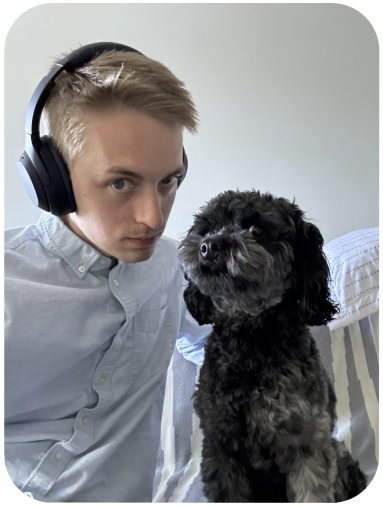
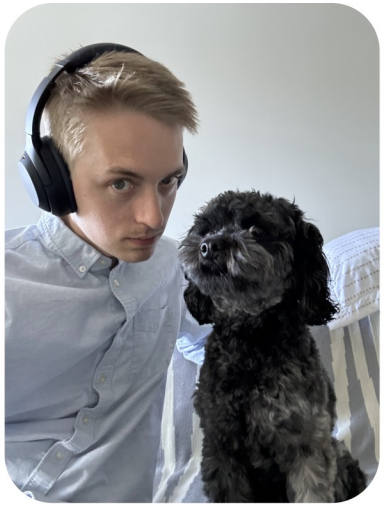


Tor Shepherd

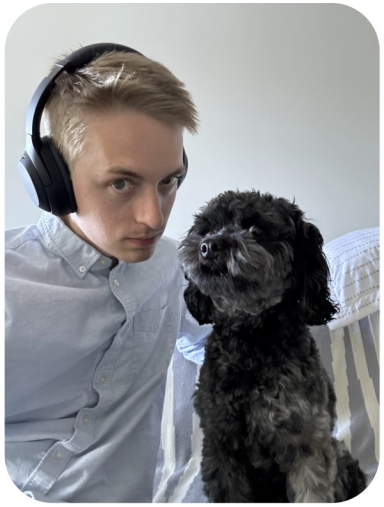
Motion Planning
@ Motional



Tor Shepherd

Motion Planning
@ Motional

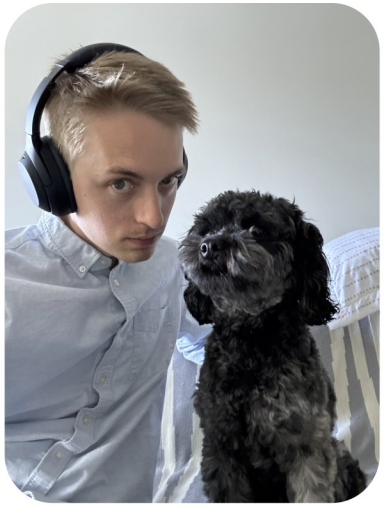
```
1 // Why do we use C++ ?
```



Tor Shepherd

Motion Planning
@ Motional

```
1 // Why do we use C++ ?
2
3 // Convenient?
4 // Fast?
```



Tor Shepherd

Motion Planning
@ Motional

```
1 // Why do we use C++ ?
2
3 // Convenient to write
4 // Fast code
```

Part I

Basic types of copies

Part I

Basic types of copies

Part II

Everything from Part I was wrong

Part I	Part II	Part III
Basic types of copies	Everything from Part I was wrong	What can we do about it

Let’s write some C 

Scalar copies

```
1 void foo() {
2   int i = 2;
3   int other = i;
4   return;
5 }
```

Scalar copies

```
1 void foo() {
2   int i = 2;
3   int other = i;
4   return;
5 }
```

Scalar copies

```
1 void foo() {
2   int i = 2;
3   int other = i;
4   return;
5 }
```

Scalar copies

```
1 void foo() {
2   int i = 2;
3   int other = i;
4   return;
5 }
```

Struct (trivial) copies

```
1  struct S {
2    int i;
3    double j;
4    char k;
5  };
6
7  void foo() {
8    struct S object = {2, 3.14, 'n'};
9    struct S other = object; // ?
10   return;
11 }
```

Struct (trivial) copies

```
1  struct S {
2    int i;
3    double j;
4    char k;
5  };
6
7  void foo() {
8    struct S object = {2, 3.14, 'n'};
9    struct S other = object; // ?
10   return;
11 }
```

Struct (trivial) copies

```
1  struct S {
2    int i;
3    double j;
4    char k;
5  };
6
7  void foo() {
8    struct S object = {2, 3.14, 'n'};
9    struct S other;
10   other.i = object.i;
11   other.j = object.j;
12   other.k = object.k;
13   return;
14 }
```

Vector in C

```c
struct vector {
  int* data;
  size_t size;
  size_t capacity;
};

void foo() {
  struct vector v = {NULL, 0U, 0U};
  vector_push_back(&v, 33);
  return; // oops!
}
```

Vector in C

```
1  struct vector {
2    int* data;
3    size_t size;
4    size_t capacity;
5  };
6
7  void foo() {
8    struct vector v = {NULL, 0U, 0U};
9    vector_push_back(&v, 33);
10   return; // oops!
11 }
```

Vector in C

```c
struct vector {
  int* data;
  size_t size;
  size_t capacity;
};

void foo() {
  struct vector v = {NULL, 0U, 0U};
  vector_push_back(&v, 33);
  return; // oops!
}
```

Vector in C

```
1  struct vector {
2    int* data;
3    size_t size;
4    size_t capacity;
5  };
6
7  void foo() {
8    struct vector v = {NULL, 0U, 0U};
9    vector_push_back(&v, 33);
10   return; // oops!
11 }
```

Vector in C

```c
struct vector {
  int* data;
  size_t size;
  size_t capacity;
};

void foo() {
  struct vector v = {NULL, 0U, 0U};
  vector_push_back(&v, 33);
  // Memory leak if we don't free!
  free(v.data);
  return;
}
```

Vector in C++

```
1 void foo() {
2   vector<int> v{}; // ?
3   v.push_back(33);
4   // ?
5   return;
6 }
```

"Construction"

```
1 void foo() {
2   // allocate the stack memory
3   char alloc[sizeof(vector<int>)];
4   new (&alloc) vector<int>();
5   auto v = (vector<int>*) &alloc;
6   v.push_back(33);
7   // ?
8   return;
9 }
```

"Destruction"

```
1  void foo() {
2    // allocate the stack memory
3    char alloc[sizeof(vector<int>)];
4    new (&alloc) vector<int>();
5    auto v = (vector<int>*) &alloc;
6    v.push_back(33);
7    v.~vector<int>();
8    // release the stack memory
9    return;
10 }
```

Vector in C++

```
1 void foo() {
2   vector<int> v{};
3   v.push_back(33);
4   return;
5 }
```

Wait...

```
1 void foo() {
2   vector<int> v{};
3   v.push_back(33);
4   vector<int> other = v;
5   return;
6 }
```

Can we just trivial copy?

```
1 void foo() {
2   vector<int> v{};
3   v.push_back(33);
4   // pseudocode
5   vector<int> other;
6   other.data = v.data
7   // ...
8   return;
9 }
```

Can we just trivial copy?

```
1 void foo() {
2   vector<int> v{};
3   v.push_back(33);
4   // pseudocode
5   vector<int> other;
6   other.data = v.data
7   // ...
8   return;
9 }
```

Can we just trivial copy?

```
1 void foo() {
2   vector<int> v{};
3   v.push_back(33);
4   // pseudocode
5   vector<int> other;
6   other.data = v.data
7   // ...
8   return;
9 }
```

Can we just trivial copy?

```
1 void foo() {
2   vector<int> v{};
3   v.push_back(33);
4   // pseudocode
5   vector<int> other;
6   other.data = v.data
7   // ...
8   return;
9 }
```

Can we just trivial copy?

```
1  void foo() {
2    vector<int> v{};
3    v.push_back(33);
4    // pseudocode
5    vector<int> other;
6    other.data = v.data
7    // ...
8    // other.~vector<int>() frees other.data
9    // v.~vector<int>() frees v.data again!
10   return;
11 }
```

Deep copies

```
void foo() {
  vector<int> v{};
  v.push_back(33);
  // vector<int> other = v; expands to:
  char alloc[sizeof(vector<int>)];
  new (&alloc) vector<int>(v);
  auto other = (vector<int>*) &alloc;
  return;
}
```

Deep copies

```
1  void foo() {
2    vector<int> v{};
3    v.push_back(33);
4    // vector<int> other = v; expands to:
5    char alloc[sizeof(vector<int>)];
6    // (just like in C, the source is
7    // supposed to be unchanged!)
8    new (&alloc) vector<int>(v);
9    auto other = (vector<int>*) &alloc;
10   return;
11 }
```

Is this always what we want?

Vector would copy on resize 

```
1 void foo() {
2   vector<vector<int>> v{};
3   v.emplace_back();
4   v.back().push_back(33);
5   v.emplace_back();
6   return;
7 }
```

Vector would copy on resize

```cpp
void foo() {
  vector<vector<int>> v{};
  v.emplace_back();
  v.back().push_back(33);
  v.emplace_back();
  return;
}
```

Vector would copy on resize

```
1 void foo() {
2   vector<vector<int>> v{};
3   v.emplace_back();
4   v.back().push_back(33);
5   v.emplace_back();
6   return;
7 }
```

Vector would copy on resize 

```
1 void foo() {
2   vector<vector<int>> v{};
3   v.emplace_back();
4   v.back().push_back(33);
5   v.emplace_back();
6   return;
7 }
```

Vector would copy on resize

```
1 void foo() {
2   vector<vector<int>> v{};
3   v.emplace_back();
4   v.back().push_back(33);
5   v.emplace_back();
6   return;
7 }
```

"Move semantics"

```
class vector {
  // Copy constructor
  vector(const vector& other) {
    this->data = malloc(other.size * ...);
    for (...) { this->data[i] = other.data[i]; }
  }
}
```

"Move semantics"

```
class vector {
  // Copy constructor
  vector(const vector& other) {
    this->data = malloc(other.size * ...);
    for (...) { this->data[i] = other.data[i]; }
  }
  // Move constructor
  vector(/* ?? */ other) {
    this->data = other.data;
  }
}
```

"Move semantics"

```
class vector {
  // Copy constructor
  vector(const vector& other) {
    this->data = malloc(other.size * ...);
    for (...) { this->data[i] = other.data[i]; }
  }
  // Move constructor
  vector(/* ?? */ other) {
    this->data = other.data;
    other.data = nullptr;
  }
}
```

"Move semantics"

```
class vector {
  // Copy constructor
  vector(const vector& other) {
    this->data = malloc(other.size * ...);
    for (...) { this->data[i] = other.data[i]; }
  }
  // Move constructor
  vector(vector&& other) {
    this->data = other.data;
    other.data = nullptr;
  }
}
```

"Move semantics"

```
1 void foo() {
2   vector<int> v{};
3   v.push_back(33);
4   vector<int> copied = v;
5   return;
6 }
```

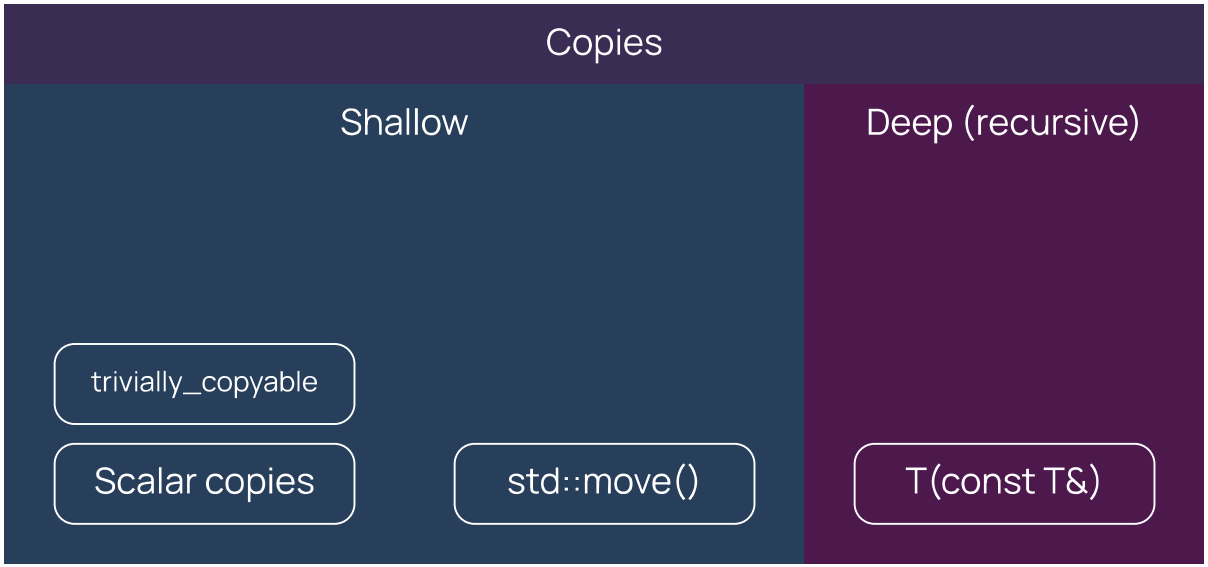
"Move semantics"

```
1 void foo() {
2   vector<int> v{};
3   v.push_back(33);
4   vector<int> copied = v;
5   vector<int> moved =
6     static_cast<vector<int>&&>(v);
7   return;
8 }
```

"Move semantics"

```
1 void foo() {
2   vector<int> v{};
3   v.push_back(33);
4   vector<int> copied = v;
5   vector<int> moved = std::move(v);
6   return;
7 }
```

Family tree of copies



Part II: So that's it?

Part II: So that's it?

Move Semantics 💔 Exceptions

```
1 struct S {
2   std::vector<int> data;
3 };
```

Move Semantics 💔 Exceptions

```
1 struct S {
2   S(const S&) = default;
3   S(S&&) = default;
4   std::vector<int> data;
5 };
```

Move Semantics 💔 Exceptions

```cpp
struct S {
  S(const S&) = default;
  S(S&&) = default;
  std::vector<int> data;
};

void foo() {
  vector<S> v{};
  v.emplace_back();
  v.back().data.push_back(33);
  v.emplace_back();
  // moves v[0] over with S(S&&)
  return;
}
```

Move Semantics 💔 Exceptions

```
1  struct S {
2    S(const S&) = default;
3    S(S&&) = default;
4    std::vector<int> data;
5  };
6
7  void foo() {
8    vector<S> v{};
9    v.emplace_back();
10   v.back().data.push_back(33);
11   v.emplace_back();
12   // moves v[0] over with S(S&&)
13   return;
14 }
```

Move Semantics 💔 Exceptions

```
1  struct S {
2    S(const S&) = default;
3    S(S&&);
4    std::vector<int> data;
5  };
6
7  void foo() {
8    vector<S> v{};
9    v.emplace_back();
10   v.back().data.push_back(33);
11   v.emplace_back();
12   // ???
13   return;
14 }
```

Move Semantics 💔 Exceptions

```
1  struct S {
2    S(const S&) = default;
3    S(S&&);
4    std::vector<int> data;
5  };
6
7  void foo() {
8    vector<S> v{};
9    v.emplace_back();
10   v.back().data.push_back(33);
11   v.emplace_back();
12   // ???
13   return;
14 }
```

Move Semantics 💔 Exceptions

```
1  struct S {
2    S(const S&) = default;
3    S(S&&);
4    std::vector<int> data;
5  };
6
7  void foo() {
8    vector<S> v{};
9    v.emplace_back();
10   v.back().data.push_back(33);
11   v.emplace_back();
12   // Deep copies??? Why???
13   return;
14 }
```

Move Semantics 💔 Exceptions

```
1 struct S {
2   S(const S&) noexcept = default;
3   S(S&&) noexcept(false);
4   // Assumes this may throw an exception...
5   std::vector<int> data;
6 };
```

Exception guarantees

std::vector<T,Allocator>::emplace_back

```
template< class... Args >                                  (since C++11)
void emplace_back( Args&&... args );                       (until C++17)

template< class... Args >                                  (since C++17)
reference emplace_back( Args&&... args );                  (until C++20)

template< class... Args >
constexpr reference emplace_back( Args&&... args );        (since C++20)
```

Appends a new element to the end of the container. The element is constructed through `std::allocator_traits::construct`, which typically uses placement-new to construct the element in-place at the location provided by the container. The arguments `args...` are forwarded to the constructor as `std::forward<Args>(args)...`.

If after the operation the new `size()` is greater than old `capacity()` a reallocation takes place, in which case all iterators (including the `end()` iterator) and all references to the elements are invalidated. Otherwise only the `end()` iterator is invalidated.

Parameters

args - arguments to forward to the constructor of the element

Type requirements

- `T (the container's element type)` must meet the requirements of *MoveInsertable* and *EmplaceConstructible*.

Return value

(none) (until C++17)
A reference to the inserted element. (since C++17)

std::move_if_noexcept

std::move_if_noexcept

Defined in header <utility>

```
template< class T >
typename std::conditional<
    !std::is_nothrow_move_constructible<T>::value && std::is_copy_constructible<T>::value,
    const T&,
    T&&
>::type move_if_noexcept( T& x ) noexcept;
```
(since C++11)
(until C++14)

```
template< class T >
constexpr typename std::conditional<
    !std::is_nothrow_move_constructible<T>::value && std::is_copy_constructible<T>::value,
    const T&,
    T&&
>::type move_if_noexcept( T& x ) noexcept;
```
(since C++14)

`std::move_if_noexcept` obtains an rvalue reference to its argument if its move constructor does not throw exceptions or if there is no copy constructor (move-only type), otherwise obtains an lvalue reference to its argument. It is typically used to combine move semantics with strong exception guarantee.

Parameters

x - the object to be moved or copied

Return value

`std::move(x)` or `x`, depending on exception guarantees.

Complexity

Move Semantics 💔 const

Move Semantics 💔 const

```
1 void foo() {
2   vector<int> v{};
3   v.push_back(33);
4   vector<int> copied = v;
5   vector<int> moved = std::move(v);
6   return;
7 }
```

Move Semantics 💔 const

```
1 void foo() {
2   const vector<int> v{33};
3   vector<int> copied = v;
4   vector<int> moved = std::move(v);
5   return;
6 }
```

Move Semantics 💔 const

```
1 void foo() {
2   const vector<int> v{33};
3   vector<int> copied = v;
4   vector<int> moved = std::move(v);
5   // Silently copies :(
6   return;
7 }
```

🌊 Ripple effects in the standard library 🌊

shared_future::get() copies 

```
1 void foo() {
2   shared_future<string> resultFuture
3     = threadPool.submit(doStuff);
4   return;
5 }
```

shared_future::get() copies 

```
1 void foo() {
2   shared_future<string> resultFuture
3     = threadPool.submit(doStuff);
4   string result
5     = std::move(resultFuture.get());
6   return;
7 }
```

shared_future::get() copies 

```
1 void foo() {
2   shared_future<string> resultFuture
3     = threadPool.submit(doStuff);
4   string result
5     = std::move(resultFuture.get());
6   // nice try 🤣 *deep copies your string*
7   return;
8 }
```

shared_future::get() copies 

std::shared_future<T>::get

Main template		
`const T& get() const;`	(1)	(since C++11)
std::shared_future<T&> specializations		
`T& get() const;`	(2)	(since C++11)
std::shared_future<void> specialization		
`void get() const;`	(3)	(since C++11)

The `get` member function waits (by calling `wait()`) until the shared state is ready, then retrieves the value stored in the shared state (if any).

If `valid()` is `false` before the call to this function, the behavior is undefined.

Return value

1) A const reference to the value stored in the shared state. The behavior of accessing the value through this reference after the shared state has been destroyed is undefined.
2) The reference stored as value in the shared state.
3) (none)

Exceptions

If an exception was stored in the shared state referenced by the future (e.g. via a call to `std::promise::set_exception()`) then that exception will be thrown.

Surely this is fine...?

```
1 void foo() {
2   vector<vector<int>> v{{1, 2, 3}, {4, 5, 6}};
3   return;
4 }
```

Surely this is fine...?

```
1 void foo() {
2   vector<vector<int>> v{
3     initializer_list<vector<int>>{
4       vector<int>{1, 2, 3}, vector<int>{4, 5, 6}
5     }
6   };
7   return;
8 }
```

What the heck is an init list?

std::initializer_list

(not to be confused with member initializer list)

Defined in header <initializer_list>

```
template< class T >
class initializer_list;
```
(since C++11)

An object of type `std::initializer_list<T>` is a lightweight proxy object that provides access to an array of objects of type `const T` (that may be allocated in read-only memory).

A `std::initializer_list` object is automatically constructed when:

- a brace-enclosed initializer list is used to list-initialize an object, where the corresponding constructor accepts an `std::initializer_list` parameter,
- a brace-enclosed initializer list is used as the right operand of assignment or as a function call argument, and the corresponding assignment operator/function accepts an `std::initializer_list` parameter,
- a brace-enclosed initializer list is bound to `auto`, including in a ranged for loop.

`std::initializer_list` may be implemented as a pair of pointers or pointer and length. Copying a `std::initializer_list` does not copy the backing array of the corresponding initializer list.

The program is ill-formed if an explicit or partial specialization of `std::initializer_list` is declared.

So this...

```
1 void foo() {
2   vector<vector<int>> v{{1, 2, 3}, {4, 5, 6}};
3   return;
4 }
```

... is equivalent to this

```
1 void foo() {
2   const vector<int> a{1, 2, 3};
3   const vector<int> b{4, 5, 6};
4   vector<vector<int>> v;
5   v.push_back(a);
6   v.push_back(b);
7   return;
8 }
```

vector(initializer_list) copies 

```
1  void foo() {
2    const vector<int> a{1, 2, 3};
3    const vector<int> b{4, 5, 6};
4    vector<vector<int>> v;
5    v.push_back(a);
6    // 🤣 *deep copies your vector*
7    v.push_back(b);
8    // 🤣 *deep copies your vector*
9    return;
10 }
```

Basic vector usage prevents move??

```
1 void foo() {
2   vector<vector<int>> v{{1, 2, 3}, {4, 5, 6}};
3   // 🤣 *deep copies your vectors*
4   return;
5 }
```

Why can’t we move from const?

Why can’t we move from const?

“performance issue, not a bug”

Why can’t we move from const?

“performance issue, not a bug”

“generic code can fall back to copy”

Why can’t we move from const?

“performance issue, not a bug”

“generic code can fall back to copy”

The real reason:
move semantics are second-class in C++

Are move semantics what we actually meant?

```
1 void foo() {
2   vector<int> v{1, 2, 3};
3   const auto other = std::move(v);
4   return;
5 }
```

Are move semantics what we actually meant?

```
1 void foo() {
2   vector<int> v{1, 2, 3};
3   // v.data = malloc(...);
4   const auto other = std::move(v);
5   return;
6 }
```

Are move semantics what we actually meant?

```
1 void foo() {
2   vector<int> v{1, 2, 3};
3   // v.data = malloc(...);
4   const auto other = std::move(v);
5   // other.data = v.data;
6   // v.data = nullptr;
7   return;
8 }
```

Are move semantics what we actually meant?

```
1  void foo() {
2    vector<int> v{1, 2, 3};
3    // v.data = malloc(...);
4    const auto other = std::move(v);
5    // other.data = v.data;
6    // v.data = nullptr;
7    // destroy other:
8    //   free(other.data);
9    return;
10 }
```

Are move semantics what we actually meant?

```
1  void foo() {
2    vector<int> v{1, 2, 3};
3    // v.data = malloc(...);
4    const auto other = std::move(v);
5    // other.data = v.data;
6    // v.data = nullptr;
7    // destroy other:
8    //   free(other.data);
9    // destroy v:
10   //   free(nullptr); // noop
11   return;
12 }
```

Are move semantics what we actually meant?

```c
// C
void foo() {
  vector_t v
    = {malloc(...), 3};
  v.data[0] = 1; ...
  const vector_t other = v;
  // other.data = v.data;
  free(other.data);
  return;
}
```

```cpp
// C++
void foo() {
  vector<int> v{1, 2, 3};
  // v.data = malloc(...);
  const auto other = std::move(v);
  // other.data = v.data;
  // v.data = nullptr;
  // destroy other:
  //   free(other.data);
  // destroy v:
  //   free(nullptr); // noop
  return;
}
```

Are move semantics what we actually meant?

```c
// C
void foo() {
  vector_t v
    = {malloc(...), 3};
  v.data[0] = 1; ...
  const vector_t other = v;
  // other.data = v.data;
  free(other.data);
  return;
}
```

```cpp
// C++
void foo() {
  vector<int> v{1, 2, 3};
  // v.data = malloc(...);
  const auto other = std::move(v);
  // other.data = v.data;
  // v.data = nullptr;
  // destroy other:
  //   free(other.data);
  // destroy v:
  //   free(nullptr); // noop
  return;
}
```

```rust
// Rust
fn foo() {
  let mut v = vec![1, 2, 3];
  // calls malloc(...);
  let other = v;
  // trivially copies v to other
  // calls core::mem::forget(v);
  // destroy other:
  //   free(...)
}
```

Wait, you can totally move from const?

```rust
// Rust
fn foo() {
  let mut v = vec![1, 2, 3];
  // calls malloc(...);
  let other = v;
  // trivially copies v to other
  // calls core::mem::forget(v);
  // destroy other:
  //   free(...)
}
```

Wait, you can totally move from const?

```rust
// Rust
fn foo() {
  let v = vec![1, 2, 3];
  // calls malloc(...);
  let other = v;
  // trivially copies v to other
  // calls core::mem::forget(v);
  // destroy other:
  //   free(...)
}
```

std::move

→ mutate the original object at runtime

→ a trivial copy **and** preventing object from destroying resources

"trivial-relocation"

→ mutate the original **type** at compile-time

→ a single trivial copy

What’s so great about trivial copies anyway?

Flashback: trivial copies

```
1  struct S {
2    int i;
3    double j;
4    char k;
5  };
6
7  void foo() {
8    struct S object = {2, 3.14, 'n'};
9    struct S other;
10   other.i = object.i;
11   other.j = object.j;
12   other.k = object.k;
13   return;
14 }
```

What do trivial copies compile to?

Source Editor: C source #1

New Privacy Policy. Please take a moment to read it ×

Last changed on: 4/22/2024, 3:25:31 PM (*diff*)

Compiler Explorer Privacy Policy

gcc/libgcc/memcpy.c

```c
/* Public domain.  */
#include <stddef.h>

void *
memcpy (void *dest, const void *src, size_t len)
{
  char *d = dest;
  const char *s = src;
  while (len--)
    *d++ = *s++;
  return dest;
}
```

memcpy-coalescing

Source Editor: C source #1

New Privacy Policy. Please take a moment to read it ×

Last changed on: 4/22/2024, 3:25:31 PM (*diff*)

Compiler Explorer Privacy Policy

memcpy-coalescing for arrays

```
1  struct S {
2    int i{};
3    double j{};
4    char k{};
5  };
6
7  void foo() {
8    vector<S> v;
9    v.resize(4);
10   v.resize(8);
11   return;
12 }
```

memcpy-coalescing for arrays

```
1  struct S {
2    int i{};
3    double j{};
4    char k{};
5  };
6
7  void foo() {
8    vector<S> v;
9    v.resize(4);
10   // v.data = malloc(4 * 24)
11   v.resize(8);
12   // v.new_data = malloc(8 * 24)
13   return;
14 }
```

memcpy-coalescing for arrays

```
1  struct S {
2    int i{};
3    double j{};
4    char k{};
5  };
6
7  void foo() {
8    vector<S> v;
9    v.resize(4);
10   // v.data = malloc(4 * 24)
11   v.resize(8);
12   // v.new_data = malloc(8 * 24)
13   // memcpy(v.new_data, v.data, 4 * sizeof(S))
14   // v.data = v.new_data
15   return;
16 }
```


We ❤️ bulk-memcpy

What about movable types?

```
1 void foo() {
2   vector<vector<int>> v;
3   v.resize(4);
4   v.resize(8);
5   // For each element:
6   // - std::move()
7   // - ~vector<int>(nullptr)
8   return;
9 }
```

Trivial relocation

```
1 void foo() {
2   vector<vector<int>> v;
3   v.resize(4);
4   v.resize(8);
5   // memcpy(v.new_data, v.data, 4 * sizeof(vector<int>))
6   // v.data = v.new_data
7   // prevent original destructors
8 }
```

std::is_trivially_relocatable proposal

One of Arthur O'Dwyer's CppCon talks

You actually didn’t mean to move anyway

What about function boundaries?

```
1 vector<int> bar();
2
3 void foo() {
4   vector<int> v = bar();
5 }
```

Does it copy?

```
1 vector<int> bar();
2
3 void foo() {
4   vector<int> v = bar();
5   // does it create vector<int>
6   // then pass it into
7   // vector<int>(const vector<int>&) ?
8 }
```

... or maybe it moves?

```
1 vector<int> bar();
2
3 void foo() {
4   vector<int> v = bar();
5   // does it create vector<int>
6   // then pass it into
7   // vector<int>(vector<int>&&) ?
8 }
```

“Guaranteed copy elision”

```
1 void barImpl(char* returnSlot);
2
3 void foo() {
4   // pseudocode:
5   char alloc[sizeof(vector<int>)];
6   barImpl(alloc);
7   auto v = (vector<int>*) &alloc;
8 }
```

What happens inside the function?

```
1 vector<int> bar() {
2   vector<int> temp{1, 2, 3};
3   temp.push_back(4);
4   return temp;
5 }
6
7 void barImpl(char* returnSlot);
```

Return Value Optimization

```
1  vector<int> bar() {
2    vector<int> temp{1, 2, 3};
3    temp.push_back(4);
4    return temp;
5  }
6
7  void barImpl(char* returnSlot) {
8    new (&returnSlot) vector<int>(1, 2, 3);
9    ((vector<int>*)returnSlot)→push_back(4);
10   return;
11 }
```

Not a copy in sight!

```
1 vector<int> bar() {
2   vector<int> temp{1, 2, 3};
3   temp.push_back(4);
4   return temp;
5 }
6
7 void foo() {
8   vector<int> v = bar();
9 }
```

RVO is extremely easy to break

Move Semantics 💔 RVO

```
1 vector<int> bar() {
2   vector<int> temp{1, 2, 3};
3   return std::move(temp);
4 }
```

Move Semantics 💔 RVO

```
vector<int> bar() {
  vector<int> temp{1, 2, 3};
  return std::move(temp);
  // Return type is vector<int>,
  // so we have to convert!
}
```

“Can we have multiple returns?”

```
vector<int>, vector<int> bar() {
  vector<int> temp1{1, 2, 3};
  vector<int> temp2{1, 2, 3};
  return temp1, temp2;
}
```

“We have multiple returns at home:”

```
1 pair<vector<int>, vector<int>> bar() {
2   vector<int> temp1{1, 2, 3};
3   vector<int> temp2{1, 2, 3};
4   return {temp1, temp2};
5 }
```

No RVO? Maybe auto-move

```
1 vector<int> bar(bool cond) {
2   vector<int> temp1{1, 2, 3};
3   vector<int> temp2{1, 2, 3};
4   if (cond)
5     return temp1;
6   return temp2;
7 }
```

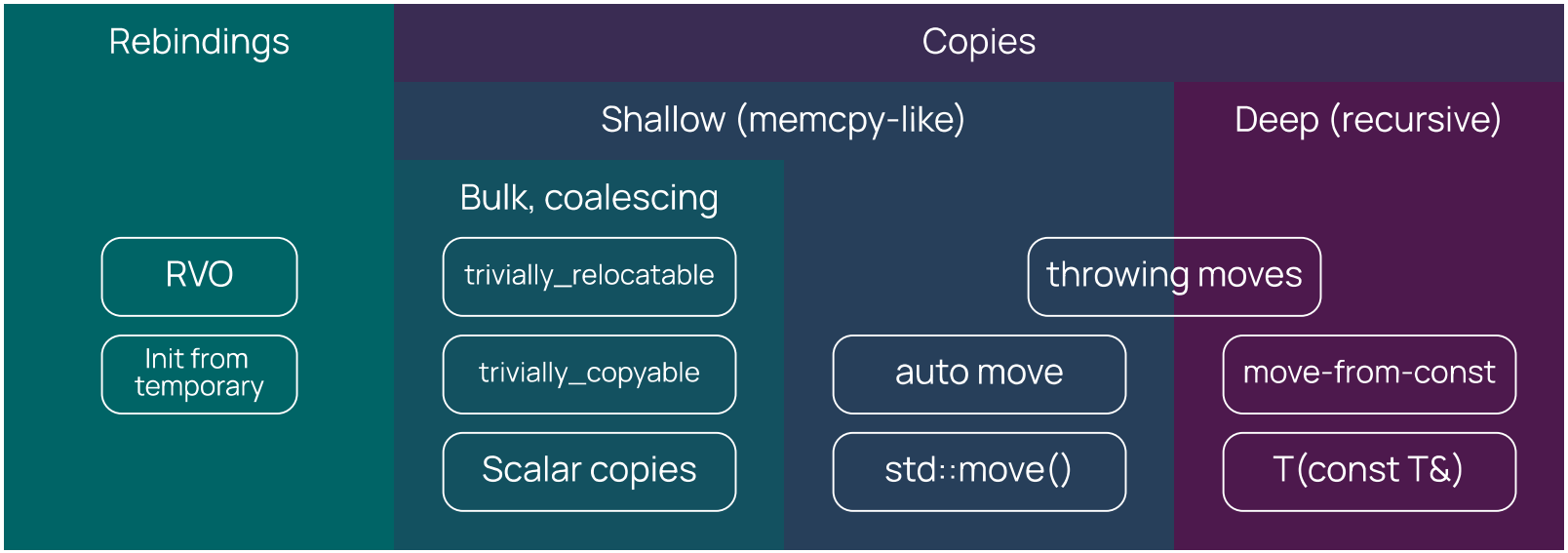
No RVO? Maybe auto-move

```
1  void barImpl(bool cond, char* returnSlot) {
2    vector<int> temp1{1, 2, 3};
3    vector<int> temp2{1, 2, 3};
4    if (cond) {
5      new (&returnSlot) vector<int>(temp1);
6      return;
7    }
8    new (&returnSlot) vector<int>(temp2);
9    return;
10 }
```

No RVO? Maybe auto-move

```
void barImpl(bool cond, char* returnSlot) {
  vector<int> temp1{1, 2, 3};
  vector<int> temp2{1, 2, 3};
  if (cond) {
    new (&returnSlot) vector<int>(std::move(temp1));
    return;
  }
  new (&returnSlot) vector<int>(std::move(temp2));
  return;
}
```

Family tree of copies



Part III: Doing something about it

How do we track down copies?

```
1 class PleaseDontCopy {
2   // ...
3 };
```

Delete the copy constructor

```
1 class PleaseDontCopy {
2   PleaseDontCopy(const PleaseDontCopy&) = delete;
3   // ...
4 };
5
6 void foo() {
7   PleaseDontCopy a{};
8   auto b = a; // error
9 }
```

Deprecate the copy constructor

```cpp
class PleaseDontCopy {
  [[deprecated]]
  PleaseDontCopy(const PleaseDontCopy&) = default;
  // ...
};

void foo() {
  PleaseDontCopy a{};
  auto b = a; // warning (-Wdeprecated-declarations)
}
```

Make copies explicit

```cpp
class PleaseDontCopy {
  explicit
  PleaseDontCopy(const PleaseDontCopy&) = default;
  // ...
};

void foo() {
  PleaseDontCopy a{};
  auto b = PleaseDontCopy{a};
}
```

C++ Weekly - Ep 241 - Using `explicit` to Find Expensive Accidental Copies

Announce copies

```
1  class TraceMe {
2    TraceMe(const TraceMe&) {
3      std::println("Copied!");
4      PERFETTO_TRACE("Copied!");
5    }
6  };
7
8  class PleaseDontCopy : TraceMe {
9    // ...
10 };
11
12 void foo() {
13   PleaseDontCopy a{};
14   auto b = a; // prints at runtime
15 }
```

<< (shift left, get it?)

Compiler warnings

Compiler warnings

- -Wmove

Compiler warnings

- -Wmove
- -Wlarge-by-value-copy

clang-tidy is awesome

Clang-Tidy Checks

Name	Offers fixes
abseil-cleanup-ctad	Yes
abseil-duration-addition	Yes
abseil-duration-comparison	Yes
abseil-duration-conversion-cast	Yes
abseil-duration-division	Yes
abseil-duration-factory-float	Yes
abseil-duration-factory-scale	Yes
abseil-duration-subtraction	Yes
abseil-duration-unnecessary-conversion	Yes
abseil-faster-strsplit-delimiter	Yes
abseil-no-internal-dependencies	
abseil-no-namespace	
abseil-redundant-strcat-calls	Yes
abseil-str-cat-append	Yes
abseil-string-find-startswith	Yes
abseil-string-find-str-contains	Yes
abseil-time-comparison	Yes

Missing check? No problem 😎

Getting Involved

clang-tidy has several own checks and can run Clang static analyzer checks, but its power is in the ability to easily write custom checks.

Checks are organized in modules, which can be linked into **clang-tidy** with minimal or no code changes in **clang-tidy**.

Checks can plug into the analysis on the preprocessor level using **PPCallbacks** or on the AST level using **AST Matchers**. When an error is found, checks can report them in a way similar to how Clang diagnostics work. A fix-it hint can be attached to a diagnostic message.

The interface provided by **clang-tidy** makes it easy to write useful and precise checks in just a few lines of code. If you have an idea for a good check, the rest of this document explains how to do this.

There are a few tools particularly useful when developing clang-tidy checks:

`add_new_check.py` is a script to automate the process of adding a new check, it will create the check, update the CMake file and create a test;
`rename_check.py` does what the script name suggests, renames an existing check;
pp-trace logs method calls on *PPCallbacks* for a source file and is invaluable in understanding the preprocessor mechanism;
clang-query is invaluable for interactive prototyping of AST matchers and exploration of the Clang AST;
clang-check with the `-ast-dump` (and optionally `-ast-dump-filter`) provides a convenient way to dump AST of a C++ program.

If CMake is configured with `CLANG_TIDY_ENABLE_STATIC_ANALYZER=NO`, **clang-tidy** will not be built with support for the `clang-analyzer-*` checks or the `mpi-*` checks.

Choosing the Right Place for your Check

Missing check? No problem 😎

```
$ git clone https://github.com/llvm/llvm-project
$ cd llvm/clang-tools-extra
```

Missing check? No problem 😎

```
$ git clone https://github.com/llvm/llvm-project
$ cd llvm/clang-tools-extra
$ clang-tidy/add_new_check.py performance vector-init-list
# Creating clang-tidy/performance/VectorInitListCheck.cpp...
```

Test in clang-query

```
1  struct Test {
2    std::string Data;
3    int Data2;
4  };
5
6  std::vector<Test> TestFn(int i) {
7    std::vector<std::string> V({"", "", {}});
8    std::vector<std::string> V1{"", ""};
9
10   return {{"hi", 1}, {}, {"there", 3}};
11 }
```

Test in clang-query

```
match cxxConstructExpr()
```

```
1  struct Test {
2    std::string Data;
3    int Data2;
4  };
5
6  std::vector<Test> TestFn(int i) {
7    std::vector<std::string> V({"", "", {}});
8    //                         ^  ^   ^   ^
9    std::vector<std::string> V1{"", ""};
10   //                         ^  ^   ^
11
12   return {{"hi", 1}, {}, {"there", 3}};
13   //     ^^^         ^   ^^
14 }
```

Test in clang-query

```
match cxxConstructExpr(
  has(cxxStdInitializerListExpr())
)
```

```
1  struct Test {
2    std::string Data;
3    int Data2;
4  };
5
6  std::vector<Test> TestFn(int i) {
7    std::vector<std::string> V({"", "", {}});
8    //                         ^~~~~~~~~~~~~~~
9    std::vector<std::string> V1{"", ""};
10   //                         ^~~~~~~~~~
11
12   return {{"hi", 1}, {}, {"there", 3}};
13   //      ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
14 }
```

Test in clang-query

```
match cxxConstructExpr(
  has(cxxStdInitializerListExpr()),
  hasDeclaration(
    cxxConstructorDecl(
      ofClass(
        cxxRecordDecl(
          classTemplateSpecializationDecl(
            hasName("::std::vector")
          )
        )
      )
    )
  )
)
```

```
1  struct Test {
2    std::string Data;
3    int Data2;
4  };
5
6  std::vector<Test> TestFn(int i) {
7    std::vector<std::string> V({"", "", {}});
8    //                        ^~~~~~~~~~~~~~~
9    std::vector<std::string> V1{"", ""};
10   //                        ^~~~~~~~~~
11
12   return {{"hi", 1}, {}, {"there", 3}};
13   //     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
14 }
```

Compile your own clang-tidy

```
$ git clone https://github.com/llvm/llvm-project
$ cd llvm/clang-tools-extra
$ clang-tidy/add_new_check.py performance vector-init-list
# Creating clang-tidy/performance/VectorInitListCheck.cpp...
$ cmake ... -DLLVM_ENABLE_PROJECTS="clang-tools-extra"
$ ninja install
```

Takeaways

Takeaways

- Design the system to avoid dynamic ownership of data (Do like llama.cpp does! `mmap` your data into the program and leave it there!)

Takeaways

- Design the system to avoid dynamic ownership of data (Do like llama.cpp does! `mmap` your data into the program and leave it there!)
- Use simple types

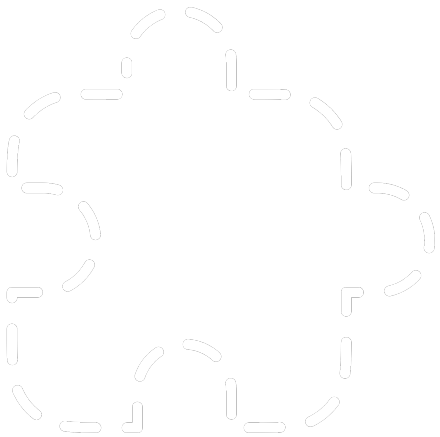
Takeaways

- Design the system to avoid dynamic ownership of data (Do like llama.cpp does! `mmap` your data into the program and leave it there!)
- Use simple types
- Trace your programs

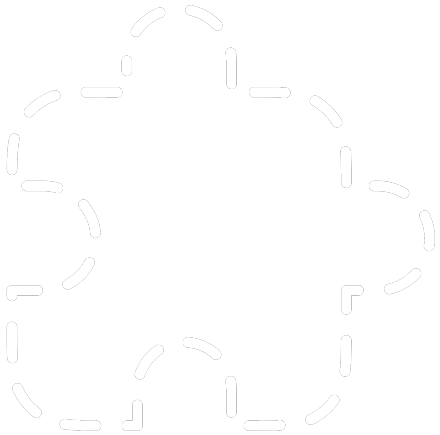
Takeaways

- Design the system to avoid dynamic ownership of data (Do like llama.cpp does! `mmap` your data into the program and leave it there!)
- Use simple types
- Trace your programs
- Use compiler warnings and linters

What are we missing?

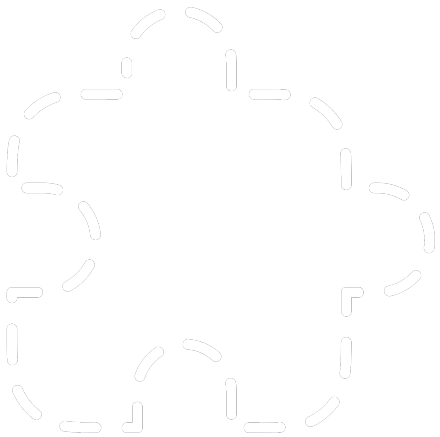


What are we missing?



True move semantics in C++? Borrow checker?

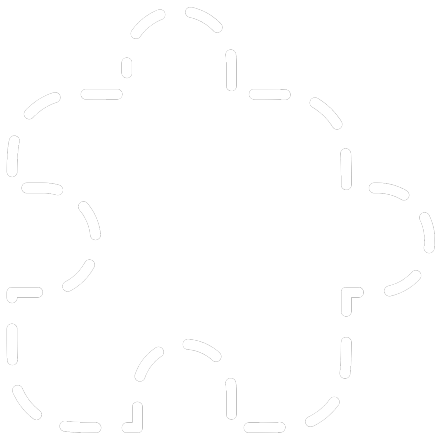
What are we missing?



True move semantics in C++? Borrow checker?

Multiple returns in C++?

What are we missing?



True move semantics in C++? Borrow checker?

Multiple returns in C++?

grep + clang-tidy + checks-from-json?