STRICT_VARIANT

A simpler variant in C++

Chris Beck

https://github.com/cbeck88/strict_variant

What is a variant?

- A variant is a *heterogenous container.*
  - `std::vector<T>`
    many objects of one type
  - `std::variant<T, U, V>`
    one object of any of `T`, `U`, or `V`
- AKA “tagged-union”, “typesafe union”

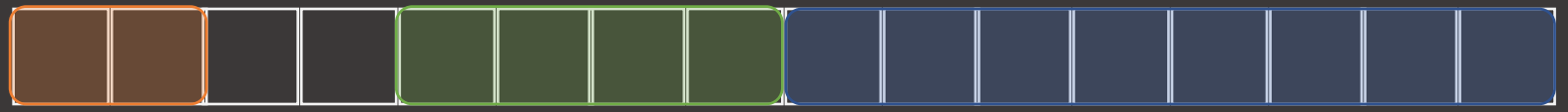
What is a union?

```
struct bar {          // Size is sum of sizes,
  short a;            // plus padding for alignment
  float b;
  double c;
};
```

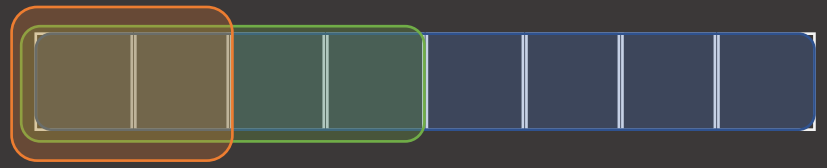
```
union foo {           // Size is max of sizes,
  short a;            // alignment is max of alignments
  float b;
  double c;
};
```

What is a union?

```
struct bar {          // Size is sum of sizes,
  short a;            // plus padding for alignment
  float b;
  double c;
};
```



```
union foo {           // Size is max of sizes,
  short a;            // alignment is max of alignments
  float b;
  double c;
};
```



```
union foo {
  short a;
  float b;
  double c;
};

int main() {
  foo f;
  f.a = 5;
  f.a += 7;
  f.b = 5;
  f.b += .5f;
}
```

Storing to union may change the *active member*.
Reading inactive member may lead to implementation-defined or undefined behavior!

Why would you use this?

- Need to store several types of objects in a collection, but no natural inheritance relation.
- Using an array of unions, store objects *contiguously*, with *very little memory wasted.*
  - Low-level signals / event objects
  - Messages matching various schema

```
struct SDL_KeyboardEvent {
  Uint32 type;        // SDL_KEYDOWN or SDL_KEYUP
  Uint8 state;        // SDL_PRESSED or SDL_RELEASED
  SDL_Keysym keysym; // Represents the key that was pressed
};

struct SDL_MouseMotionEvent {
  Uint32 type;        // SDL_MOUSEMOTION
  Uint32 state;       // bitmask of the current button state
  Sint32 x;
  Sint32 y;
};

union SDL_Event {
  SDL_KeyboardEvent key;
  SDL_MouseMotionEvent motion;
  ...
};
```

Why would you use this?

- A variant is a type-safe alternative to a union
- Prevents you from using inactive members
- Ensures that destructors are called when the active member changes – crucial for C++!

Query the active member using `get`:

```
void print_variant(boost::variant<int, float, double> v) {
  if (const int * i = boost::get<int>(&v)) {
    std::cout << *i;
  } else if (const float * f = boost::get<float>(&v) {
    std::cout << *f;
  } else if (const double * d = boost::get<double>(&v) {
    std::cout << *d;
  } else {
    assert(false);
  }
}
```

Query the active member using `get:`

```
void print_variant(boost::variant<int, float, double> v) {
  if (const int * i = boost::get<int>(&v)) {
    std::cout << *i;
  } else if (const float * f = boost::get<float>(&v) {
    std::cout << *f;
  } else if (const double * d = boost::get<double>(&v) {
    std::cout << *d;
  } else {
    assert(false);
  }
}
```

`boost::get` returns `null` if requested type doesn't match run-time type.

Better, use a visitor:

```
void print_double(double d) {
  std::cout << d;
}

void print_variant(boost::variant<int, float, double> v) {
  boost::apply_visitor(print_double, v);
}
```

Better, use a visitor:

```
void print_double(double d) {
  std::cout << d;
}

void print_variant(boost::variant<int, float, double> v) {
  boost::apply_visitor(print_double, v);
}
```

This only works because `int`, `float` can be promoted to `double` as part of overload resolution.

Using a lambda as a visitor (C++14):

```
void print_variant(boost::variant<int, float, double> v) {
  boost::apply_visitor([](auto val) {
                         std::cout << val;
                       }, v);
}
```

Using a lambda as a visitor (C++14):

```
void print_variant(boost::variant<int, float, double> v) {
  boost::apply_visitor([](auto val) {
                         std::cout << val;
                       }, v);
}
```

No promotion here!
More generally, use templates in the visitor object.

Recursive Data Structures (XML)

```
struct mini_xml;

using mini_xml_node =
  boost::variant<boost::recursive_wrapper<mini_xml>,
                 std::string>;

struct mini_xml {
  std::string name;
  std::vector<mini_xml_node> children;
};
```

Recursive Data Structures (XML)

```
struct mini_xml;

using mini_xml_node =
  boost::variant<boost::recursive_wrapper<mini_xml>,
                 std::string>;

struct mini_xml {
  std::string name;
  std::vector<mini_xml_node> children;
};
```

`recursive_wrapper<T>` is “syntactic sugar”

It works like `std::unique_ptr<T>`

But when visiting, or using `get`, can pretend it is `T`.

Pattern Matching (Rust):

```
enum Message {
  Quit,
  ChangeColor(i32, i32, i32),
  Move { x: i32, y: i32 },
  Write(String),
}

fn process_message(msg: Message) {
  match msg {
    Message::Quit => quit(),
    Message::ChangeColor(r, g, b) => change_color(r, g, b),
    Message::Move { x, y } => move_cursor(x, y),
    Message::Write(s) => println!("{}", s);
  }
}
```

Pattern Matching (C++):

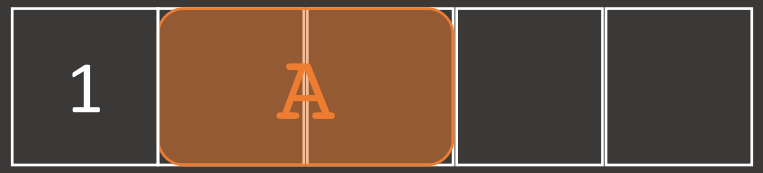
```
using Message = boost::variant<Quit,
                               ChangeColor,
                               Move,
                               Write>;

void process_message(const Message & msg) {
  boost::apply_visitor(
    overload([](Quit) { quit(); },
             [](ChangeColor c) { change_color(c.r, c.g, c.b); }
             [](Move m) { move_cursor(m.x, m.y); }
             [](Write w) { std::cout << w.s << std::endl; }),
    , msg);
}
```

Existing Implementations

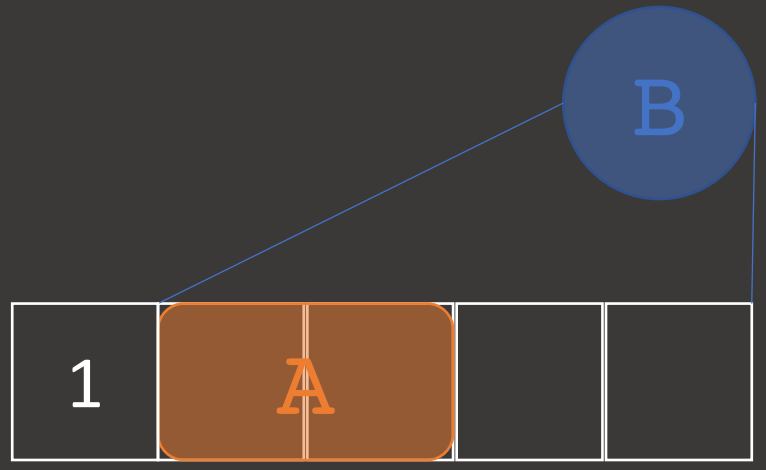
- `boost::variant`
- `std::variant` (C++17)
- `strict_variant` (this talk)
- and others...

Surprisingly, many significant design differences and tradeoffs!

Problem: Exception Safety



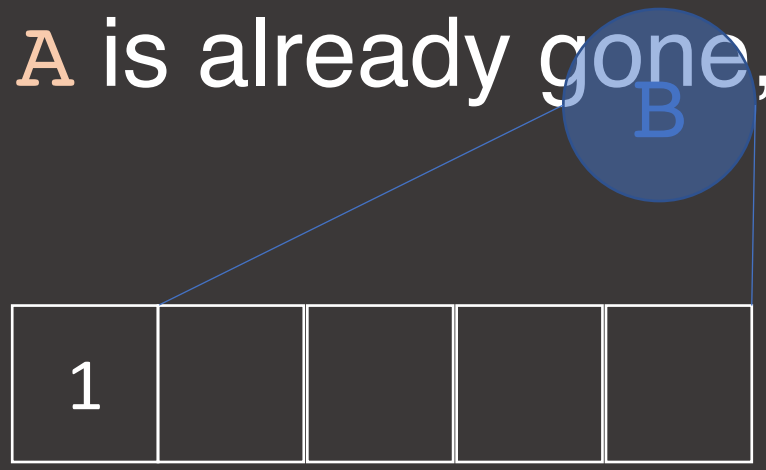
Problem: Exception Safety



Problem: Exception Safety

How to handle *throwing*, *type-changing* assignment.

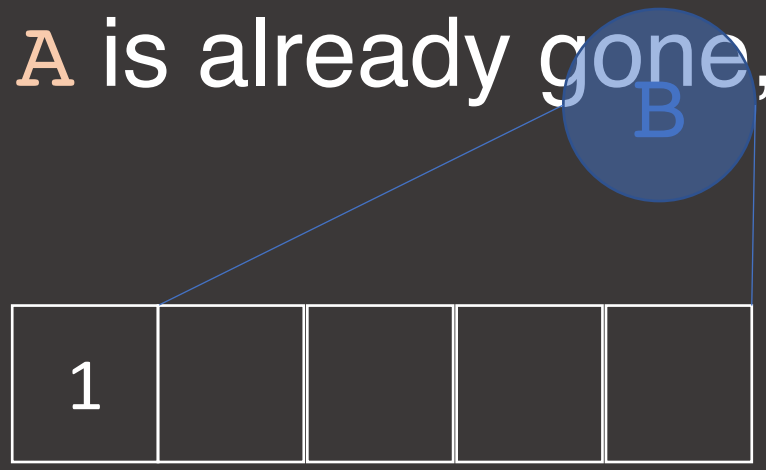
- `~A()`
- Now `B(...)` throws...
- Now what? `A` is already gone, and have no `B`



Problem: Exception Safety

How to handle *throwing*, *type-changing* assignment.

- `~A()`
- Now `B(...)` throws...
- Now what? `A` is already gone, and have no `B`



Problem: Exception Safety

How to handle *throwing*, *type-changing* assignment.

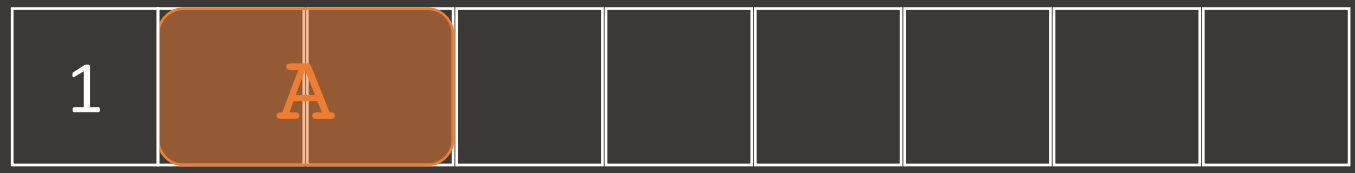
- `~A()`
- Now `B(...)` throws...
- Now what? `A` is already gone, and have no `B`

Problem: Exception Safety

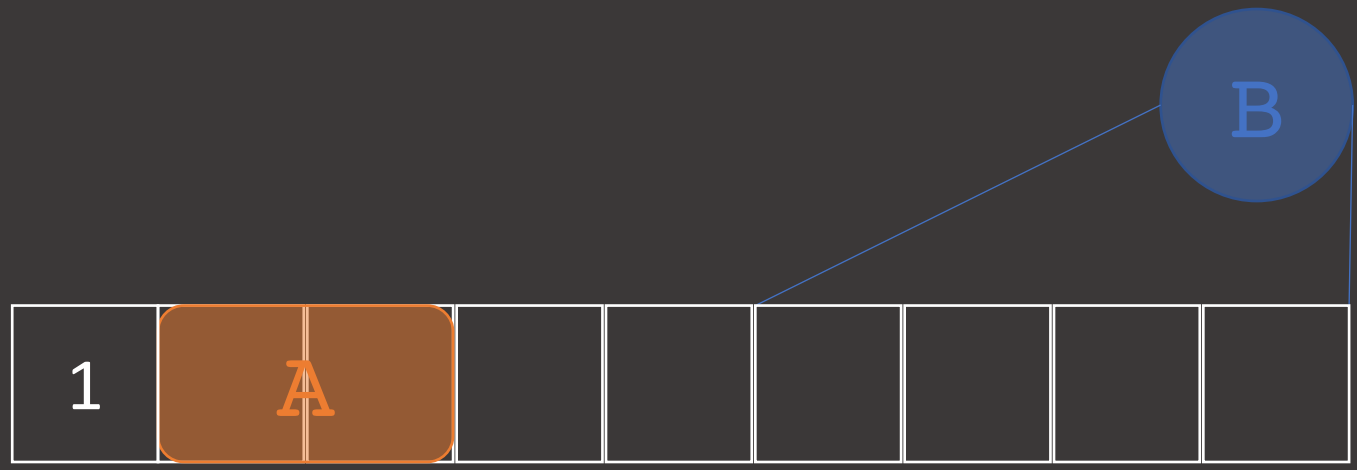
How to handle *throwing*, *type-changing* assignment.

- `~A()`
- Now `B(...)` throws...
- Now what? `A` is already gone, and have no `B`

Solution: Double Storage

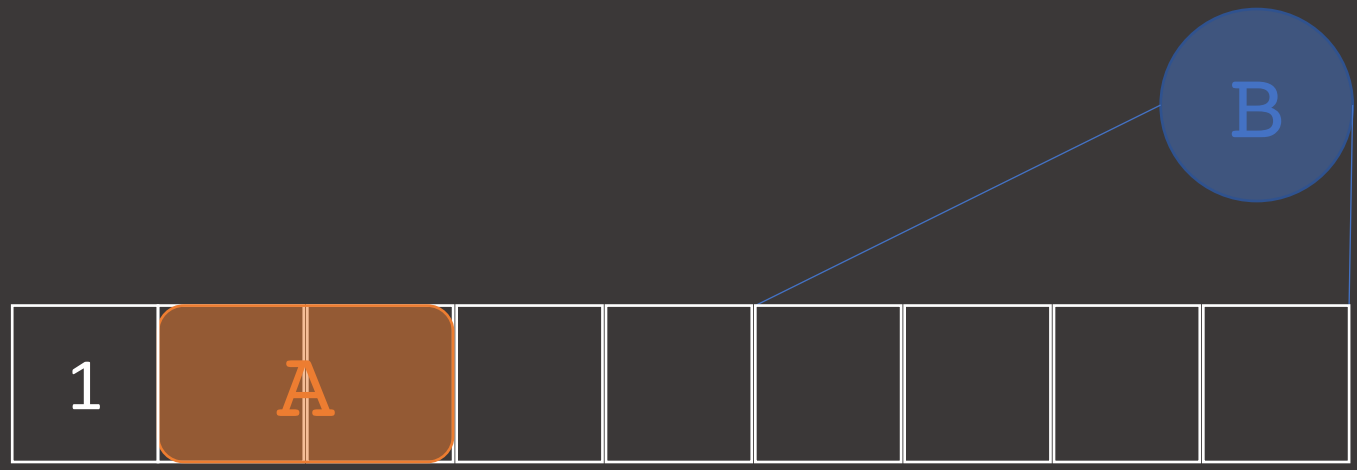


Solution: Double Storage



Solution: Double Storage

- If `B(...)` throws, still have `A`
- `~A()`
- When C comes, flip back to first side



Solution: Double Storage

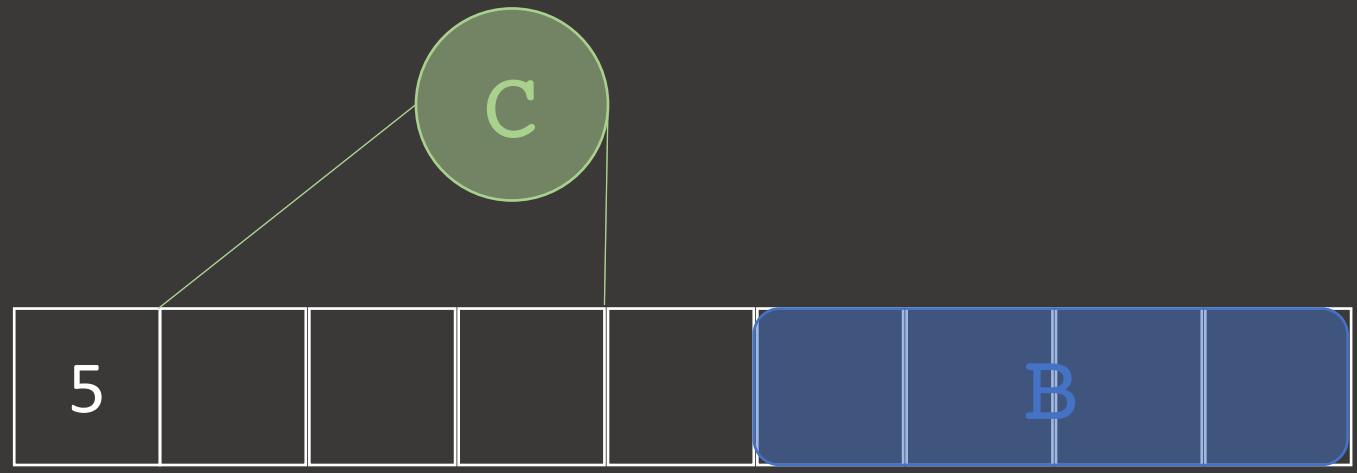
- If `B(...)` throws, still have `A`
- `~A()`
- When C comes, flip back to first side

Solution: Double Storage

- If `B(...)` throws, still have `A`
- `~A()`
- When C comes, flip back to first side

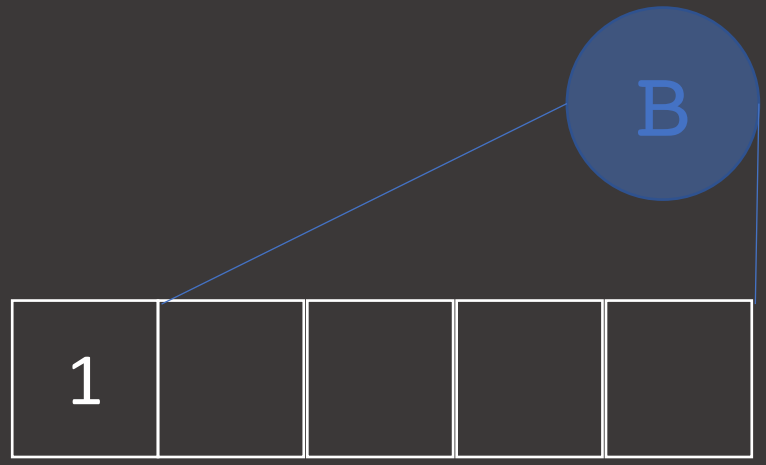
Solution: Double Storage

- If `B(...)` throws, still have `A`
- `~A()`
- When C comes, flip back to first side



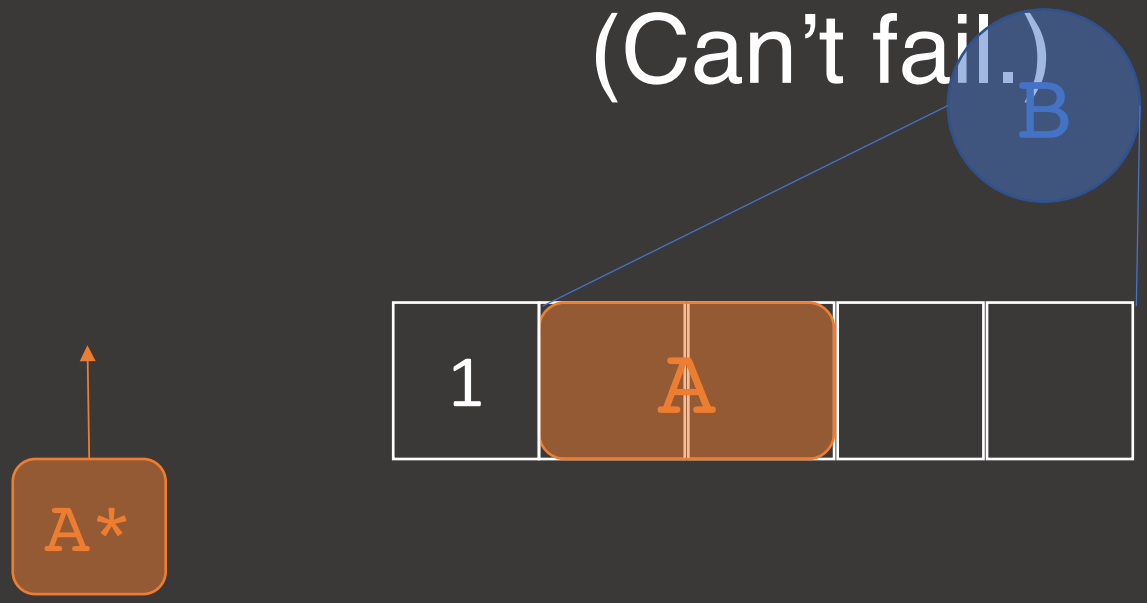
Solution: `boost::variant`

Solution: `boost::variant`



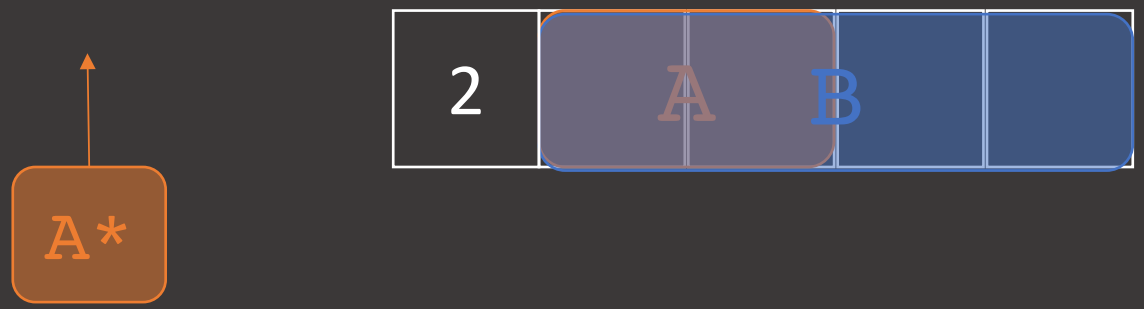
Solution: `boost::variant`

- First move `A` to heap. (If it fails, we are still ok.)
- If `B(...)` succeeds, delete `A` pointer.
- If `B(...)` fails, move `A` pointer to storage. (Can't fail.)



Solution: `boost::variant`

- First move `A` to heap. (If it fails, we are still ok.)
- If `B(...)` succeeds, delete `A` pointer.
- If `B(...)` fails, move `A` pointer to storage.
  (Can't fail.)

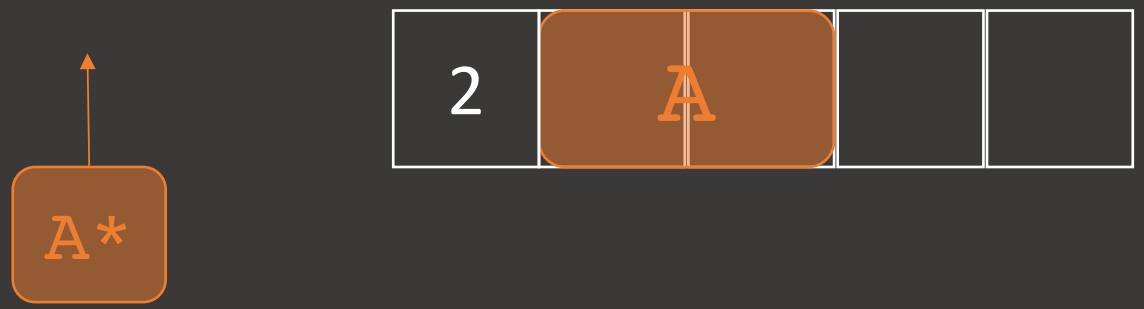


Solution: `boost::variant`

- First move `A` to heap. (If it fails, we are still ok.)
- If `B(...)` succeeds, delete `A` pointer.
- If `B(...)` fails, move `A` pointer to storage. (Can't fail.)

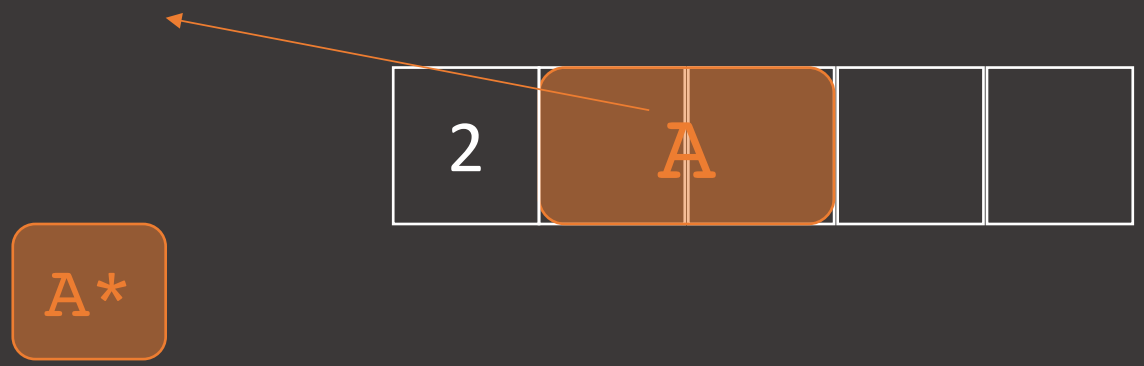
Solution: `boost::variant`

- First move `A` to heap. (If it fails, we are still ok.)
- If `B(...)` succeeds, delete `A` pointer.
- If `B(...)` fails, move `A` pointer to storage.
  (Can't fail.)



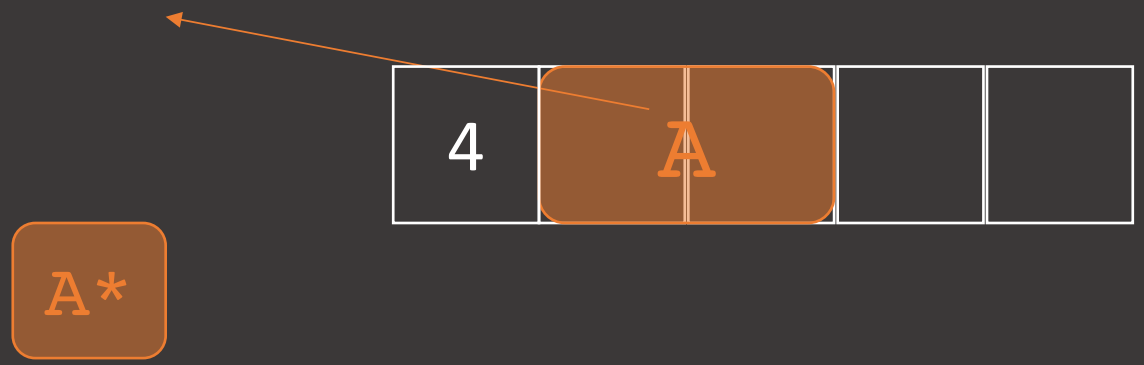
Solution: `boost::variant`

- First move `A` to heap. (If it fails, we are still ok.)
- If `B(...)` succeeds, delete `A` pointer.
- If `B(...)` fails, move `A` pointer to storage.
  (Can't fail.)

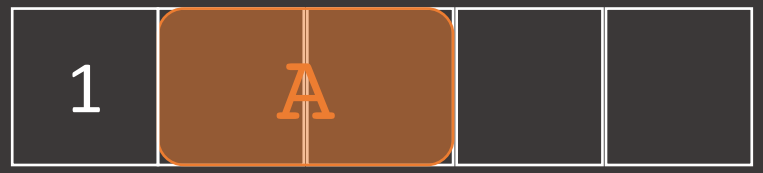


Solution: `boost::variant`

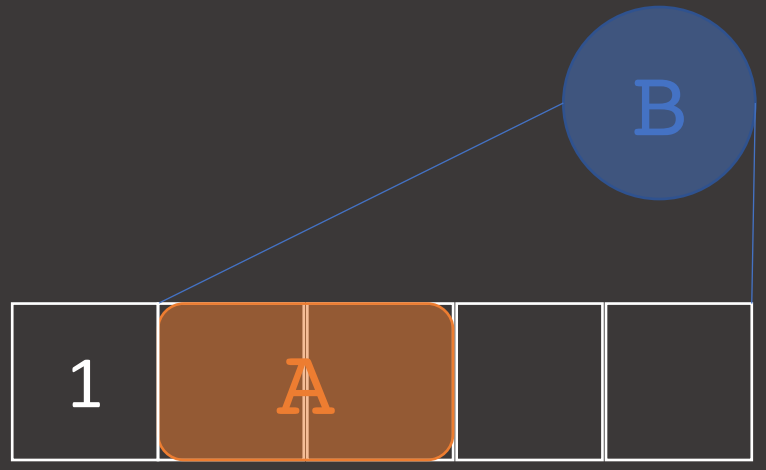
- First move `A` to heap. (If it fails, we are still ok.)
- If `B(...)` succeeds, delete `A` pointer.
- If `B(...)` fails, move `A` pointer to storage.
  (Can't fail.)



Solution: `std::variant`

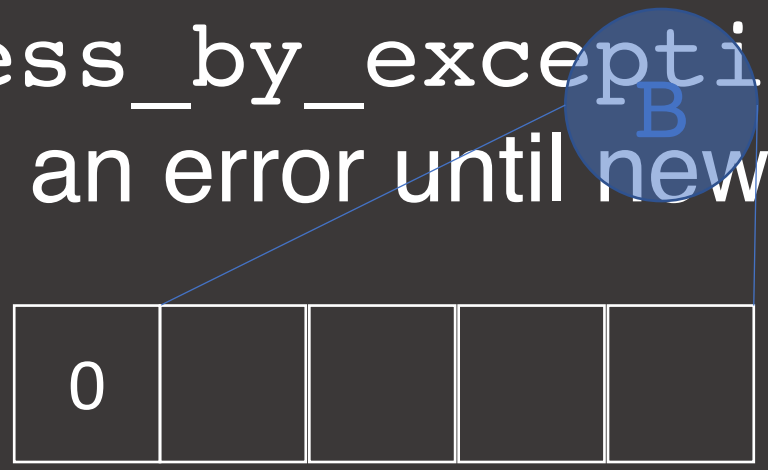


Solution: `std::variant`



Solution: `std::variant`

- `~A()`, set counter to `0`
- Now `B(...)` throws...
- Now we are “empty”.
  - `valueless_by_exception()` reports `true`
  - visiting is an error until new value provided!



Solution: `std::variant`

- `~A()`, set counter to `0`
- Now `B(...)` throws...
- Now we are “empty”.
  - `valueless_by_exception()` reports `true`
  - visiting is an error until new value provided!

Solution: `std::variant`

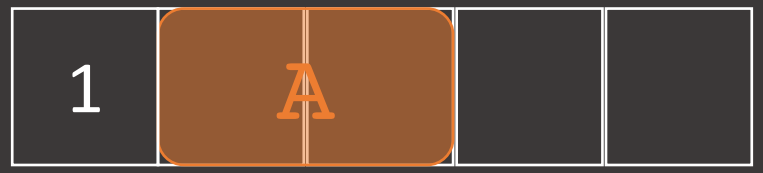
- `~A()`, set counter to `0`
- Now `B(...)` throws...
- Now we are “empty”.
  - `valueless_by_exception()` reports `true`
  - visiting is an error until new value provided!

Tradeoffs

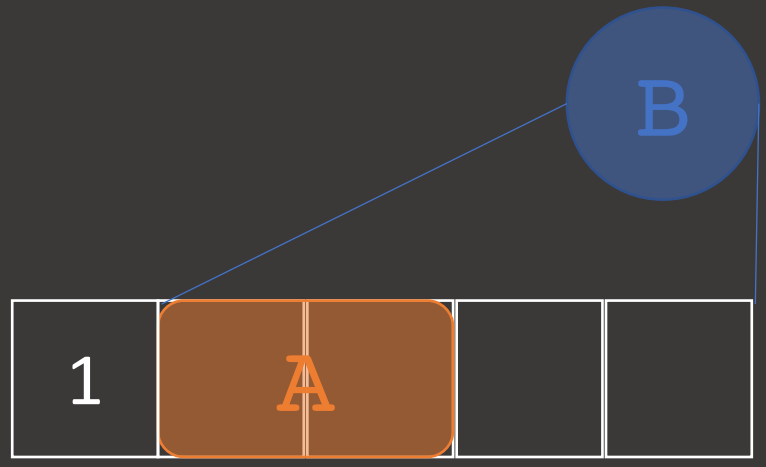
Because of C++ language rules,
we can’t have everything we want.

- No wasted memory
- No empty state
- Strong exception-safety, rollback semantics
- No dynamic allocations, backup copies

Solution: `strict_variant`

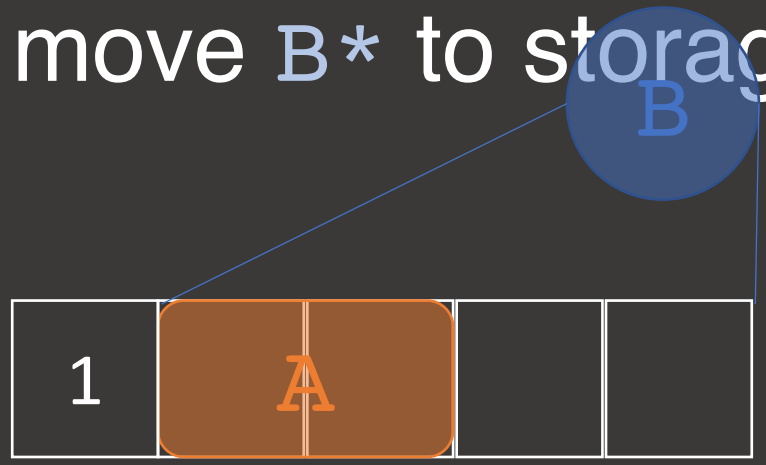


Solution: `strict_variant`



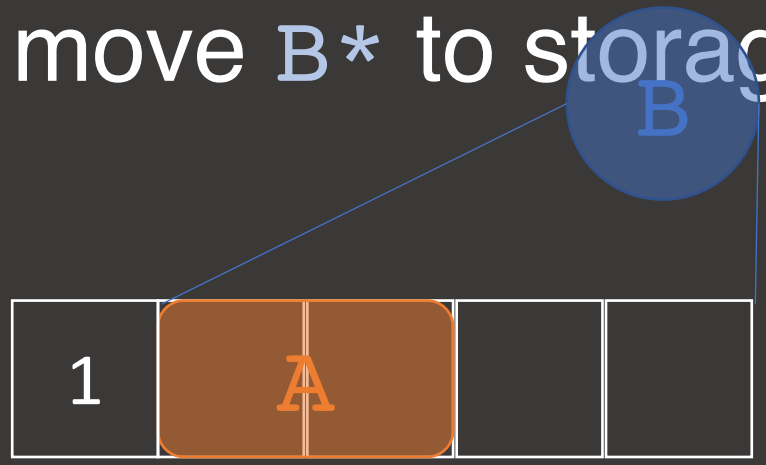
Solution: `strict_variant`

- If `B(B&&)` can't throw, great, do the obvious.
- If `B(B&&)` can throw, `B` always lives on heap.
- Construct `B` on heap. If it fails, didn't touch `A`.
- `~A()`, then move `B*` to storage. Can't fail.



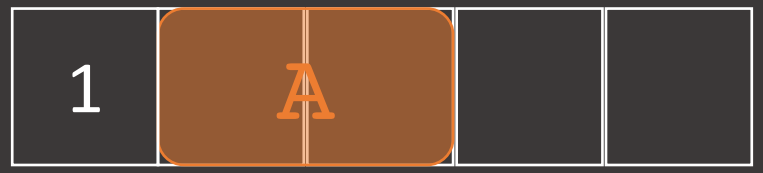
Solution: `strict_variant`

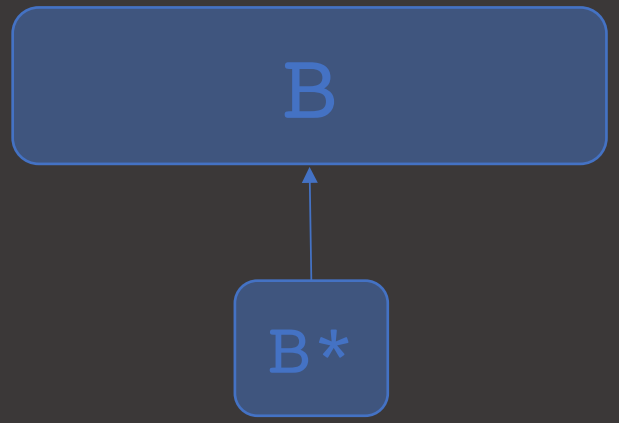
- If `B(B&&)` can't throw, great, do the obvious.
- If `B(B&&)` can throw, `B` always lives on heap.
- Construct `B` on heap. If it fails, didn't touch `A`.
- `~A()`, then move `B*` to storage. Can't fail.



Solution: `strict_variant`

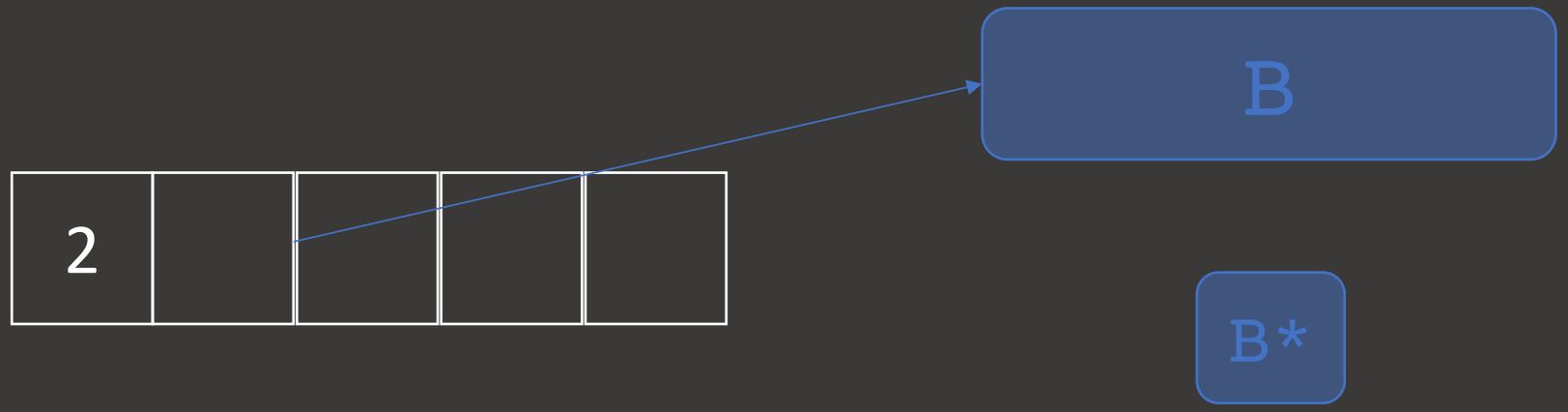
- If `B(B&&)` can't throw, great, do the obvious.
- If `B(B&&)` can throw, `B` always lives on heap.
- Construct `B` on heap. If it fails, didn't touch `A`.
- `~A()`, then move `B*` to storage. Can't fail.





Solution: `strict_variant`

- If `B(B&&)` can't throw, great, do the obvious.
- If `B(B&&)` can throw, `B` always lives on heap.
- Construct `B` on heap. If it fails, didn't touch `A`.
- `~A()`, then move `B*` to storage. Can't fail.



`strict_variant` design

High level design: Reducing to a simpler problem.

`strict_variant` design

High level design: Reducing to a simpler problem.

1. Make a “simple” variant which assumes members are nothrow moveable. (This is easy!)

2. Then, to make a general variant, stick anything that throws in a `recursive_wrapper` and use the simple code. (Pointers can always be moved!)

`strict_variant` design

High level design: Reducing to a simpler problem.

1. Make a “simple” variant which assumes members are nothrow moveable. (This is easy!)

2. Then, to make a general variant, stick anything that throws in a `recursive_wrapper` and use the simple code. (Pointers can always be moved!)

“Step 2”, the reduction, fits here on the screen.

```
template <typename T>
struct wrap_if_throwing_move {
  using type =
    typename std::conditional<
      std::is_nothrow_move_constructible<T>::value,
      T,
      recursive_wrapper<T>
    >::type;
};

template <typename T>
using wrap_if_throwing_move_t = typename wrap_if_throwing_move<T>::type;

template <typename... Ts>
using variant = simple_variant<wrap_if_throwing_move_t<Ts>...>;
```

Why use `strict_variant` instead of `boost::variant`?

- `boost::variant` supports even C++98
- This means, it has to basically work even if we can't check `noexcept` status of operations. This greatly limits design options.
- `strict_variant` targets C++11
  This allows an, IMO, simpler and better strategy.

	Empty State	Exception Safety	Backup Copies	Number of states
double storage	no	yes	no	2n
`std::variant`	yes	no	no	n+1
`boost::variant`	no	yes	yes	2n
`strict_variant`	no	yes	no	n

Other features

- `boost::variant` and `std::variant` sometimes do annoying things

```
std::variant<int, std::string> v;
v = true; // Compiles! Because of bool -> int   :(

std::variant<bool, std::string> u;
u = "The future is now!"; // Selects bool, not std::string!  :(
```

- `strict_variant` uses SFINAE to prevent many “evil” standard conversions here.

THANK YOU

THANK YOU

http://chrisbeck.co

http://github.com/cbeck88/