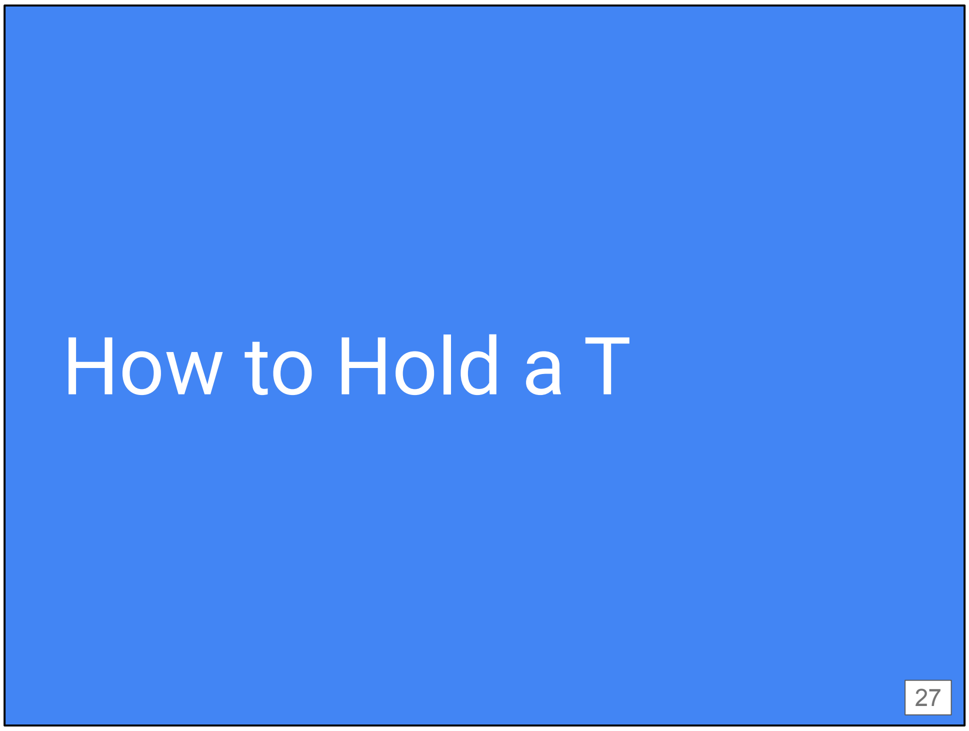
In order to account for all cases, your union types must be written in this way.

Depending on what type T is, the compiler may not be able to default the special member functions for the union.

So it is your responsibility to manually implement them, DELIBERATELY leaving value uninitialized.

As you can see here, the constructor and destructor of MyUnion have no content whatsoever.

That’s what you need to do in your unions as well, to account for all Ts.


How to Hold a T

27

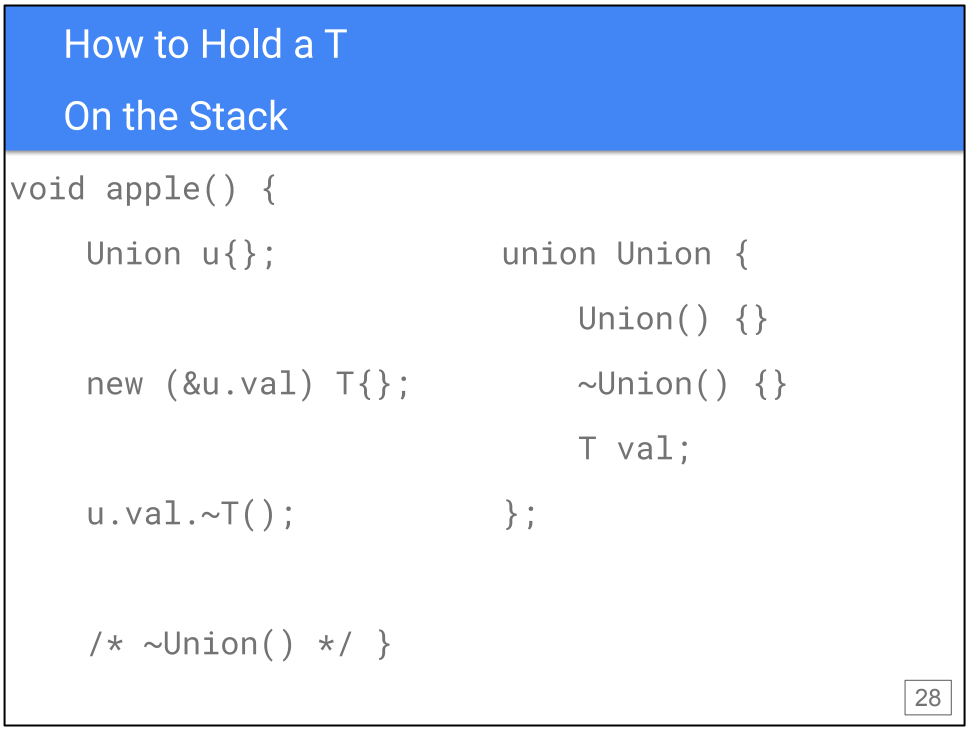
Alright!

We’ve made it this far!

Congratudolences.

But the title of this talk is “How to Hold a T”.

To round everything out, to really bring it home, let’s take what we’ve learned and do EXACTLY that.



How to Hold a T

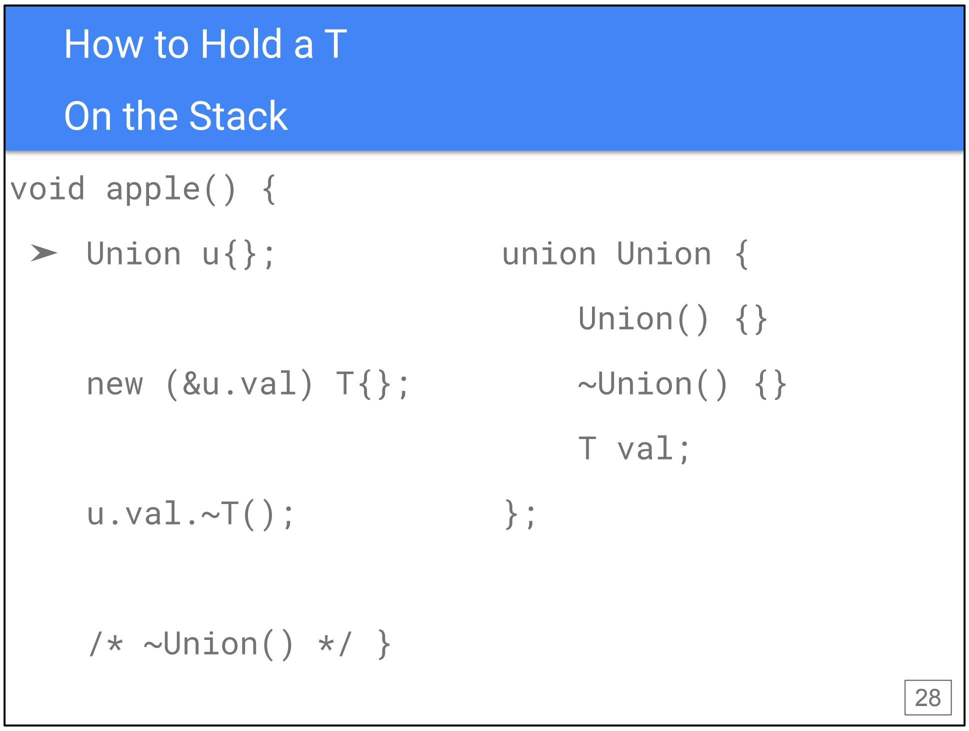
On the Stack

```
void apple() {
    Union u{};               union Union {
                                 Union() {}
    new (&u.val) T{};            ~Union() {}
                                 T val;
    u.val.~T();              };

    /* ~Union() */ }
```

28

Here we have the same union that we can use for making T storage types.

And inside apple, we're using it to decouple storage duration and object lifetime, just like with std::optional.



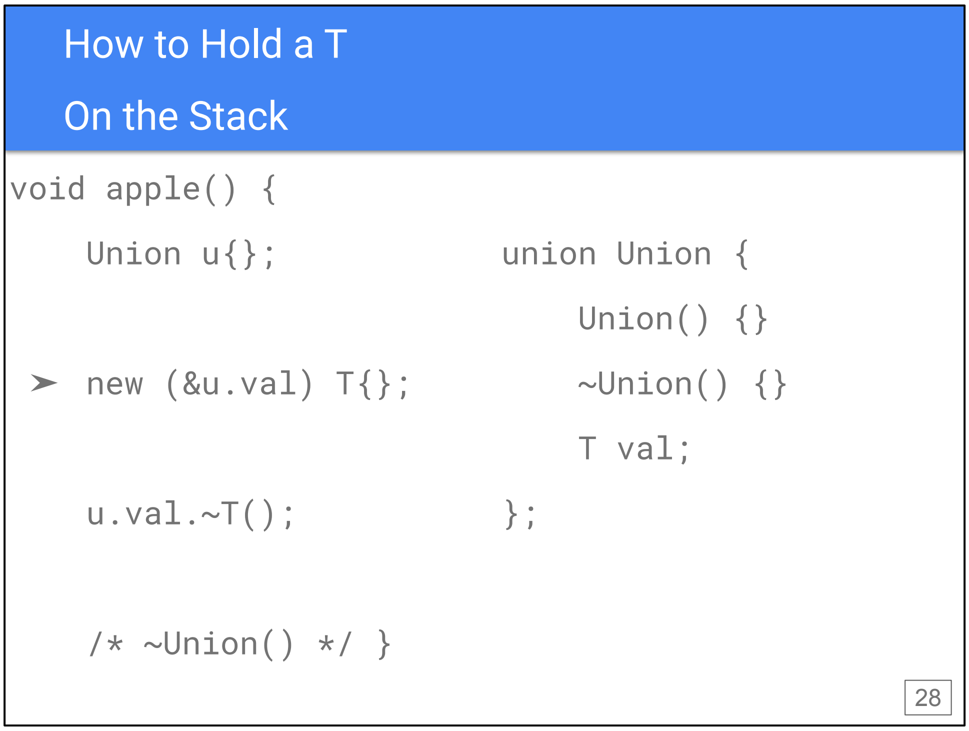
How to Hold a T

On the Stack

```
void apple() {
 ➤  Union u{};                union Union {
                                  Union() {}
    new (&u.val) T{};             ~Union() {}
                                  T val;
    u.val.~T();               };


    /* ~Union() */ }
```

Initially we construct an instance of the union on the stack.

This begins the storage duration for T, but NOT the object lifetime.



How to Hold a T

On the Stack

```
void apple() {                union Union {
    Union u{};                    Union() {}
 ➤  new (&u.val) T{};             ~Union() {}
                                  T val;
    u.val.~T();               };

    /* ~Union() */ }
```

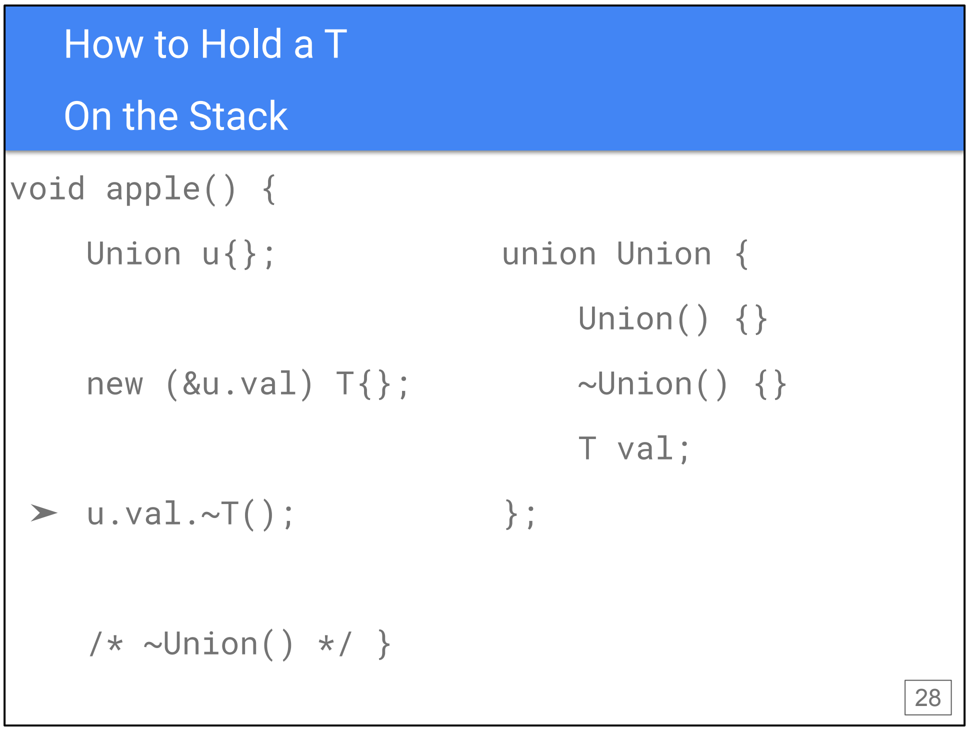
28

Then, fully decoupled from the storage duration, we use something called placement new to invoke T's constructor.

Placement new is like regular new, except it allows us to specify a place in memory into which the T will be constructed.

By using it here, we can use the EXISTING storage duration as the place in memory to begin the object lifetime.

In this case, since the union is on the stack, the T is thus also begin constructed on the stack.



How to Hold a T

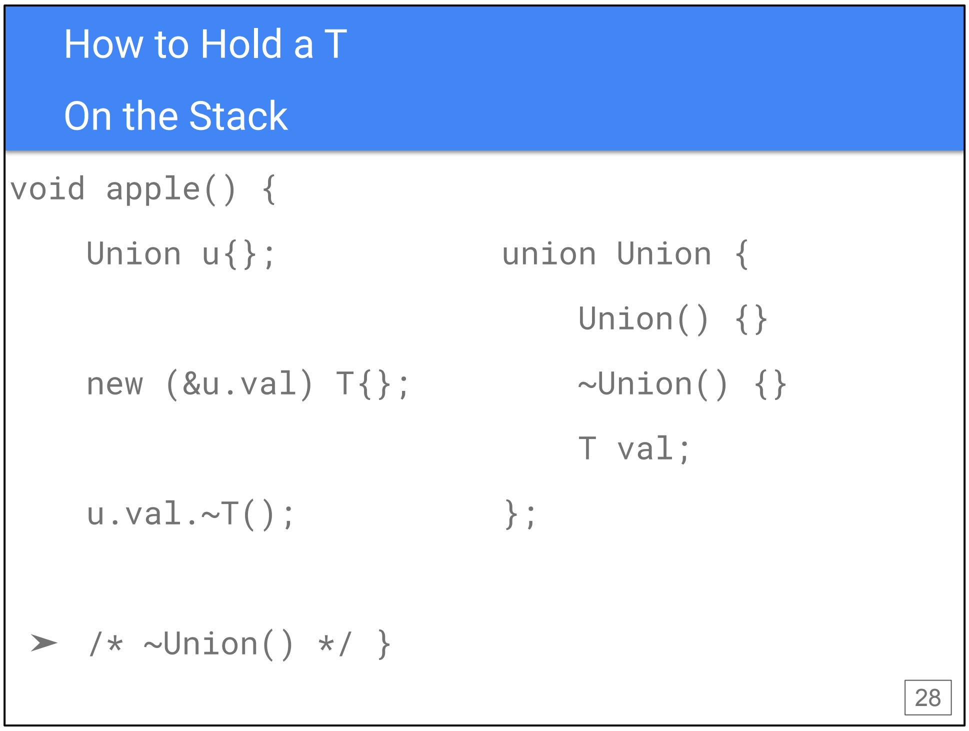
On the Stack

```
void apple() {
    Union u{};                union Union {
                                  Union() {}
    new (&u.val) T{};             ~Union() {}
                                  T val;
 ➤  u.val.~T();               };

    /* ~Union() */ }
```

Then, we manually end the object lifetime of T by calling the destructor directly.

Since this is a manual destructor call, it ONLY ends the lifetime of T.

It does not attempt to deallocate the memory.



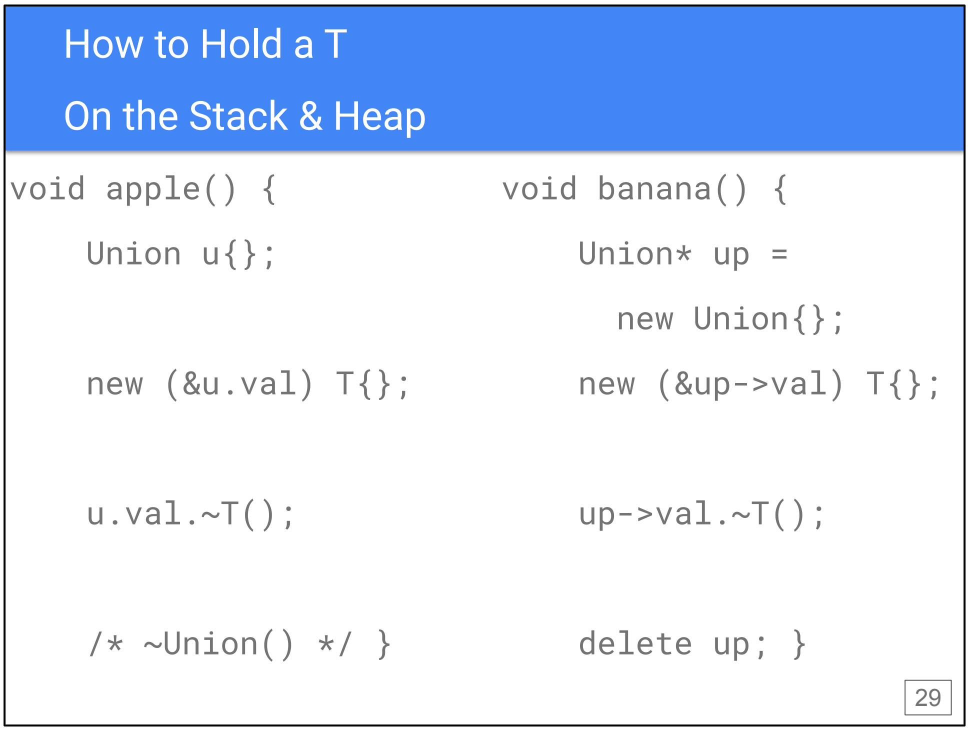
How to Hold a T

On the Stack

```
void apple() {
    Union u{};                union Union {
                                  Union() {}
    new (&u.val) T{};             ~Union() {}
                                  T val;
    u.val.~T();               };

 ➤  /* ~Union() */ }
```

28

Finally, at the end of the scope, the union’s destructor is called, deallocating the stack memory, thus ending T’s storage duration.



How to Hold a T

On the Stack & Heap

```
void apple() {                void banana() {
    Union u{};                    Union* up =
                                    new Union{};
    new (&u.val) T{};             new (&up->val) T{};

    u.val.~T();                   up->val.~T();

    /* ~Union() */ }              delete up; }
```

And for clarity, the same is true of heap decoupling.

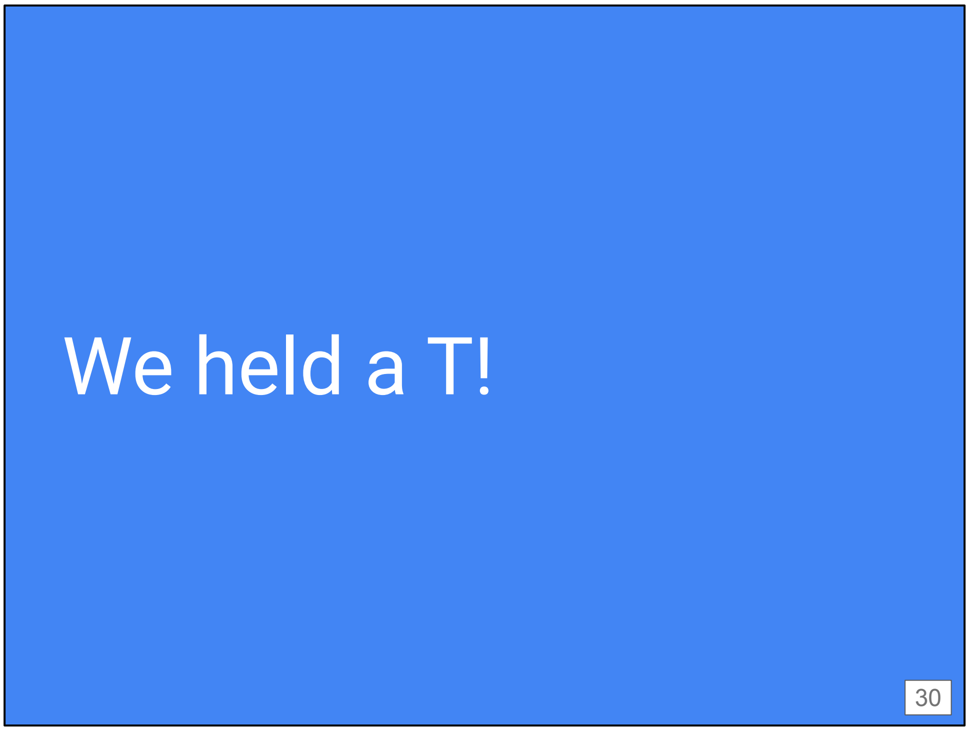
Instead of making the union on the stack, you just use new and delete so make it on the heap, and the rest is the same.

THIS, this code right here, this is what happens inside std::vector, inside std::optional, inside std::variant.

Those types can hold Ts because they

1) use a storage type instead of T directly

and 2) because they use placement new as well as explicit destructor calls, manually managing the object lifetime of T.



We held a T!

30

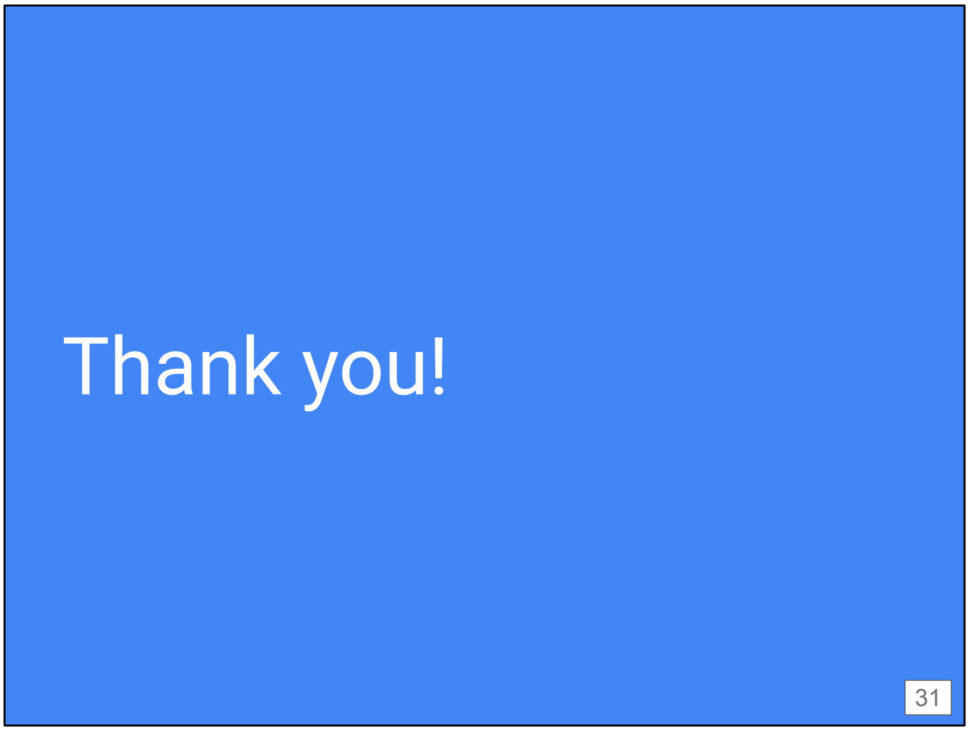
We did it!

We held a T!

Now, there’s a ton involved in building generic containers.

This only one small part of a very wide and very deep topic.

I encourage you to check out other talks and to experiment on your own, to see what you learn.



Thank you!

31

Thank you all for listening!

[PAUSE]

Alright, question time. Any questions from the audience?