RUNTIME POLYMORPHISM: BACK TO THE BASICS

LOUIS DIONNE, CPPCON 2017

WHAT IS RUNTIME POLYMORPHISM AND WHEN DO YOU NEED IT?

CONSIDER THE FOLLOWING

```
struct Car {
  void accelerate();
};

struct Truck {
  void accelerate();
};

struct Plane {
  void accelerate();
};
```

RETURNING RELATED TYPES FROM A FUNCTION

```
??? getVehicle(std::istream& user) {
  std::string choice;
  user >> choice;
  if      (choice == "car")   return Car{...};
  else if (choice == "truck") return Truck{...};
  else if (choice == "plane") return Plane{...};
  else                        die();
}
```

STORING RELATED TYPES IN A CONTAINER

```cpp
int main() {
  // Should store anything that has an accelerate() method
  std::vector<???> vehicles;

  vehicles.push_back(Car{...});
  vehicles.push_back(Truck{...});
  vehicles.push_back(Plane{...});

  for (auto& vehicle : vehicles) {
    vehicle.accelerate();
  }
}
```

`variant` SOMETIMES DOES THE TRICK

- But it only works for closed set of types
- Using visitation is sometimes (often?) not convenient

BOTTOM LINE:

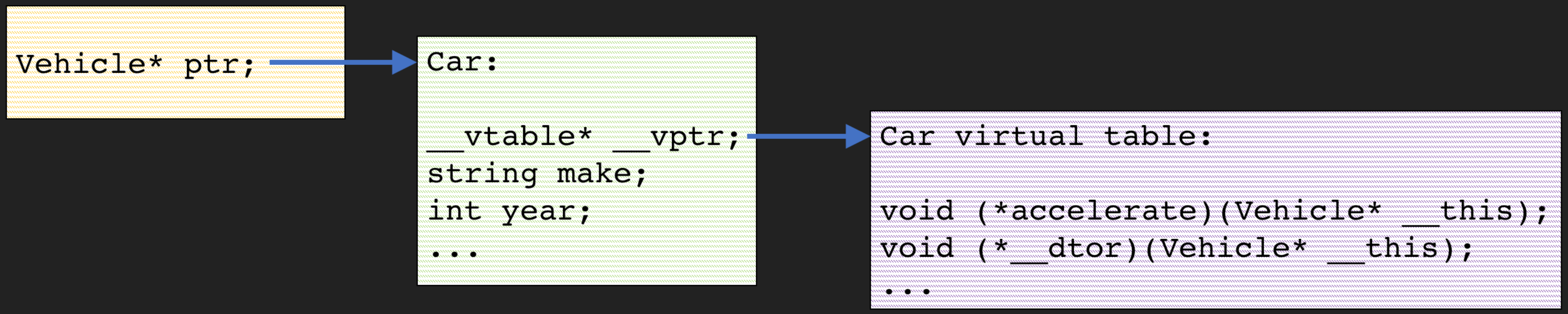
MANIPULATING AN OPEN SET OF RELATED TYPES WITH DIFFERENT REPRESENTATIONS

C++ HAS A SOLUTION FOR THAT!

INHERITANCE

```
struct Vehicle {
  virtual void accelerate() = 0;
  virtual ~Vehicle() { }
};

struct Car : Vehicle {
  void accelerate() override;
};

struct Truck : Vehicle {
  void accelerate() override;
};

struct Plane : Vehicle {
  void accelerate() override;
};
```

UNDER THE HOOD



```
Vehicle* ptr;
```

```
Car:

__vtable* __vptr;
string make;
int year;
...
```

```
Car virtual table:

void (*accelerate)(Vehicle* __this);
void (*__dtor)(Vehicle* __this);
...
```

ASIDE

INHERITANCE HAS MANY PROBLEMS

BAKES IN REFERENCE SEMANTICS

```
void foo(Vehicle* vehicle) {
  Vehicle* copy = vehicle;
  ...
  copy->accelerate();
  ...
}
```

HEAP ALLOCATIONS

```
std::unique_ptr<Vehicle> getVehicle(std::istream& user) {
  std::string choice;
  user >> choice;
  if      (choice == "car")   return std::make_unique<Car>(...);
  else if (choice == "truck") return std::make_unique<Truck>(...);
  else if (choice == "plane") return std::make_unique<Plane>(...);
  else                        die();
}
```

BAKES IN NULLABLE SEMANTICS

```cpp
std::unique_ptr<Vehicle> vehicle = getVehicle(std::cin);
// can vehicle be null?
```

OWNERSHIP HELL

```
Vehicle*                 getVehicle(std::istream& user);
std::unique_ptr<Vehicle> getVehicle(std::istream& user);
std::shared_ptr<Vehicle> getVehicle(std::istream& user);
```

DOESN'T PLAY WELL WITH ALGORITHMS

```
std::vector<std::unique_ptr<Vehicle>> vehicles;
vehicles.push_back(std::make_unique<Car>(...));
vehicles.push_back(std::make_unique<Truck>(...));
vehicles.push_back(std::make_unique<Plane>(...));

std::sort(vehicles.begin(), vehicles.end()); // NOT what you wanted!
```

INTRUSIVE

```
namespace lib {
  struct Motorcycle { void accelerate(); };
}

void foo(Vehicle& vehicle) {
  ...
  vehicle.accelerate();
  ...
}

Motorcycle bike;
foo(bike); // can't work!
```

LISTEN TO SEAN PARENT, NOT ME

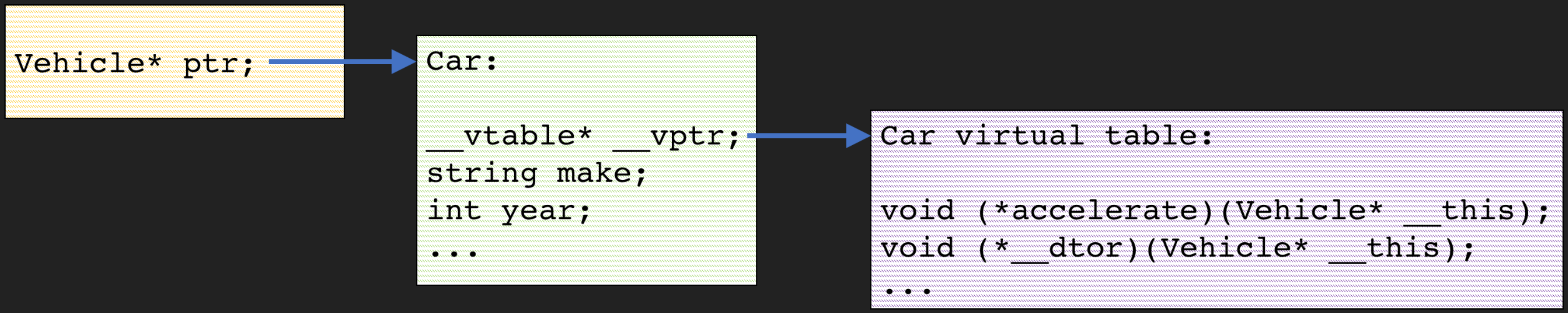
https://youtu.be/QGcVXgEVMJg

I JUST WANTED THIS!

```
interface Vehicle { void accelerate(); };

namespace lib {
  struct Motorcycle { void accelerate(); };
}
struct Car   { void accelerate(); };
struct Truck { void accelerate(); };

int main() {
  std::vector<Vehicle> vehicles;
  vehicles.push_back(Car{...});
  vehicles.push_back(Truck{...});
  vehicles.push_back(lib::Motorcycle{...});

  for (auto& vehicle : vehicles) {
    vehicle.accelerate();
  }
}
```

HOW MIGHT THAT WORK?

WITH INHERITANCE



```
Vehicle* ptr;
```

```
Car:

__vtable* __vptr;
string make;
int year;
...
```

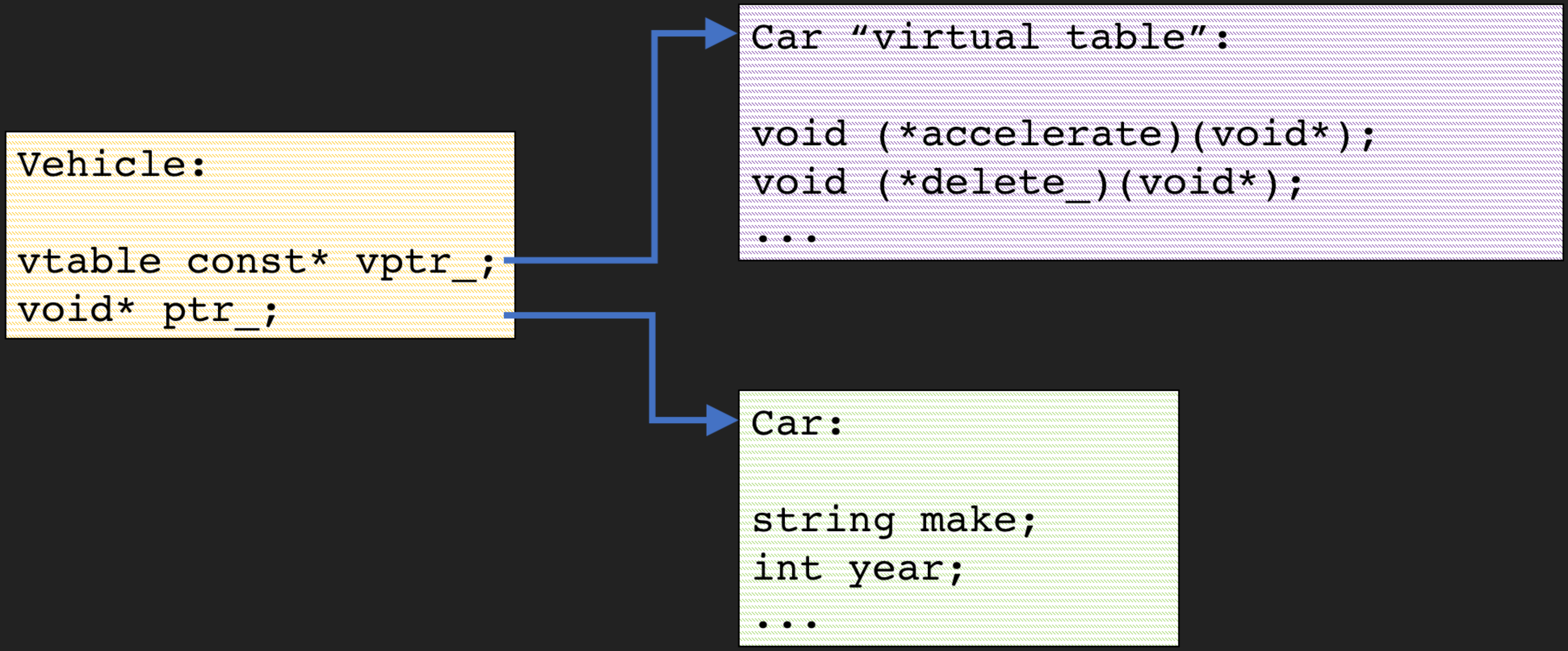
```
Car virtual table:

void (*accelerate)(Vehicle* __this);
void (*__dtor)(Vehicle* __this);
...
```

GOAL:

INDEPENDENT STORAGE AND METHOD DISPATCH

- Storage *policy*
- VTable *policy*

REMOTE STORAGE



```
Vehicle:

vtable const* vptr_;
void* ptr_;
```

```
Car "virtual table":

void (*accelerate)(void*);
void (*delete_)(void*);
...
```

```
Car:

string make;
int year;
...
```

HOW THAT'S IMPLEMENTED

```
class Vehicle {
  vtable const* const vptr_;
  void* ptr_;

public:
  template <typename Any>
    // enabled only when vehicle.accelerate() is valid
  Vehicle(Any vehicle)
    : vptr_{&vtable_for<Any>}
    , ptr_{new Any(vehicle)}
  { }

  Vehicle(Vehicle const& other); // implementation omitted

  void accelerate()
  { vptr_->accelerate(ptr_); }

  ~Vehicle()
  { vptr_->delete_(ptr_); }
};
```

THE VTABLE

```
struct vtable {
  void (*accelerate)(void* this_);
  void (*delete_)(void* this_);
};

template <typename T>
vtable const vtable_for = {
  [](void* this_) {
    static_cast<T*>(this_)->accelerate();
  },

  [](void* this_) {
    delete static_cast<T*>(this_);
  }
};
```

WITH DYNO

```
struct Vehicle {
  template <typename Any>
  Vehicle(Any vehicle) : poly_{vehicle} { }

  void accelerate()
  { poly_.virtual_("accelerate"_s)(poly_); }

private:
  dyno::poly<IVehicle, dyno::remote_storage> poly_;
  //                   ^^^^^^^^^^^^^^^^^^^^
};
```

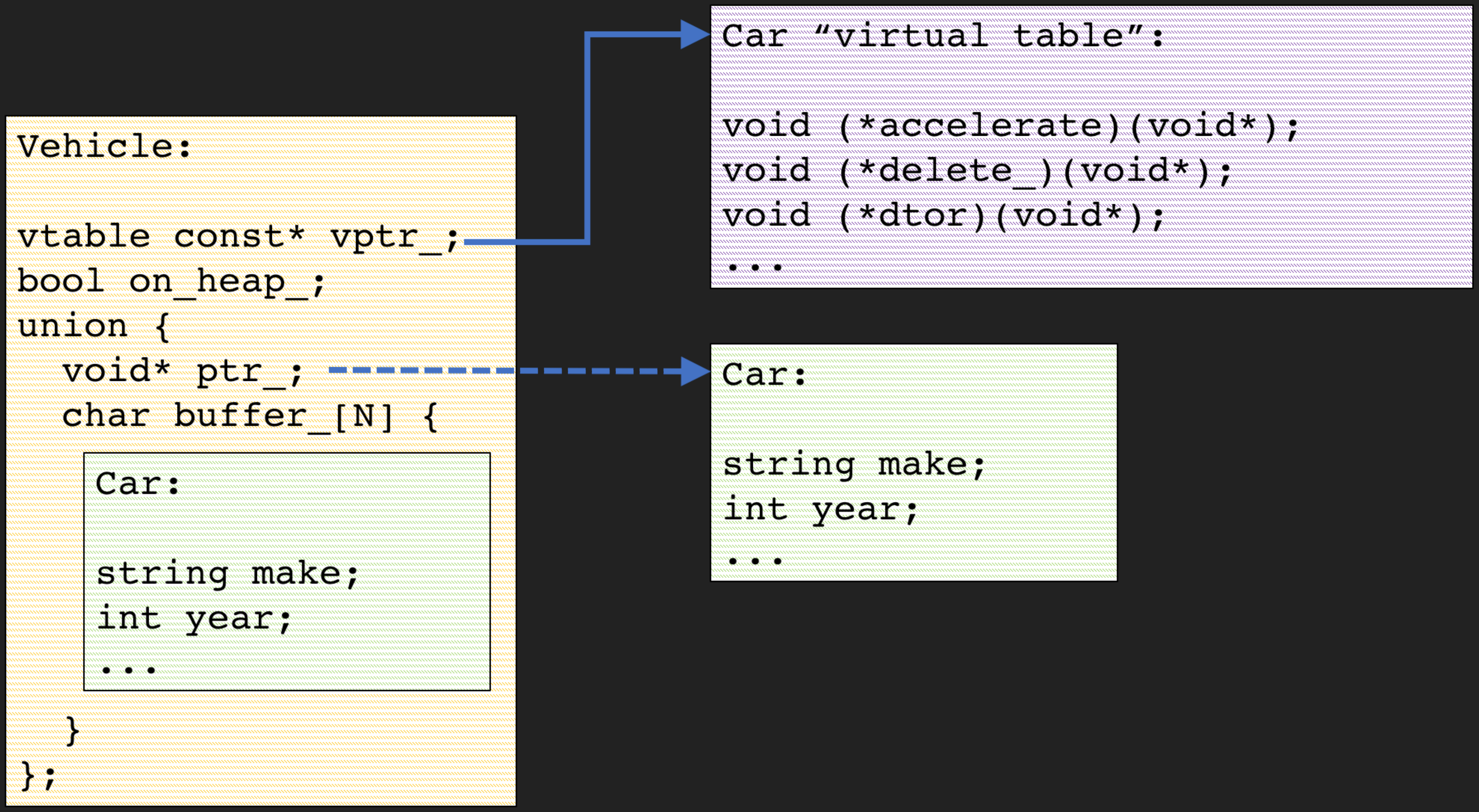
DYNO'S VTABLE

```cpp
struct IVehicle : decltype(dyno::requires(
  dyno::CopyConstructible{},
  dyno::Destructible{},
  "accelerate"_s = dyno::function<void(dyno::T&)>
)) { };

template <typename T>
auto dyno::default_concept_map<IVehicle, T> = dyno::make_concept_map(
  "accelerate"_s = [](T& vehicle) { vehicle.accelerate(); }
);
```

STRENGTHS AND WEAKNESSES

- ✓ Simple model, similar to classic inheritance
- ✗ Always requires an allocation

THE *SMALL BUFFER OPTIMIZATION* (SBO)



```
Vehicle:

vtable const* vptr_;
bool on_heap_;
union {
  void* ptr_;
  char buffer_[N] {
    Car:

    string make;
    int year;
    ...
  }
};
```

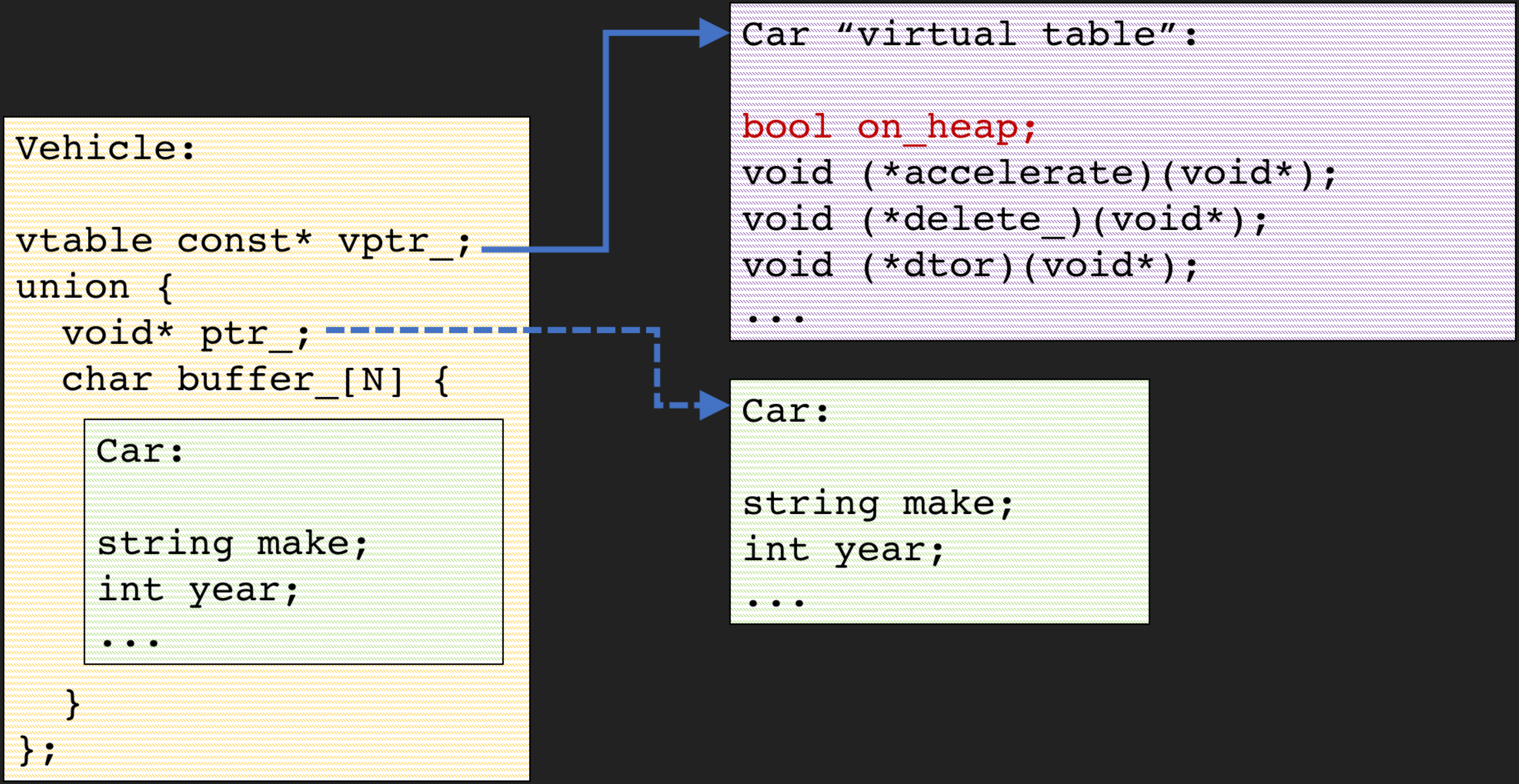
```
Car “virtual table”:

void (*accelerate)(void*);
void (*delete_)(void*);
void (*dtor)(void*);
...
```

```
Car:

string make;
int year;
...
```

HOW THAT'S IMPLEMENTED

```cpp
struct Vehicle {
  vtable const* const vptr_;
  union { void* ptr_;
          std::aligned_storage_t<16> buffer_; };
  bool on_heap_;

  template <typename Any>
  Vehicle(Any vehicle) : vptr_{&vtable_for<Any>} {
    if (sizeof(Any) > 16) {
      on_heap_ = true;
      ptr_ = new Any(vehicle);
    } else {
      on_heap_ = false;
      new (&buffer_) Any{vehicle};
    }
  }

  void accelerate()
  { vptr_->accelerate(on_heap_ ? ptr_ : &buffer_); }
};
```

ALTERNATIVE IMPLEMENTATION 1



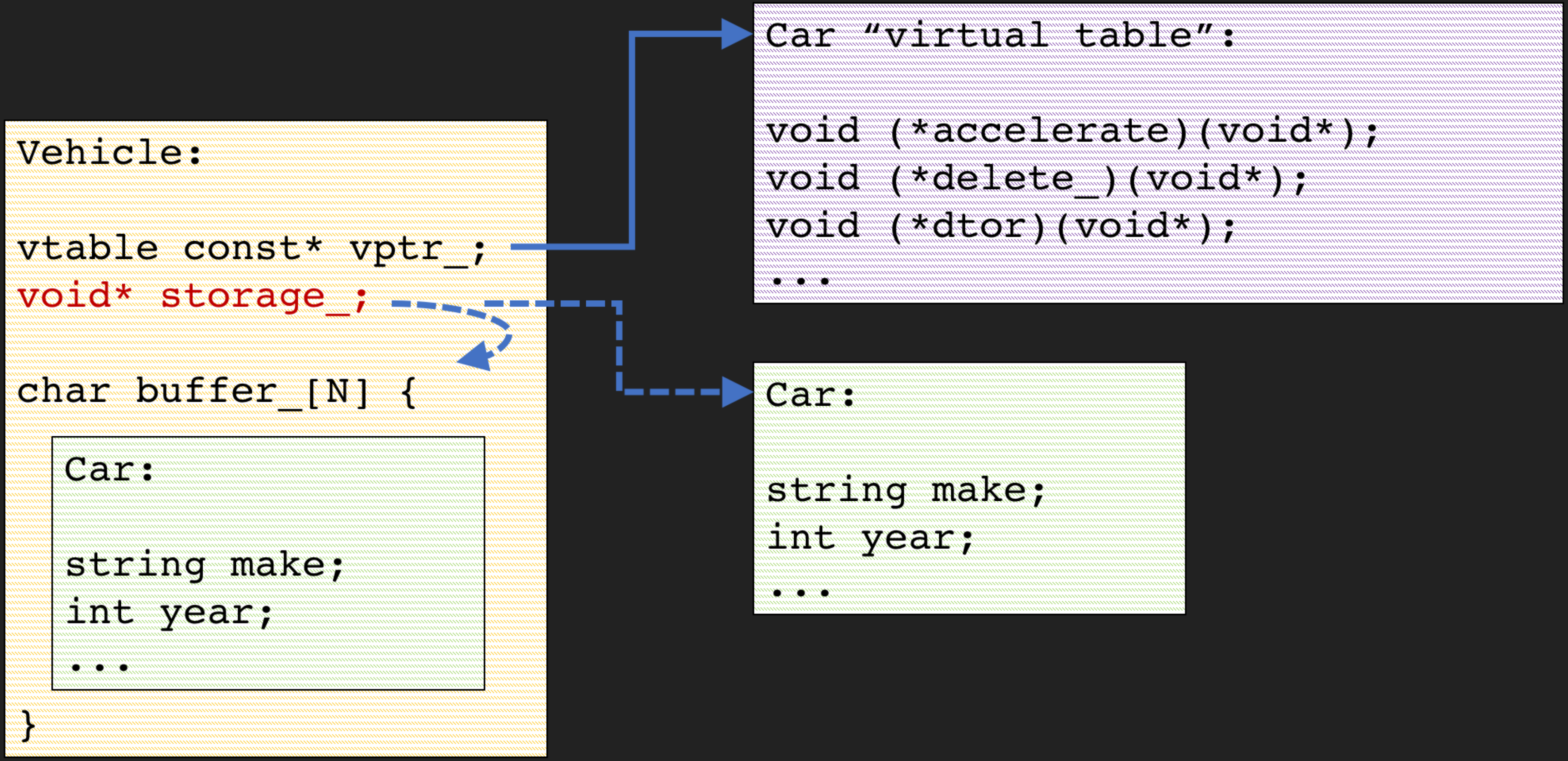
```
Vehicle:

vtable const* vptr_;
union {
   void* ptr_;
   char buffer_[N] {
     Car:

     string make;
     int year;
     ...
   }
};
```

```
Car “virtual table”:

bool on_heap;
void (*accelerate)(void*);
void (*delete_)(void*);
void (*dtor)(void*);
...
```

```
Car:

string make;
int year;
...
```

ALTERNATIVE IMPLEMENTATION 2

(seems to be the fastest)



```
Vehicle:

vtable const* vptr_;
void* storage_;

char buffer_[N] {
  Car:

  string make;
  int year;
  ...
}
```

```
Car "virtual table":

void (*accelerate)(void*);
void (*delete_)(void*);
void (*dtor)(void*);
...
```

```
Car:

string make;
int year;
...
```

WITH DYNO

```cpp
struct Vehicle {
  template <typename Any>
  Vehicle(Any vehicle) : poly_{vehicle} { }

  void accelerate()
  { poly_.virtual_("accelerate"_s)(poly_); }

private:
  dyno::poly<IVehicle, dyno::sbo_storage<16>> poly_;
  //                   ^^^^^^^^^^^^^^^^^^^^^
};
```

STRENGTHS AND WEAKNESSES

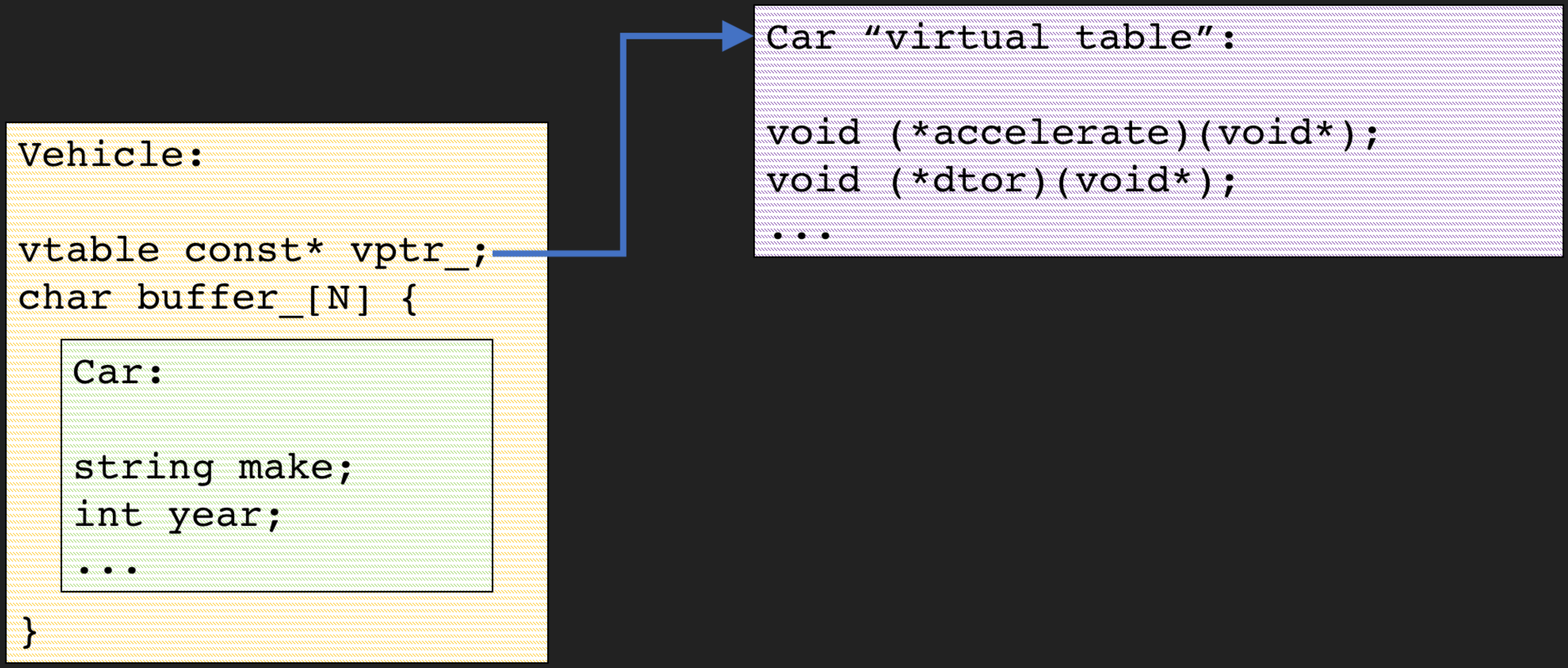
✓ Does not always require allocating

✗ Takes up more space

✗ Copy/move/swap is more complicated

✗ Dispatching may be more costly

ALWAYS-LOCAL STORAGE



```
Vehicle:

vtable const* vptr_;
char buffer_[N] {
   Car:

   string make;
   int year;
   ...
}
```

```
Car “virtual table”:

void (*accelerate)(void*);
void (*dtor)(void*);
...
```

DOESN'T FIT? DOESN'T COMPILE!

HOW THAT'S IMPLEMENTED

```cpp
class Vehicle {
  vtable const* const vptr_;
  std::aligned_storage_t<64> buffer_;

public:
  template <typename Any>
  Vehicle(Any vehicle) : vptr_{&vtable_for<Any>} {
    static_assert(sizeof(Any) <= sizeof(buffer_),
      "can't hold such a large object in a Vehicle");
    new (&buffer_) Any(vehicle);
  }

  void accelerate()
  { vptr_->accelerate(&buffer_); }

  ~Vehicle()
  { vptr_->dtor(&buffer_); }
};
```

WITH DYNO

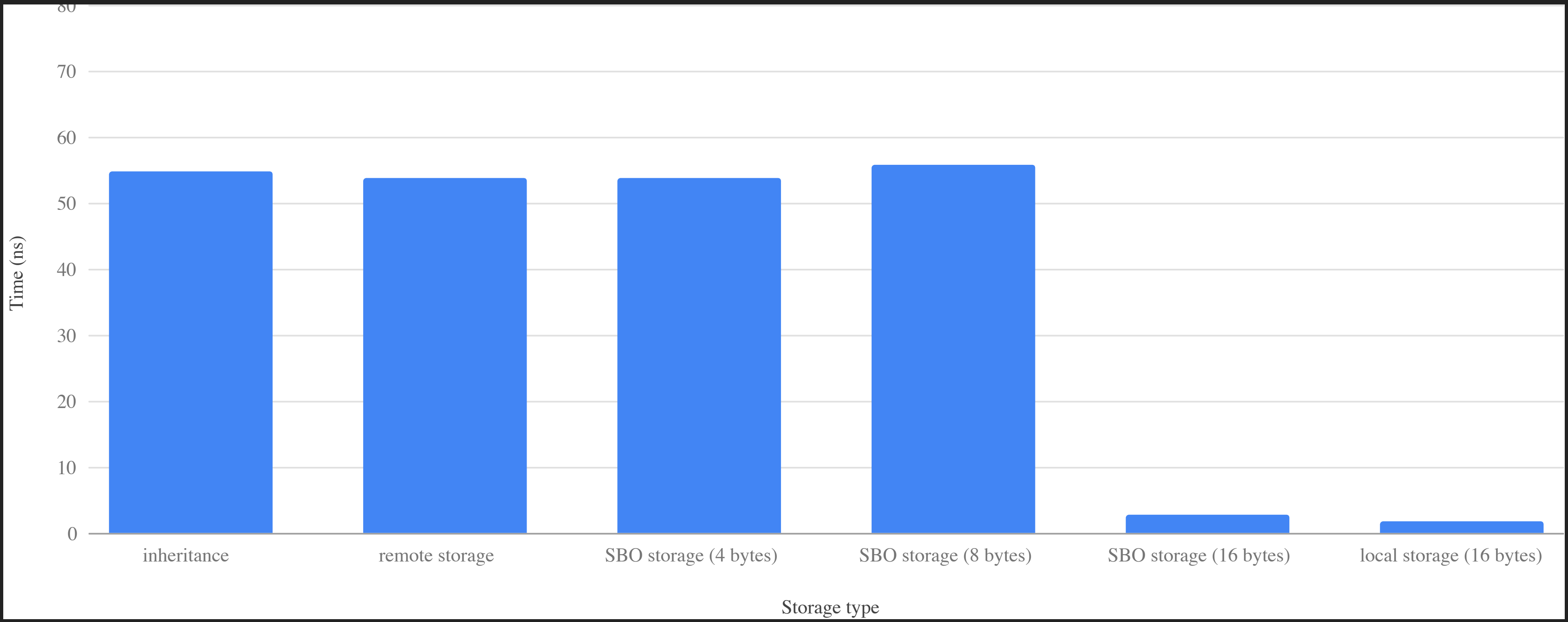
```
struct Vehicle {
  template <typename Any>
  Vehicle(Any vehicle) : poly_{vehicle} { }

  void accelerate()
  { poly_.virtual_("accelerate"_s)(poly_); }

private:
  dyno::poly<IVehicle, dyno::local_storage<64>> poly_;
  //                   ^^^^^^^^^^^^^^^^^^^^^^^
};
```

STRENGTHS AND WEAKNESSES

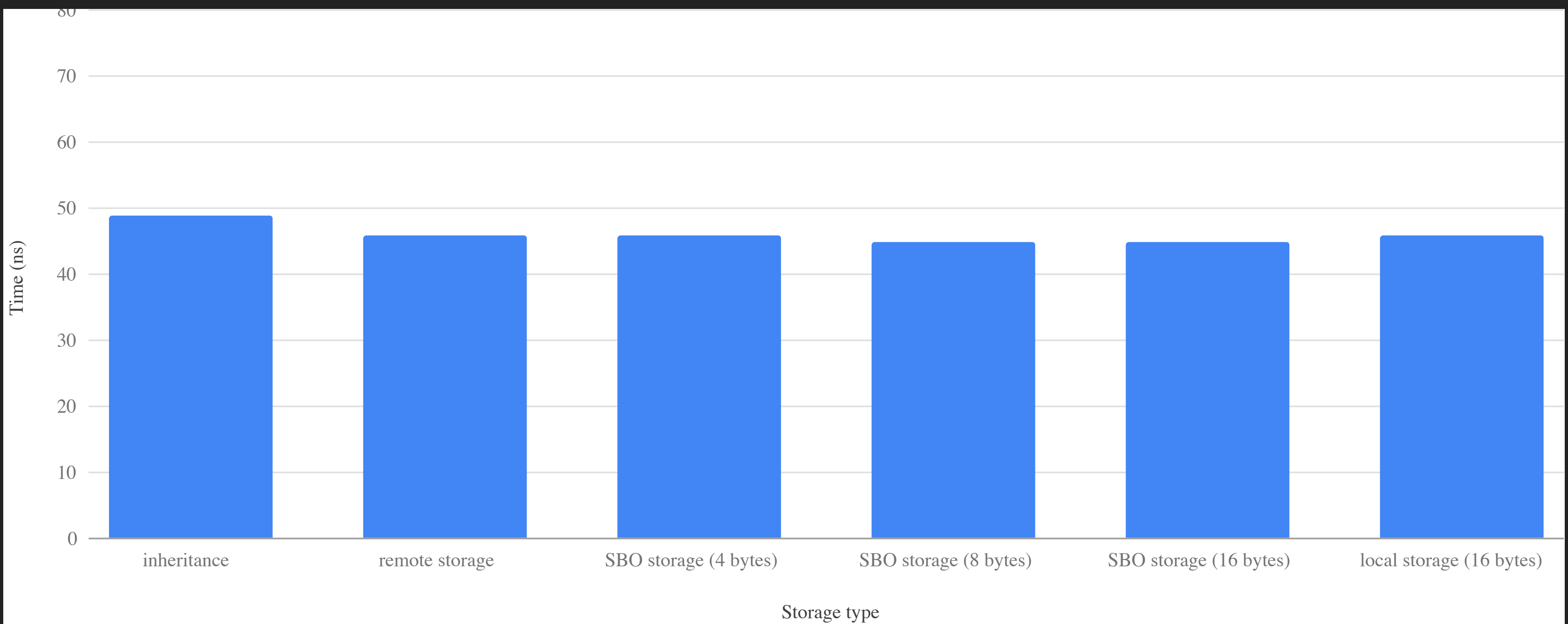
- ✓ No allocation – ever
- ✓ Simple dispatching
- ✗ Takes up more space

SOME BENCHMARKS

Creating many 16 bytes objects



Accessing many 4 bytes objects
(10 x 3 method calls, SBO with pointer)

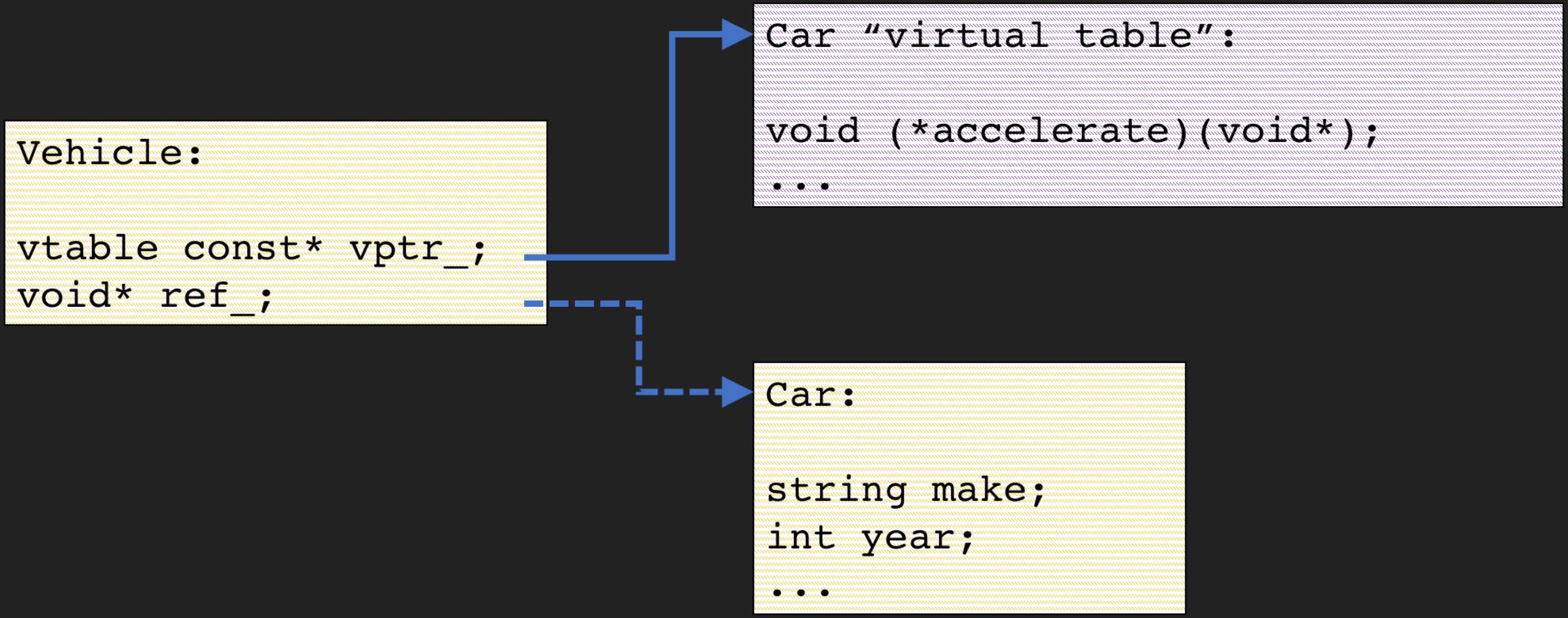


GUIDELINES

- Use local storage whenever you can afford it
- Otherwise, use SBO with the largest reasonable size
- Use purely-remote storage only when
  - Object sizes are so scattered SBO wouldn't help

NON-OWNING STORAGE

(reference semantics, not value semantics)



```
Vehicle:

vtable const* vptr_;
void* ref_;
```

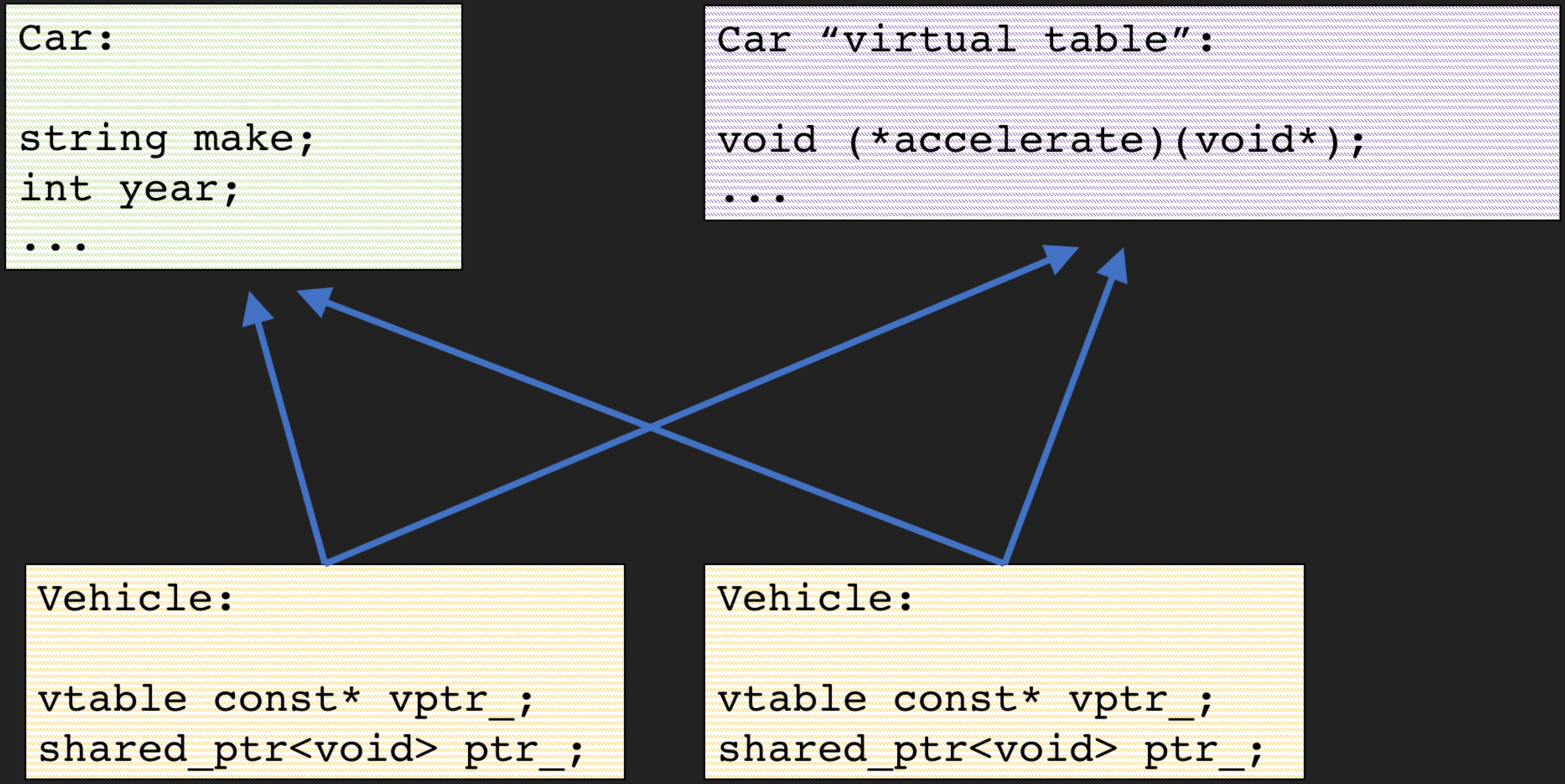
```
Car "virtual table":

void (*accelerate)(void*);
...
```

```
Car:

string make;
int year;
...
```

BASICALLY A POLYMORPHIC VIEW

```
void process(Vehicle vehicle) {
  ...
  vehicle.accelerate();
  ...
}

int main() {
  Truck truck{...};
  process(truck); // No copy!
}
```

SHARED REMOTE STORAGE



```
Car:

string make;
int year;
...
```

```
Car “virtual table”:

void (*accelerate)(void*);
...
```

```
Vehicle:

vtable const* vptr_;
shared_ptr<void> ptr_;
```

```
Vehicle:

vtable const* vptr_;
shared_ptr<void> ptr_;
```

NOW, LET ME SHOW YOU WHY YOU CARE

HAVE YOU HEARD OF THE FOLLOWING?

- `std::function`
- `inplace_function`
- `function_view`

CONSIDER THIS

```cpp
template <typename Signature, typename StoragePolicy>
struct basic_function;

template <typename R, typename ...Args, typename StoragePolicy>
struct basic_function<R(Args...), StoragePolicy> {
  template <typename F>
  basic_function(F&& f) : poly_{std::forward<F>(f)} { }

  R operator()(Args ...args) const
  { return poly_.virtual_("call"_s)(poly_, args...); }

private:
  dyno::poly<Callable<R(Args...)>, StoragePolicy> poly_;
};
```

HERE'S ALL OF THEM:

```
template <typename Signature>
using function = basic_function<Signature,
                                dyno::sbo_storage<16>>;
```

```
template <typename Signature, std::size_t Size = 32>
using inplace_function = basic_function<Signature,
                                        dyno::local_storage<Size>>;
```

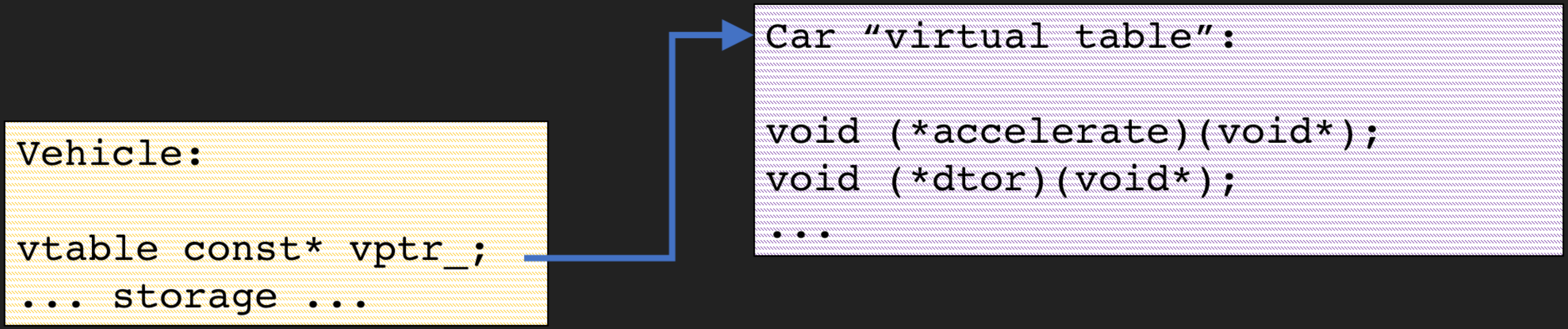
```
template <typename Signature>
using function_view = basic_function<Signature,
                                     dyno::non_owning_storage>;
```

```
template <typename Signature>
using shared_function = basic_function<Signature,
                                       dyno::shared_remote_storage>;
```

WE'VE TALKED ABOUT STORAGE

WHAT ABOUT VTABLES?

NORMALLY, IT IS REMOTE

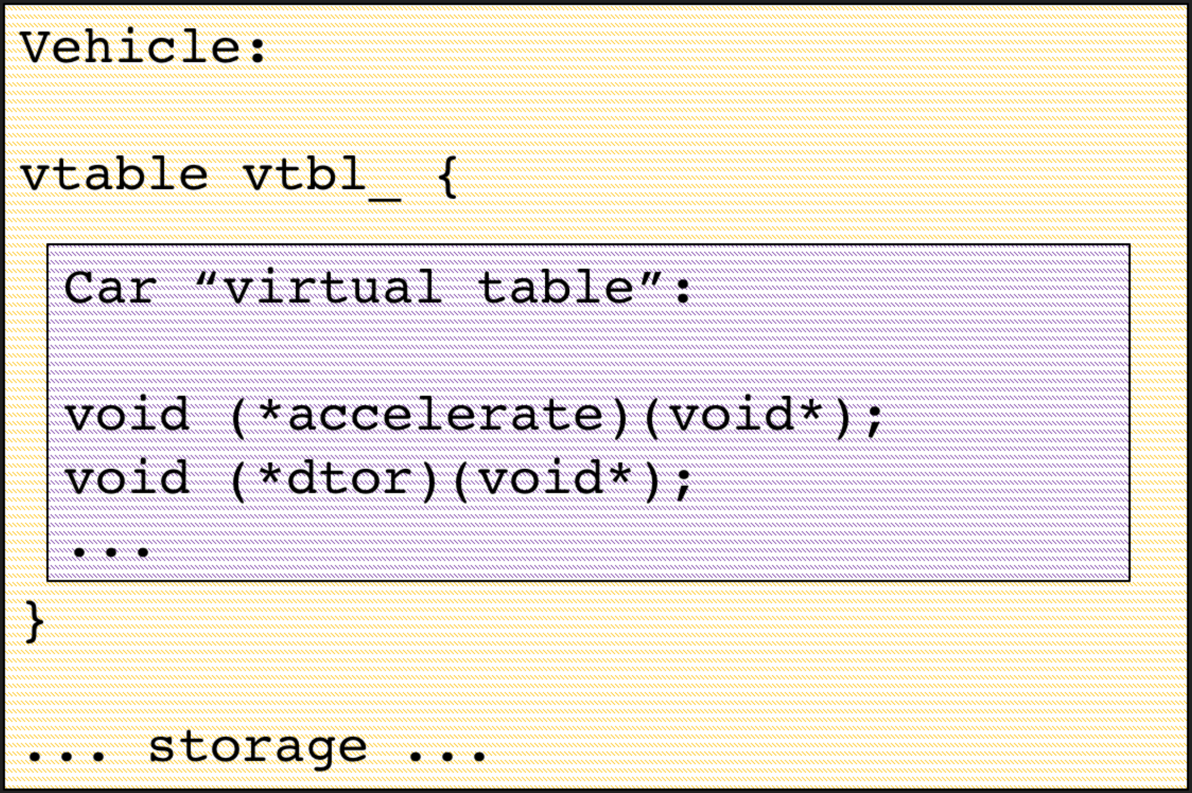


```
Vehicle:

vtable const* vptr_;
... storage ...
```

```
Car “virtual table”:

void (*accelerate)(void*);
void (*dtor)(void*);
...
```

TURNS OUT WE HAVE SOME CHOICES

INLINING THE VTABLE IN THE OBJECT



```
Vehicle:

vtable vtbl_ {

  Car “virtual table”:

  void (*accelerate)(void*);
  void (*dtor)(void*);
  ...
}

... storage ...
```

HOW THAT'S IMPLEMENTED

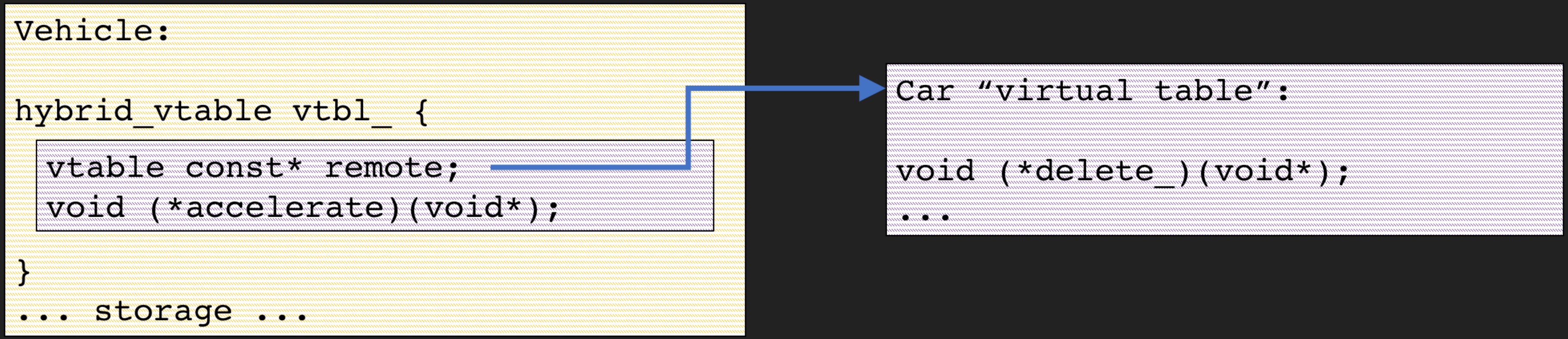
```
struct Vehicle {
  template <typename Any>
  Vehicle(Any vehicle)
    : vtbl_{vtable_for<Any>}
    , ptr_{new Any(vehicle)}
  { }

  void accelerate()
  { vtbl_.accelerate(ptr_); }

  ~Vehicle()
  { vtbl_.delete_(ptr_); }

private:
  vtable const vtbl_; // <= not a pointer!
  void* ptr_;
};
```

WITH DYNO

```
struct Vehicle {
  template <typename Any>
  Vehicle(Any vehicle) : poly_{vehicle} { }

  void accelerate()
  { poly_.virtual_("accelerate"_s)(poly_); }

private:
  using VTable = dyno::vtable<dyno::local<dyno::everything>>;
  //             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  dyno::poly<IVehicle, dyno::remote_storage, VTable> poly_;
};
```

USUALLY A PESSIMIZATION

PARTIAL VTABLE INLINING



```
Vehicle:

hybrid_vtable vtbl_ {
  vtable const* remote;
  void (*accelerate)(void*);
}
... storage ...
```

```
Car “virtual table”:

void (*delete_)(void*);
...
```

THE VTABLE — REMOTE PART

```
struct vtable {
  void (*delete_)(void* this_);
  // ...
};

template <typename T>
vtable const vtable_for = {
  [](void* this_) {
    delete static_cast<T*>(this_);
  }
  // ...
};
```

THE VTABLE — LOCAL PART

```
struct joined_vtable {
  vtable const* const remote;
  void (*accelerate)(void* this_);
};

template <typename T>
joined_vtable const joined_vtable_for = {
  &vtable_for<T>,
  [](void* this_) {
    static_cast<T*>(this_)->accelerate();
  }
};
```

THE POLYMORPHIC WRAPPER

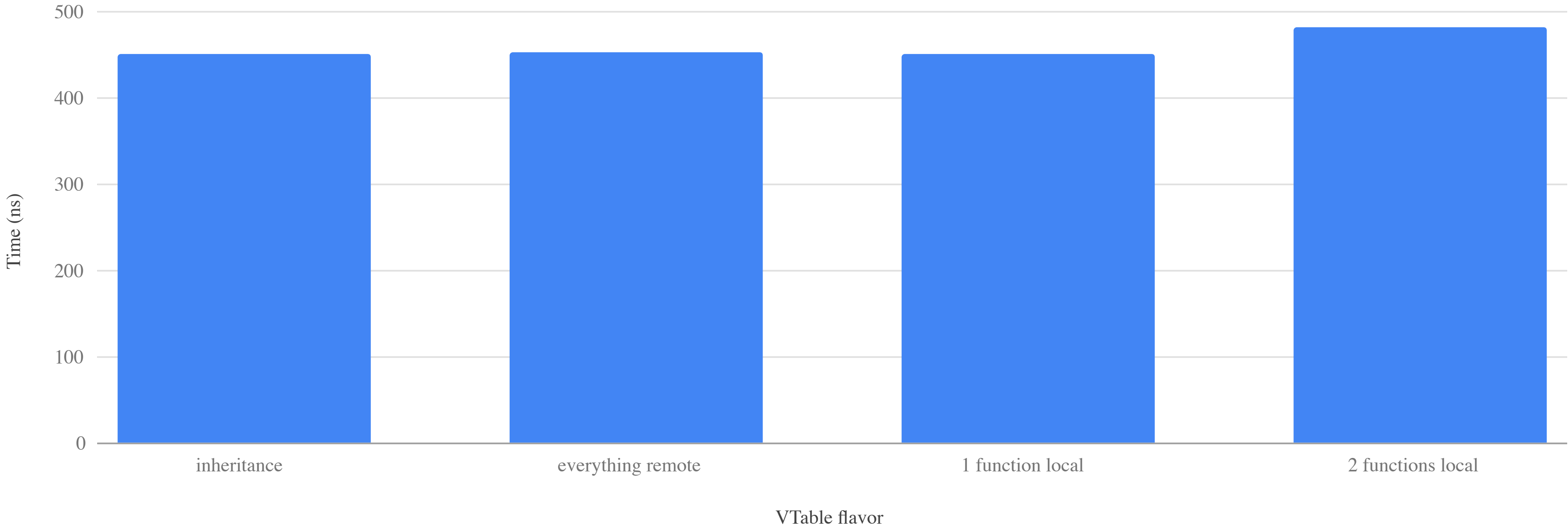
```
class Vehicle {
  joined_vtable const vtbl_;
  void* ptr_;

public:
  template <typename Any>
  Vehicle(Any vehicle)
    : vtbl_{joined_vtable_for<Any>}
    , ptr_{new Any(vehicle)}
  { }

  void accelerate()
  { vtbl_.accelerate(ptr_); }

  ~Vehicle()
  { vtbl_.remote->delete_(ptr_); }
};
```

WITH DYNO

```
struct Vehicle {
  template <typename Any>
  Vehicle(Any vehicle) : poly_{vehicle} { }

  void accelerate()
  { poly_.virtual_("accelerate"_s)(poly_); }

private:
  using VTable = dyno::vtable<
                  dyno::local<dyno::only<decltype("accelerate"_s)>>,
                  dyno::remote<dyno::everything_else>>;
  //              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  dyno::poly<IVehicle, dyno::remote_storage, VTable> poly_;
};
```

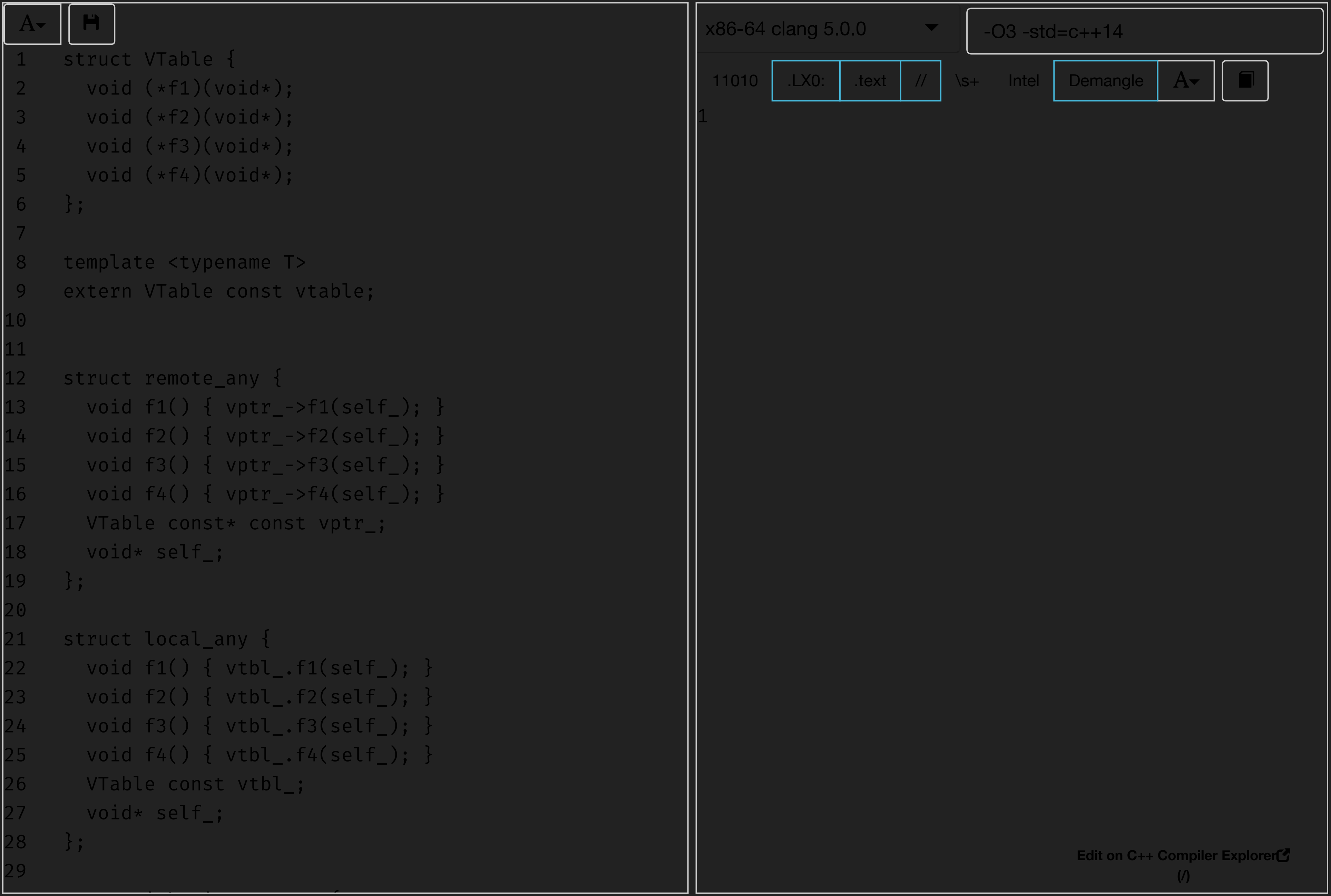
SOME BENCHMARKS

Calling 2 virtual functions (100 x 2 calls)



NOT VERY CONCLUSIVE

LET'S LOOK AT ASSEMBLY

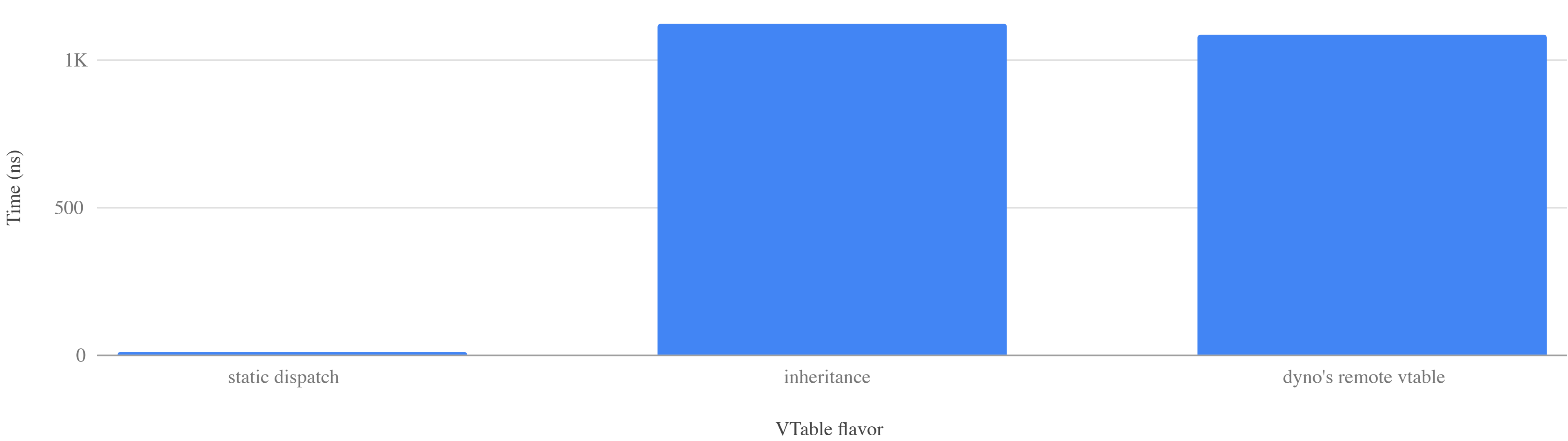


-O3 -std=c++14

```
struct VTable {
  void (*f1)(void*);
  void (*f2)(void*);
  void (*f3)(void*);
  void (*f4)(void*);
};

template <typename T>
extern VTable const vtable;


struct remote_any {
  void f1() { vptr_->f1(self_); }
  void f2() { vptr_->f2(self_); }
  void f3() { vptr_->f3(self_); }
  void f4() { vptr_->f4(self_); }
  VTable const* const vptr_;
  void* self_;
};

struct local_any {
  void f1() { vtbl_.f1(self_); }
  void f2() { vtbl_.f2(self_); }
  void f3() { vtbl_.f3(self_); }
  void f4() { vtbl_.f4(self_); }
  VTable const vtbl_;
  void* self_;
};

```

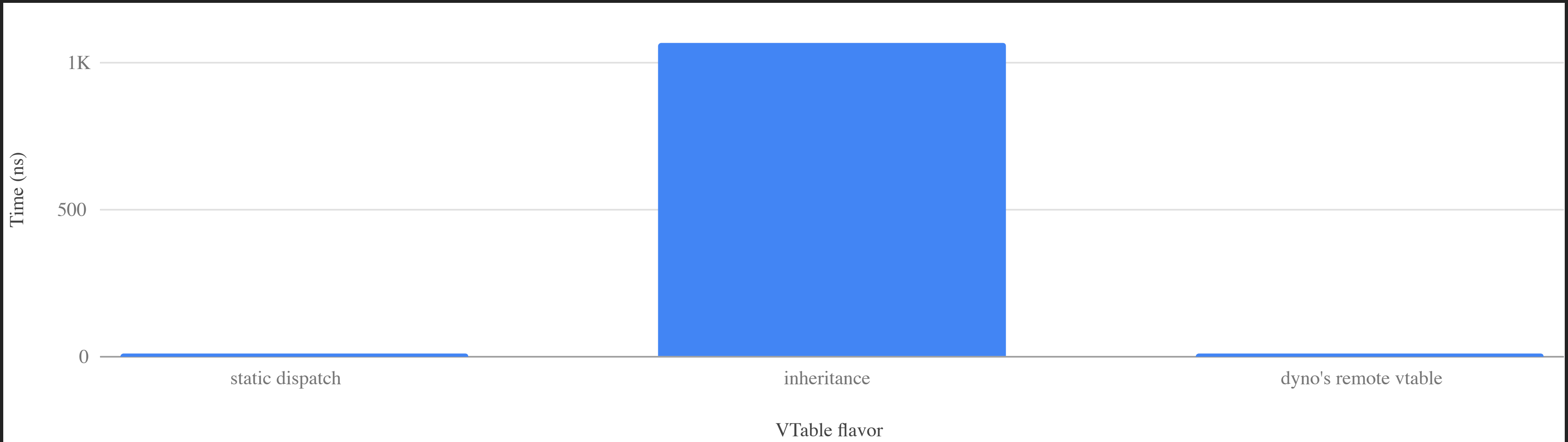
A STORY OF INLINING

```
template <typename AnyIterator, typename It>
__attribute__((noinline)) AnyIterator make(It it) {
  return AnyIterator{std::move(it)};
}

template <typename AnyIterator>
void benchmark_any_iterator(benchmark::State& state) {
  std::vector<int> input{...};
  std::vector<int> output{...};

  while (state.KeepRunning()) {
    auto first = make<AnyIterator>(input.begin());
    auto last = make<AnyIterator>(input.end());
    auto result = make<AnyIterator>(output.begin());

    for (; !(first == last); ++first, ++result) {
      *result = *first;
    }
  }
}
```


1K
500
0
Time (ns)
static dispatch
inheritance
dyno's remote vtable
VTable flavor

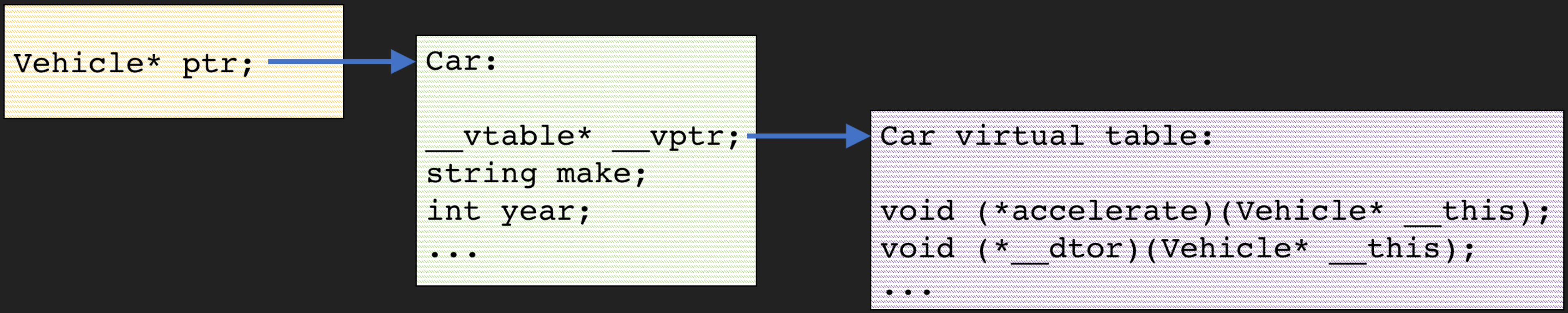
NOW, JUST A SMALL TWEAK

```
template <typename AnyIterator, typename It>
// __attribute__((noinline))
AnyIterator make(It it) {
  return AnyIterator{std::move(it)};
}
```


1K
500
0
Time (ns)
static dispatch
inheritance
dyno's remote vtable
VTable flavor

WHAT HAPPENED?

Inheritance:

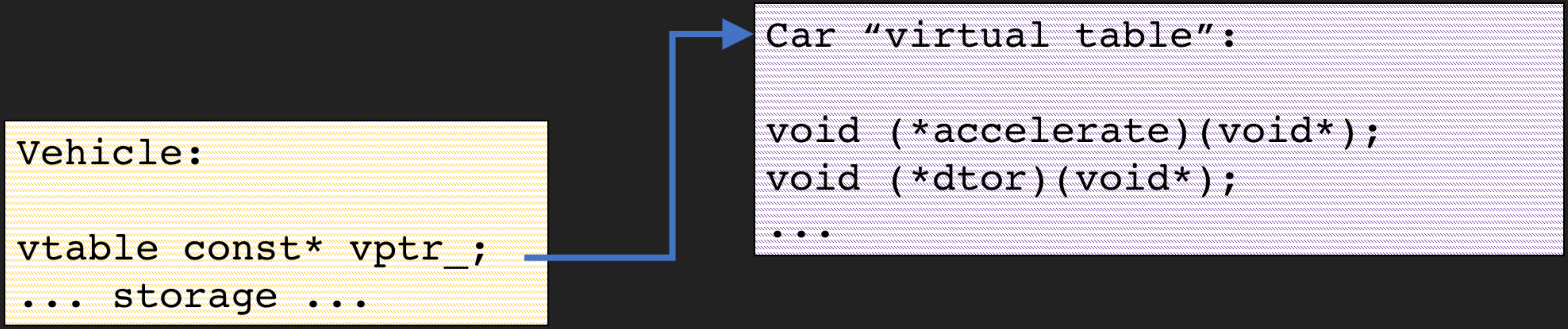


```
Vehicle* ptr;
```

```
Car:

__vtable* __vptr;
string make;
int year;
...
```

```
Car virtual table:

void (*accelerate)(Vehicle* __this);
void (*__dtor)(Vehicle* __this);
...
```

Dyno's remote vtable:



```
Vehicle:

vtable const* vptr_;
... storage ...
```

```
Car “virtual table”:

void (*accelerate)(void*);
void (*dtor)(void*);
...
```

WHAT'S THE LESSON?

- Reducing pointer hops can lead to unexpected inlining
- When that happens, giant optimizations become possible

GUIDELINES

- By default, all methods are in the remote vtable
- Consider inlining some methods if
    - you have slack space
    - you know you're calling them often
    - you've measured it makes a difference

THE FUTURE WITH REFLECTION?

```
struct Vehicle {
  void accelerate();
};

int main() {
  std::vector<poly<Vehicle>> vehicles;
  vehicles.push_back(Car{...});
  vehicles.push_back(Truck{...});
  vehicles.push_back(lib::Motorcycle{...});

  for (auto& vehicle : vehicles) {
    vehicle.accelerate();
  }
}
```

THE FUTURE WITH METACLASSES?

```
interface Vehicle {
  void accelerate();
};

int main() {
  std::vector<Vehicle> vehicles;
  vehicles.push_back(Car{...});
  vehicles.push_back(Truck{...});
  vehicles.push_back(lib::Motorcycle{...});

  for (auto& vehicle : vehicles) {
    vehicle.accelerate();
  }
}
```

SUMMARY

- Inheritance model is just one option amongst others
  - Don't bake that choice in
- Many ways of storing polymorphic objects
  - As always, space/time tradeoff
- Vtables can be inlined (measure!)
- Type erasure is tedious to do manually
  - Reflection will be there to help

THE DYNO LIBRARY IS AVAILABLE

https://github.com/ldionne/dyno

USEFUL LINKS AND RELATED MATERIAL

- Sean Parent's NDC 2017 talk:
  https://youtu.be/QGcVXgEVMJg
- Zach Laine's CppCon 2014 talk:
  https://youtu.be/0I0FD3N5cgM
- Boost.TypeErasure:
  http://www.boost.org/doc/libs/release/doc/html/boost_typeerasure.html
- Adobe Poly:
  https://stlab.adobe.com/group__poly__related.html
- Eraserface:
  https://github.com/badair/eraserface
- liberasure:
  https://github.com/atomgalaxy/liberasure
- 2004 thread on interfaces:
  https://goo.gl/zaBN6X

THANK YOU

https://ldionne.com