Back to Basics: Type Erasure

Arthur O'Dwyer
2019-09-20

Outline

- Representation, behavior, and affordances [4–14]
- Let’s make a `TypeErasedNumberRef` [15–16]
- Value semantics and `TypeErasedNumber` [17–22]
- Different layout strategies [23–30]
- Case studies and fun with `std::any` [31–38]
- Questions?

“Oh, I know type erasure from Java...”

- Forget everything you know about Java and C# “type erasure”!
- Those languages use the same term to mean something utterly ***different and unrelated*** to what I’ll talk about today.
- For gory details, see my blog post

“What is Type Erasure?” (March 2019)

Our motivation:

How do I write functions that accept lambdas as arguments?

How do I pass lambdas around?

The “STL” way to accept lambdas is to make all your code into templates, and define those templates in header files.

```
class Shelf {
    template<class Func>
    void for_each_book(Func f) {
        for (const Book& b : books_) f(b);
    }
};
```

This template definition must be visible in the same TU as a declaration of `$_1::operator()` — so, the same TU as the lambda itself.

Suppose this lambda type gets mangled as `$_1`.

```
Shelf myshelf;
myshelf.for_each_book([](auto&& book){ book.print(); });
```

How do I pass lambdas around?

Alternatively, you can use ***type erasure*** to capture your lambda inside a library type that exposes just the call operator.

```
class Shelf {
    void for_each_book(ConcreteCBType f) {
        for (const Book& b : books_) f(b);
    }
};

Shelf myshelf;
myshelf.for_each_book([](auto&& book){ book.print(); });
```

We construct a `ConcreteCBType` object (by implicit conversion) from our original rvalue of type $_1.

This function definition must be in the same TU as a declaration of `ConcreteCBType::operator()` — so, you have more freedom where to place it.

`ConcreteCBType` might just be an alias for `std::function<void(const Book&)>`.

Type erasure “concretifies” templates

`std::sort` is implemented as a function template.

- Can take any kind of `Comparator`
- All inlined all the time: very fast
- ***Must*** be defined in a header file: code bloat, slow to compile

By contrast, C-style `qsort` takes a function pointer parameter.

- Can only take that one specific kind of comparator
- ***Cannot*** be inlined; we pay a performance penalty
- ***Can*** be defined out-of-line in libc.so
- Arguably not as “type-safe” as we’d like

Can we get the best of both worlds? (Yes.)

C’s solution: qsort_r

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));

void qsort_r(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *, void *), void *arg);
```

The **`qsort_r`**`()` function is identical to **`qsort`**`()` except that the comparison function *`compar`* takes a third argument. A pointer is passed to the comparison function via *`arg`*. In this way, the comparison function does not need to use global variables to pass through arbitrary arguments...

```
int byprop(const void *a, const void *b, void *cookie) {
    const Map& x = *(const Map *)a; const Map& y = *(const Map *)b;
    const Key& prop = *(const Key*)cookie;
    return (x[prop] < y[prop]) ? -1 : (x[prop] > y[prop]);
}
... qsort_r(a, n, sizeof *a, byprop, (void*)&prop_author); ...
... qsort_r(a, n, sizeof *a, byprop, (void*)&prop_title); ...
```

With apologies to Niklaus Wirth...

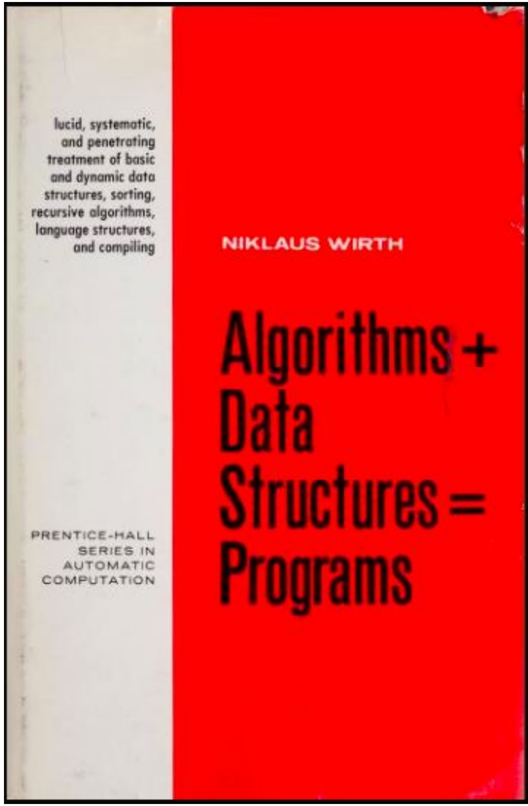
Wirth (1976) says: “Algorithms plus data structures equals programs.”

I say: “Representation plus behavior equals data type.”

```
void qsort_r(void *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *, void *),
        void *cookie);
```

Don’t think of this as a *function* compar that incidentally takes some *data input* cookie.

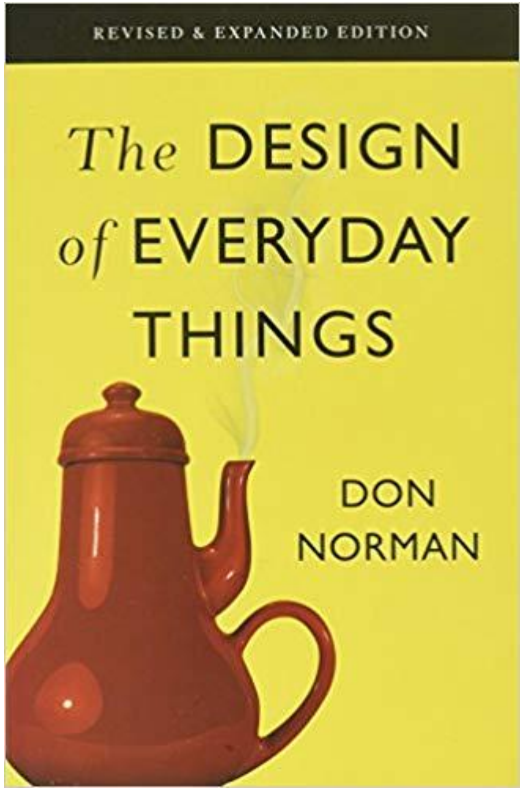
Think of it as a *data representation* cookie with an associated *behavior* compar.



and apologies to Don Norman...

Norman (1988) says: "***Affordances*** refer to the potential actions that are possible."

We don't say "When performing the action of *opening*, we must supply a door." Instead, we say "If we have a door, we ***can*** open it." A door ***affords*** opening.




```
void qsort_r(void *base, size_t nmemb, size_t size,
        int (*compar)(const void *, const void *, void *),
        void *cookie);
```

Don't think of `compar` as an action that *requires* you to supply a `cookie`...

What can `qsort_r` do with `cookie`? Pass it to `compar`, that's all.

Think about what actions are *afforded* by `cookie` itself.

`cookie` affords only one interesting behavior.

Any Callable can be split this way

Given any callable function object

```
template<class Callable>
void foo(Callable& func)
```

Suppose `func` affords calling-with-no-arguments-and-returning-an-`int`. We can split it up into its representation and its behavior, like this:

```
void *representation = &callable;

int (*behavior_when_called)(void*) = +[](void *r) {
    return (*(Callable *)r)();
};

assert(behavior_when_called(representation) == func());
```

Any Callable can be split this way

We started with an object of type `Callable`. We can't use that object without knowing about type `Callable`.

```
Callable& func = ...;
```

When we split it into its representation and its behavior, we erase all inconsequential aspects of type `Callable` (size, alignment, copyability, triviality...). We keep only objects of simple known types. Our original object's type `Callable` no longer appears in these declarations.

```
void *representation = ...;
int (*behavior_when_called)(void*) = ...;
```

We are working our way toward ***type erasure***.

Any *affordance* can be split this way

It's not just for calls! Given any ***negatable*** object

```
template<class Negatable>
void foo(Negatable& number)
```

Suppose `number` affords being-bitwise-negated-and-returning-an-`int`. We can split it up into its representation and its behavior, like this:

```
void *representation = &number;

int (*behavior_when_negated)(void*) = +[](void *r) {
    return ~(*(Negatable *)r);
};

assert(behavior_when_negated(representation) == ~number);
```

Multiple *affordances* may exist

Suppose "`Numeric number`" affords being-bitwise-negated-and-returning-an-`int`, and ***also*** affords being-boolean-notted-and-returning-a-`bool`. We can split it up into its representation and its behavior***s***, like this:

```
void *representation = &number;

int (*behavior_when_negated)(void*) = +[](void *r) {
    return ~(*(Number *)r);
};
bool (*behavior_when_notted)(void*) = +[](void *r) {
    return !(*(Number *)r);
};

assert(behavior_when_negated(representation) == ~number);
assert(behavior_when_notted(representation) == !number);
```

Wrap rep and behavior into a struct

`qsort_r` suffers from being written in C, which has no class types.

Also, `qsort_r` is a very simple case because `cookie` affords only one operation — comparison.

In C++ we can do better.

```
struct TypeErasedNumberRef {
    void *representation_;
    int (*negate_)(void*);
    bool (*not_)(void*);
    int operator~() const { return negate_(representation_); }
    bool operator!() const { return not_(representation_); }
};
```

We know how to make a `TypeErasedNumberRef` out of any kind of `Number`.

Think: *constructor*

Think: *template*

```
struct TypeErasedNumberRef {
    template<class Number> TypeErasedNumberRef(Number& n) :
        representation_( (void*)&n ),
        negate_( [](void *r)->int  { return -(*(Number*)r); } ),
        not_(    [](void *r)->bool { return !(*(Number*)r); } )
        {}
    void *representation_;
    int (*negate_)(void*);
    bool (*not_)(void*);
    int operator~() const { return negate_(representation_); }
    bool operator!() const { return not_(representation_); }
};
```

```
int x = 42; TypeErasedNumberRef ref(x);
assert(~ref == ~x); assert(!ref == !x);
```

We can use our *constructor template* to wrap any Number into a TypeErasedNumberRef.

But what about ownership?

Our `TypeErasedNumberRef` has ***reference*** semantics.

- It captures the original object’s address
- When the original object’s lifetime ends, the reference is invalidated

```
auto x = TypeErasedNumberRef(42);  // fails to compile

TypeErasedNumberRef danger(int x) {
    return TypeErasedNumberRef(x);
}                               // dangling reference at runtime
```

These semantics are still very useful...

- C++2a `std::function_ref`, for example

...but let’s see how to deal with lifetime and ownership

“Destructibility” is an affordance

Let’s start building a `TypeErasedNumber` with value semantics (not reference semantics).

`TypeErasedNumber` no longer “refers to” an `int`, `BigNum`, etc.

It needs to “capture” an actual object of type `int`, `BigNum`, etc. inside itself, and manage that object’s lifetime.

`~TypeErasedNumber` is responsible for destroying the captured object.

The captured object has a representation and some affordances.

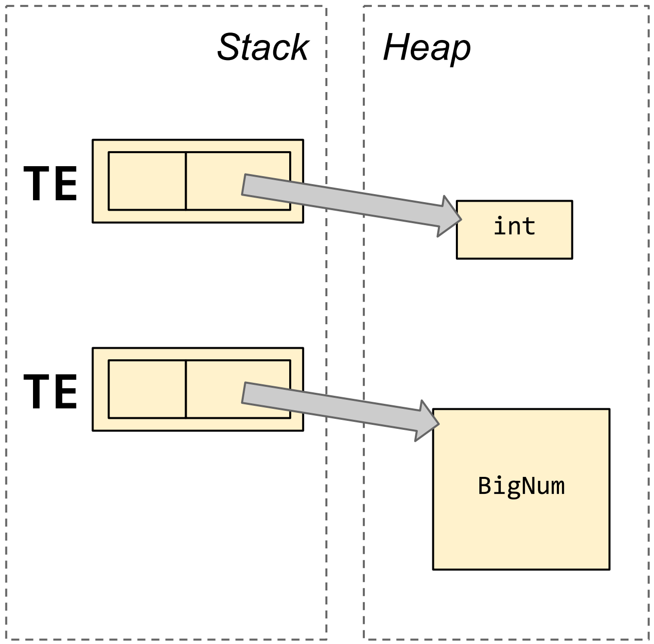
- “Destructibility” must now be one of those affordances.

The captured object may be large

To capture a copy of the user's original `int`, `BigNum`, etc., inside our `TypeErasedNumber`, we must have enough space to do it.

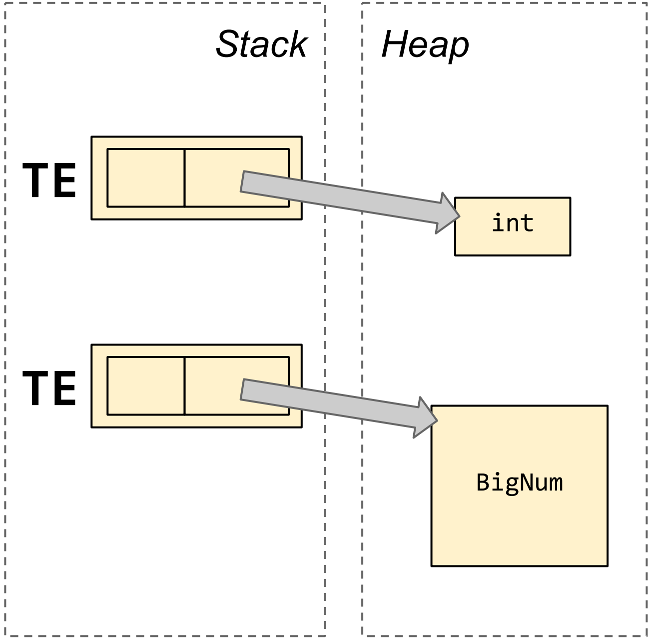
What should `sizeof(TypeErasedNumber)` be?

- No matter what we pick, `sizeof(UserType)` might be bigger.
- So we punt and use the heap.
- There are other options, but they are out of scope for this talk.



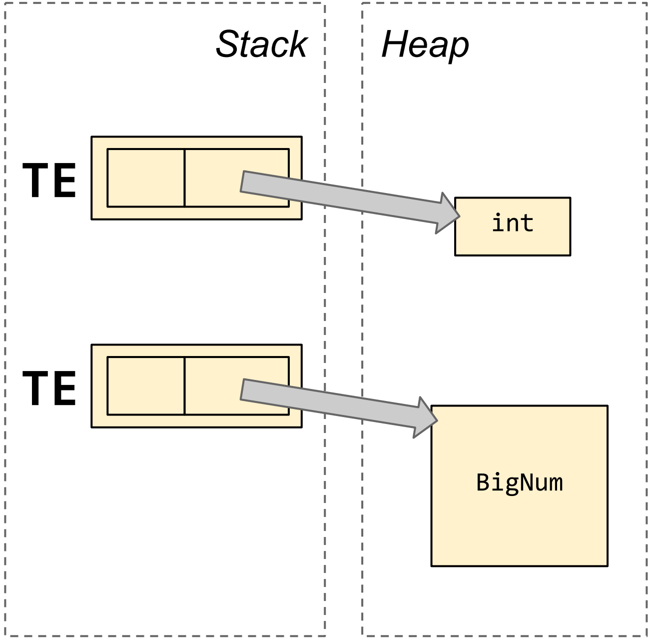
Manually managing lifetime: destroy

```
struct TE {
    void *repr_;
    void (*delete_)(void *);

    template<class Number>
    TE(Number n) :
      repr_(new Number(std::move(n))),
      delete_([](void *r) {
        delete (Number*)r;
      })
    {}

    ~TE() { delete_(repr_); }
};
```



Manually managing lifetime: copy

```
struct TE {
    void *repr_;
    void* (*clone_)(void *);

    template<class Number>
    TE(Number n) :
        repr_(new Number(std::move(n))),
        clone_([](void *r) {
            return new Number(*(Number*)r);
        })
    {}

    TE(const TE& rhs) :
      repr_(rhs.clone_(rhs.repr_)),
      clone_(rhs.clone_) {}
};
```



Guideline: *Affordances*

Our essential strategy here is:

- **List** what operations must be afforded by a `Number` in order for `TE` to do its job.
  - Special members (copy, move, destroy) also count as affordances!
- For each operation, write it as a lambda in terms of `representation_` and `Number`, with a **fixed function signature**.
  - Our constructor template will initialize function pointers using those lambdas.
  - Each lambda's *behavior* depends on `Number`, but its *signature* is fixed.

But `TE`'s footprint is growing...

Our `TypeErasedNumber` is accumulating too many function pointers inside itself.

```
struct TE {
    void *representation_;
    void *(*clone_)(void*);
    void (*delete_)(void*);
    int (*negate_)(void*);
    bool (*not_)(void*);
};
```

8 bytes...

16...

24...

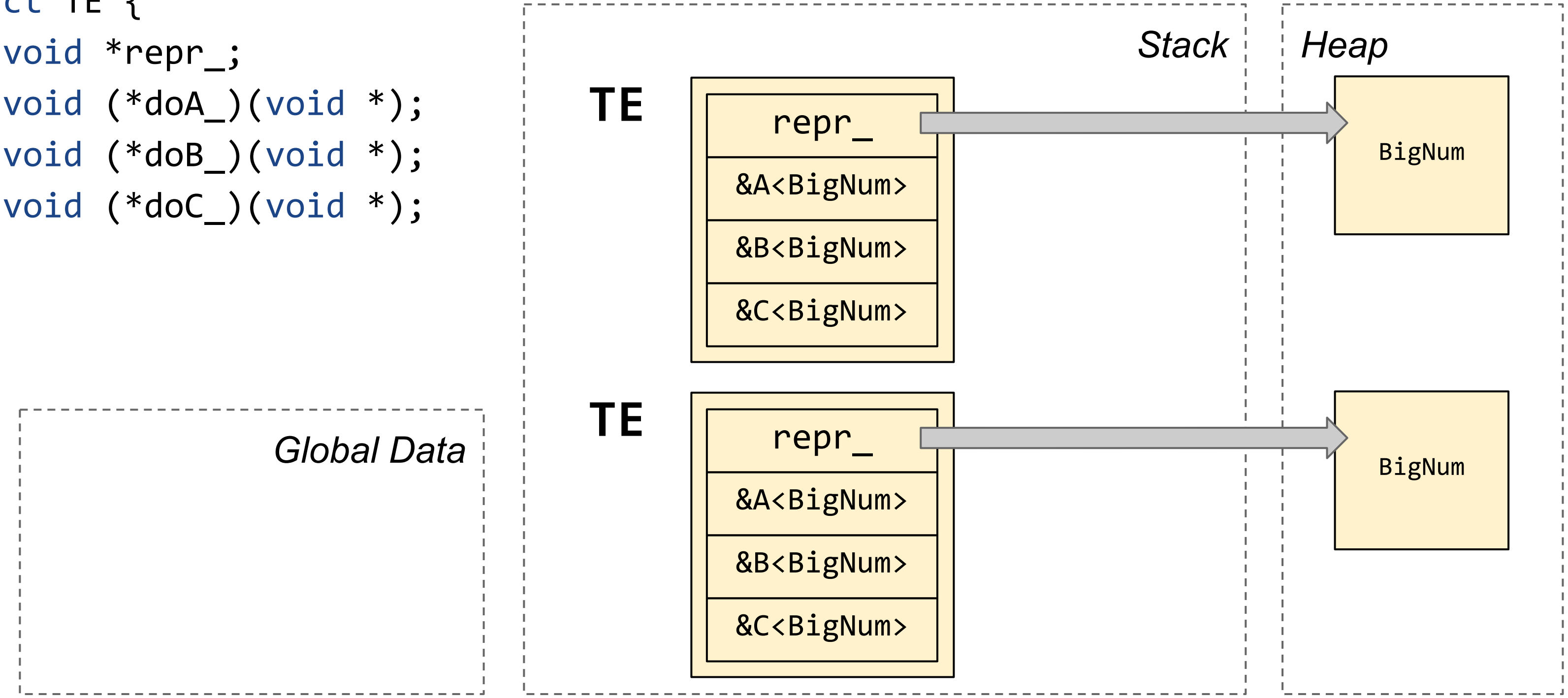
32...

40. TE is now *ten times* `sizeof(int)`!

How can we shrink `TE`'s memory footprint?

Where should the behaviors go?

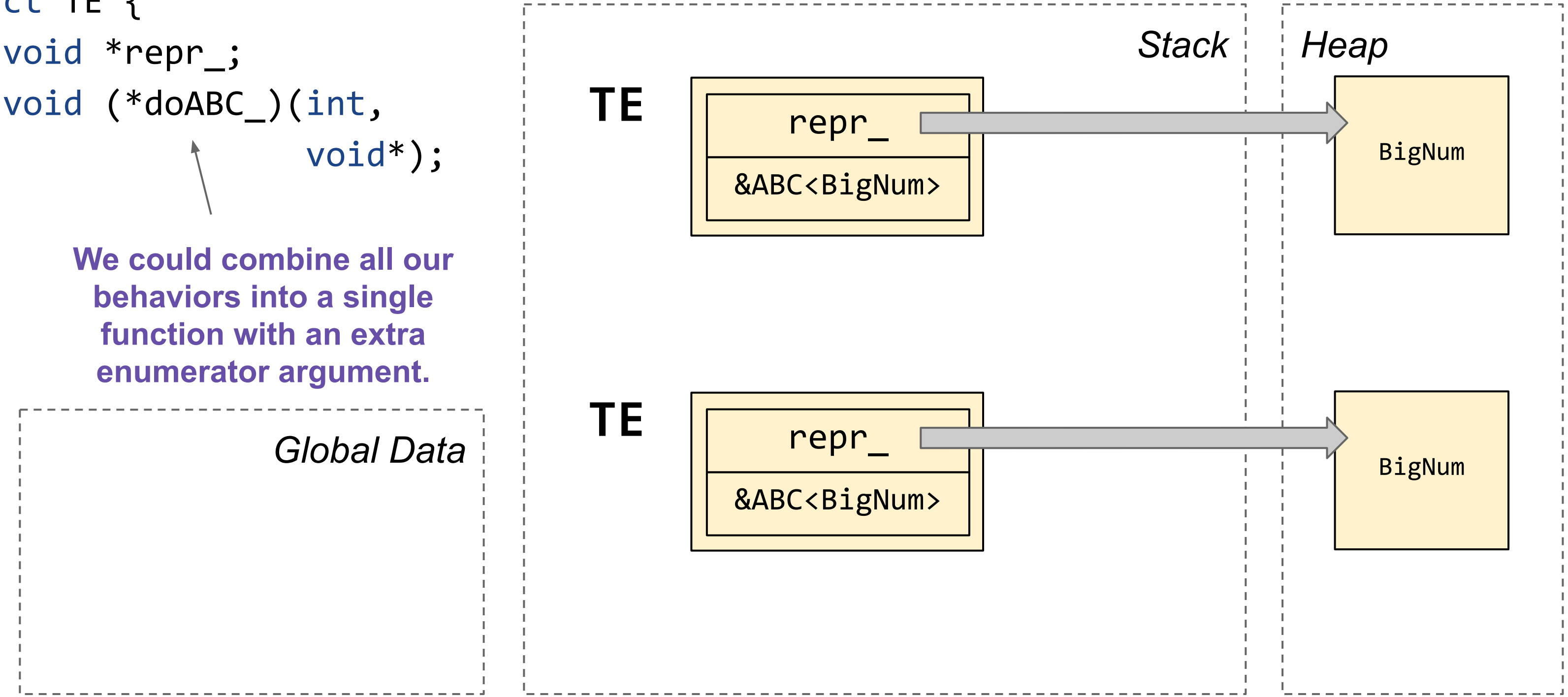
```
struct TE {
    void *repr_;
    void (*doA_)(void *);
    void (*doB_)(void *);
    void (*doC_)(void *);
};
```



Where should the behaviors go?

```
struct TE {
    void *repr_;
    void (*doABC_)(int,
                   void*);
};
```

We could combine all our behaviors into a single function with an extra enumerator argument.



One-function is popular these days

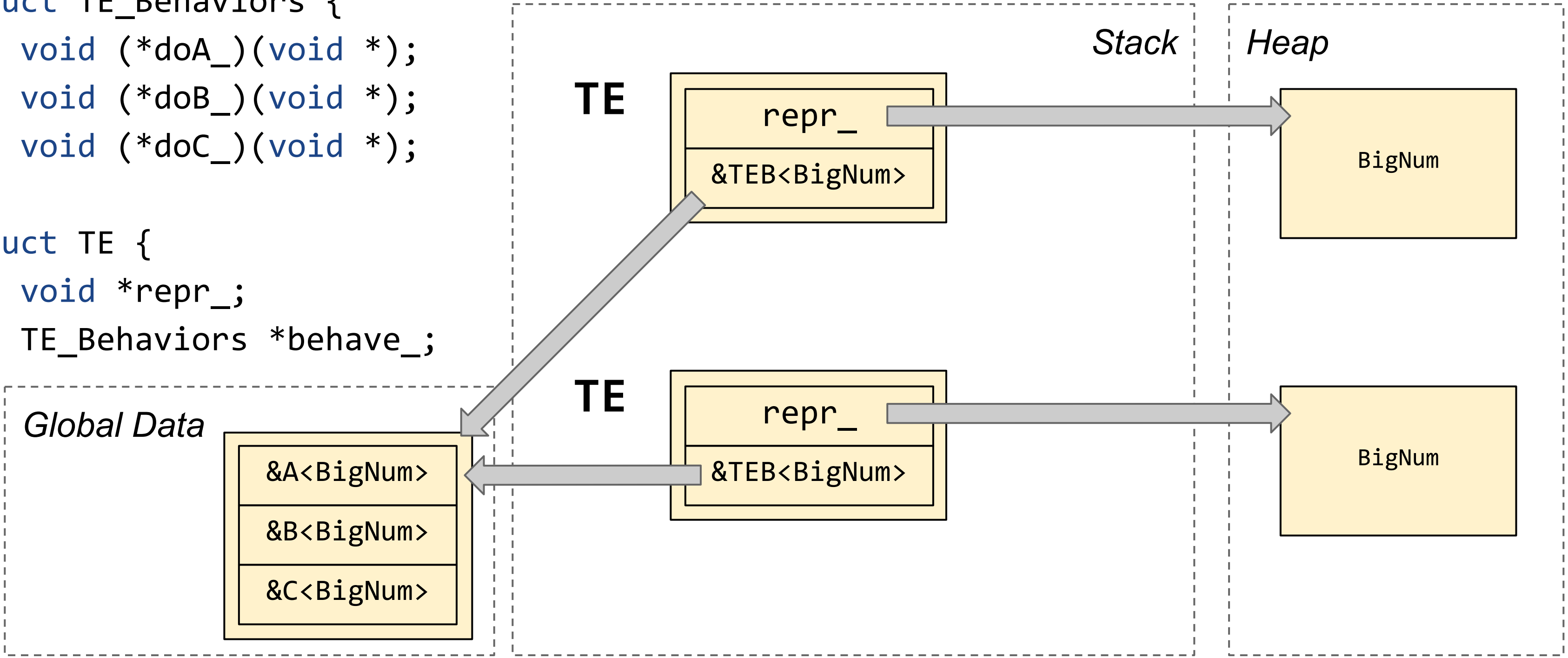
```
struct TE {
    void *repr_ = nullptr;
    void *(*abc_)(int, void*, void*) = nullptr;
    template<class Number> TE(Number n) :
        repr_(new Number(std::move(n))),
        abc_(ABC<Number>) {}

    void *clone() const { return abc_(0, repr_, nullptr); }
    TE(const TE& rhs) : repr_(rhs.clone()), abc_(rhs.abc_) {}
    int operator-() const { int v; abc_(1, repr_, &v); return v; }
    bool operator!() const { bool v; abc_(2, repr_, &v); return v; }
    ~TE() { abc_(3, repr_, nullptr); }

    void swap(TE& rhs) noexcept {
        std::swap(repr_, rhs.repr_); std::swap(abc_, rhs.abc_);
    }

    TE(TE&& rhs) noexcept { this->swap(rhs); }
    TE& operator=(TE rhs) noexcept { this->swap(rhs); return *this; }
};
```

Only two data members now!

```
template<class Number>
void *ABC(int op, void *r,
                  void *out)
{
  Number& num = *(Number *)r;
  switch (op) {
    case 0:
      return new Number(num);

    case 1:
      *(int*)out = -num;
      return nullptr;

    case 2:
      *(bool*)out = !num;
      return nullptr;

    case 3:
      delete &num;
      return nullptr;
  }
}
```

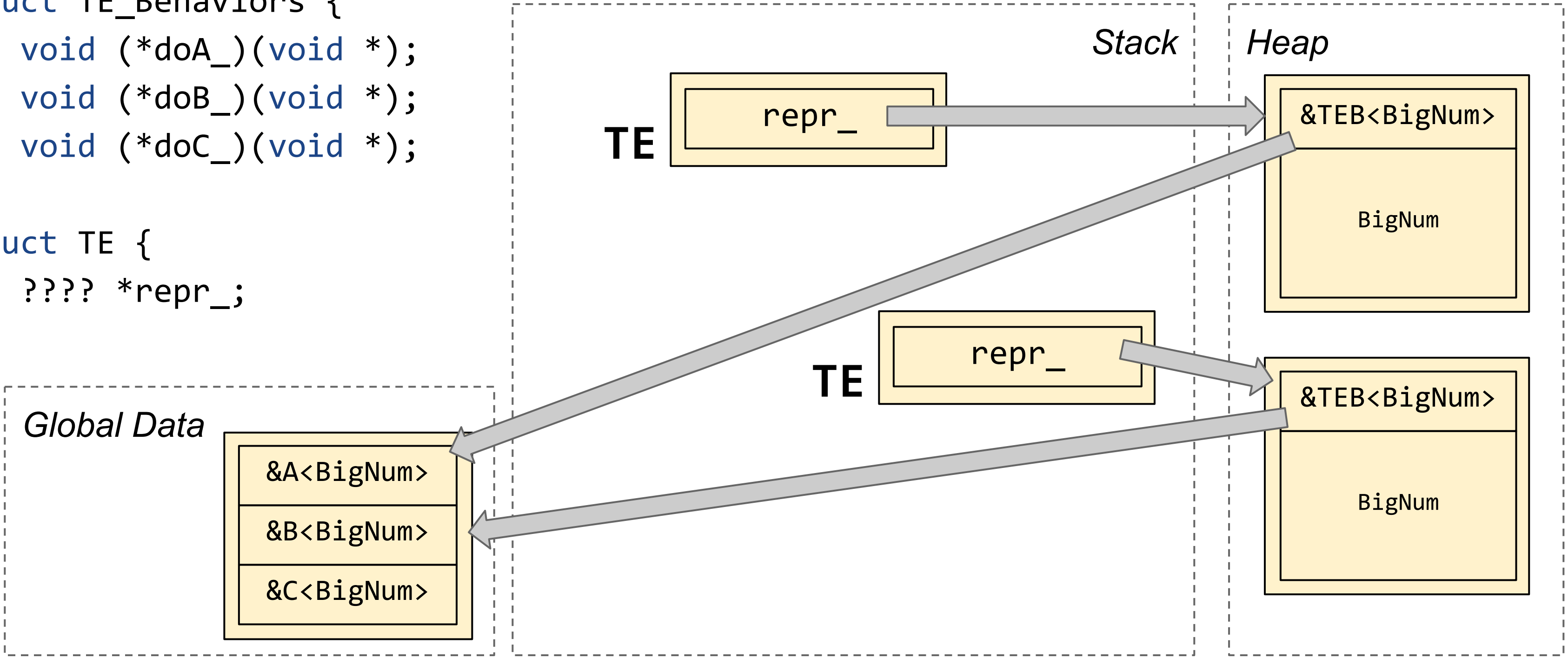
Where should the behaviors go?

```
struct TE_Behaviors {
    void (*doA_)(void *);
    void (*doB_)(void *);
    void (*doC_)(void *);
};
struct TE {
    void *repr_;
    TE_Behaviors *behave_;
};
```



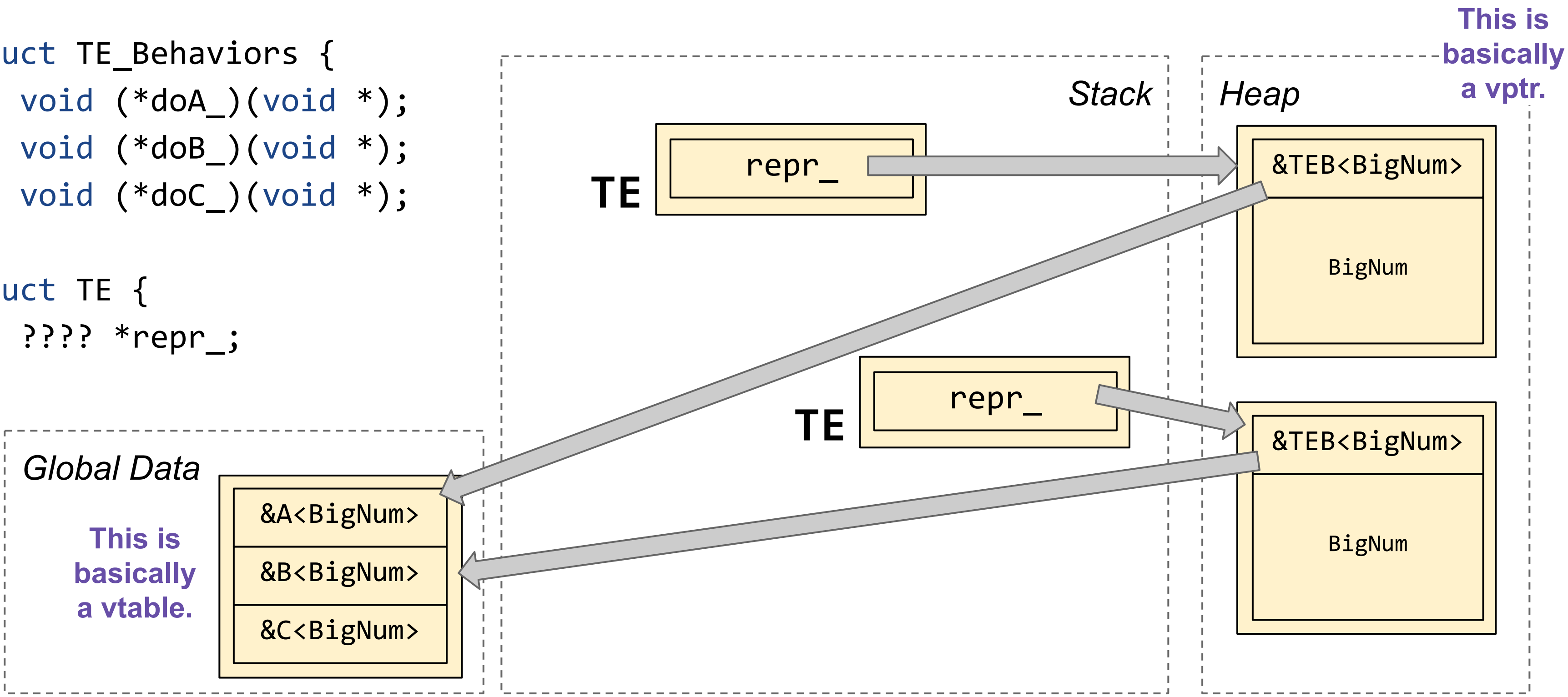
Where should the behaviors go?

```
struct TE_Behaviors {
    void (*doA_)(void *);
    void (*doB_)(void *);
    void (*doC_)(void *);
};
struct TE {
    ???? *repr_;
};
```



Now we can do something type-safe...

```
struct TE_Behaviors {
    void (*doA_)(void *);
    void (*doB_)(void *);
    void (*doC_)(void *);
};
struct TE {
    ???? *repr_;
};
```



https://godbolt.org/z/2VBBTv

Type-safe type erasure!

```
struct TEBase {
    virtual unique_ptr<TEBase> clone() const = 0;
    virtual int negate() const = 0;
    virtual bool not_() const = 0;
    virtual ~TEBase() = default;
};

template<class Number>
struct TED : public TEBase {
    Number num_;

    explicit TED(Number n) : num_(std::move(n)) {}

    unique_ptr<TEBase> clone() const override {
        return std::make_unique<TED>(num_);
    }
    int negate() const override { return -num_; }
    bool not_() const override { return !num_; }
};
```

```
struct TE {
    unique_ptr<TEBase> p_ = nullptr;

    template<class Number>
    TE(Number n) : p_(
        make_unique<TED<Number>>(std::move(n))
    ) {}

    TE(const TE& rhs) : p_(rhs.p_->clone()) {}
    TE(TE&&) noexcept = default;
    TE& operator=(TE rhs) {
        std::swap(p_, rhs.p_); return *this;
    }
    ~TE() = default;

    int operator-() const {
        return p_->negate();
    }
    bool operator!() const {
        return p_->not_();
    }

};
```

What can we make with type erasure?

- std::function<S>
  - Wraps anything that affords ***copying***, ***destroying***, and ***calling*** with the signature S. For example, double(int, int).
- std::any
  - Wraps anything that affords ***copying*** and ***destroying***.

New in C++17!

- function_ref<S>
  - Wraps anything that affords ***calling*** with signature S.

Maybe in C++2B

- unique_function<S>
  - Wraps anything that affords ***destroying*** and ***calling*** with signature S.

Maybe in C++2B

`std::any` is a funny one

- `std::any`
  - Wraps anything that affords ***copying*** and ***destroying***.
  - But how do we get anything out of it?
  - We can get something out of a `std::function<int()>` by calling it.
  - We can get something out of a `TypeErasedNumber` by negating it (returns `int`) or logical-notting it (returns `bool`).
  - How do we get ***anything*** out of a `std::any`, if all we can do is copy and destroy them?

Well, `std::any` supports one more affordance...

std::any_cast<> implements “Go Fish”

```
std::any mya = 42;
int i = std::any_cast<int>(mya);  // returns 42
mya = 3.14;
double d = std::any_cast<double>(mya);  // returns 3.14
int j = std::any_cast<int>(mya);  // Oops! Throws std::bad_any_cast
```

https://godbolt.org/z/j_yB1i

std::any_cast (simplified)

```
struct AnyBase {
    virtual unique_ptr<AnyBase> clone() const = 0;
    virtual void *addr() = 0;
    virtual ~AnyBase() = default;
};

template<class T>
struct AnyD : public AnyBase {
    T value_;

    explicit AnyD(const T& t) : value_(t) {}

    unique_ptr<AnyBase> clone() const override {
        return std::make_unique<AnyD>(value_);
    }

    void *addr() override { return &value_; }
};
```

```
any mya(42);
int i = any_cast<int>(mya);
```

```
struct any {
    unique_ptr<AnyBase> p_ = nullptr;

    template<class T>
    explicit any(const T& t) : p_(
        std::make_unique<AnyD<T>>(t)
    ) {}

    any(const any& rhs): p_(rhs.p_->clone()) {}
    any(any&&) noexcept = default;
    any& operator=(any rhs) { ... }
    ~any() = default;
};

template<class U>
U any_cast(any& a) {
    AnyBase *b = a.p_.get();
    if (auto *d = dynamic_cast<AnyD<U>*>(b)) {
        return *(U*)a.p_->addr();
    }
    throw std::bad_any_cast();
}
```

std::any_cast (different, maybe worse?)

```
struct AnyBase {
    virtual unique_ptr<AnyBase> clone() const = 0;
    virtual void *addr() = 0;
    virtual const std::type_info& id() = 0;
    virtual ~AnyBase() = default;
};

template<class T>
struct AnyD : public AnyBase {
    T value_;

    explicit AnyD(const T& t) : value_(t) {}
    unique_ptr<AnyBase> clone() const override {
        return std::make_unique<AnyD>(value_);
    }

    void *addr() override { return &value_; }

    const std::type_info& id() override {
        return typeid(T);
    }
};
```

```
struct any {
    unique_ptr<AnyBase> p_ = nullptr;

    template<class T>
    explicit any(const T& t) : p_(
        std::make_unique<AnyD<T>>(t)
    ) {}

    any(const any& rhs): p_(rhs.p_->clone()) {}
    any(any&&) noexcept = default;
    any& operator=(any rhs) { ... }
    ~any() = default;
};

template<class T>
T any_cast(any& a) {
    if (a.p_->id() == typeid(T)) {
        return *(T*)a.p_->addr();
    }
    throw std::bad_any_cast();
}
```

nonstd::any_cast (just for fun)

```
struct AnyBase {
    virtual unique_ptr<AnyBase> clone() const = 0;
    virtual void toss() = 0;
    virtual ~AnyBase() = default;
};

template<class T>
struct AnyD : public AnyBase {
    T value_;

    explicit AnyD(const T& t) : value_(t) {}
    unique_ptr<AnyBase> clone() const override {
        return std::make_unique<AnyD>(value_);
    }

    void *addr() override { return &value_; }

    void toss() override {
        throw &value_;
    }
};
```

```
struct any {
    unique_ptr<AnyBase> p_ = nullptr;

    template<class T>
    explicit any(const T& t) : p_(
        std::make_unique<AnyD<T>>(t)
    ) {}

    any(const any& rhs): p_(rhs.p_->clone()) {}
    any(any&&) noexcept = default;
    any& operator=(any rhs) { ... }
    ~any() = default;
};

template<class T>
T any_cast(any& a) {
    try {
        a.p_->toss();
    } catch (T *ptr) {
        return *(T*)ptr;
    } catch (...) {}
    throw std::bad_any_cast();
}
```

This version is not standard-conforming. It lets you any_cast an Apple as a Fruit.

What can we make with type erasure?

- `sg14::inplace_function<S, Capacity>`
  - Wraps anything that affords ***copying***, ***destroying***, and ***calling*** with the signature `S`... as long as it fits in `Capacity` bytes.

Okay, so variations on `std::function` are the killer app for type erasure?

More or less, yes. When we use type erasure, we are erasing everything about a type except for certain ***behaviors***. Each behavior must have a fixed ***signature***.

This is practically the definition of `std::function<Sig>`!

What can we make with type erasure?

The STL also uses type erasure in the ***deleter*** of `shared_ptr`.

```
auto sw = std::make_shared<Widget>();
std::shared_ptr<int> si(std::move(sw), &sw->value);
auto sj = std::make_shared<int>();
```

`si`'s deleter destroys a `Widget`. `sj`'s deleter trivially destroys an `int`.
Yet `si` and `sj` have the same static type.

Again, a “deleter” is awfully close to a `function<void(void*)>`!

What about non-unary behaviors?

Our `TypeErasedNumber` supports `-` and `!`, but what about `/`?

```
TE one = 1;
TE two = 2.0;
TE half = one / two;  // Can we do this?
```

Sadly, this is "multiple dispatch" / "open multi-methods" in disguise.
C++ basically can't do this.

A Polyglot's Guide to Multiple Dispatch (Eli Bendersky, April 2016)

Conclusion

- `std::function` and `std::any` are done with type erasure
- Type erasure lets us pass arbitrary types ***across ABI boundaries***
    - The flexibility of templates, with the speed of separate compilation
- Type erasure is ***not*** too hard to write by yourself
    - List your affordances
    - Make a vtable (either manually or using virtual methods)
    - Initialize each behavior (in a constructor template with a bunch of lambdas, or with a derived class template)
- Copyability and destructibility are just other affordances

Questions?