Invisible Problems

On the job

Let’s imagine being a C++ programmer

We are ok with the language, but not super secure

We are tasked with upgrading a part of an older system

And in the section we are now, the best solution is a struct with two integers

Lets do this

```
struct VeryImportant
{
    int myOne;
    int myTwo;
};
```


“Please make sure everything is correctly initialized, we can’t have junk values here!”

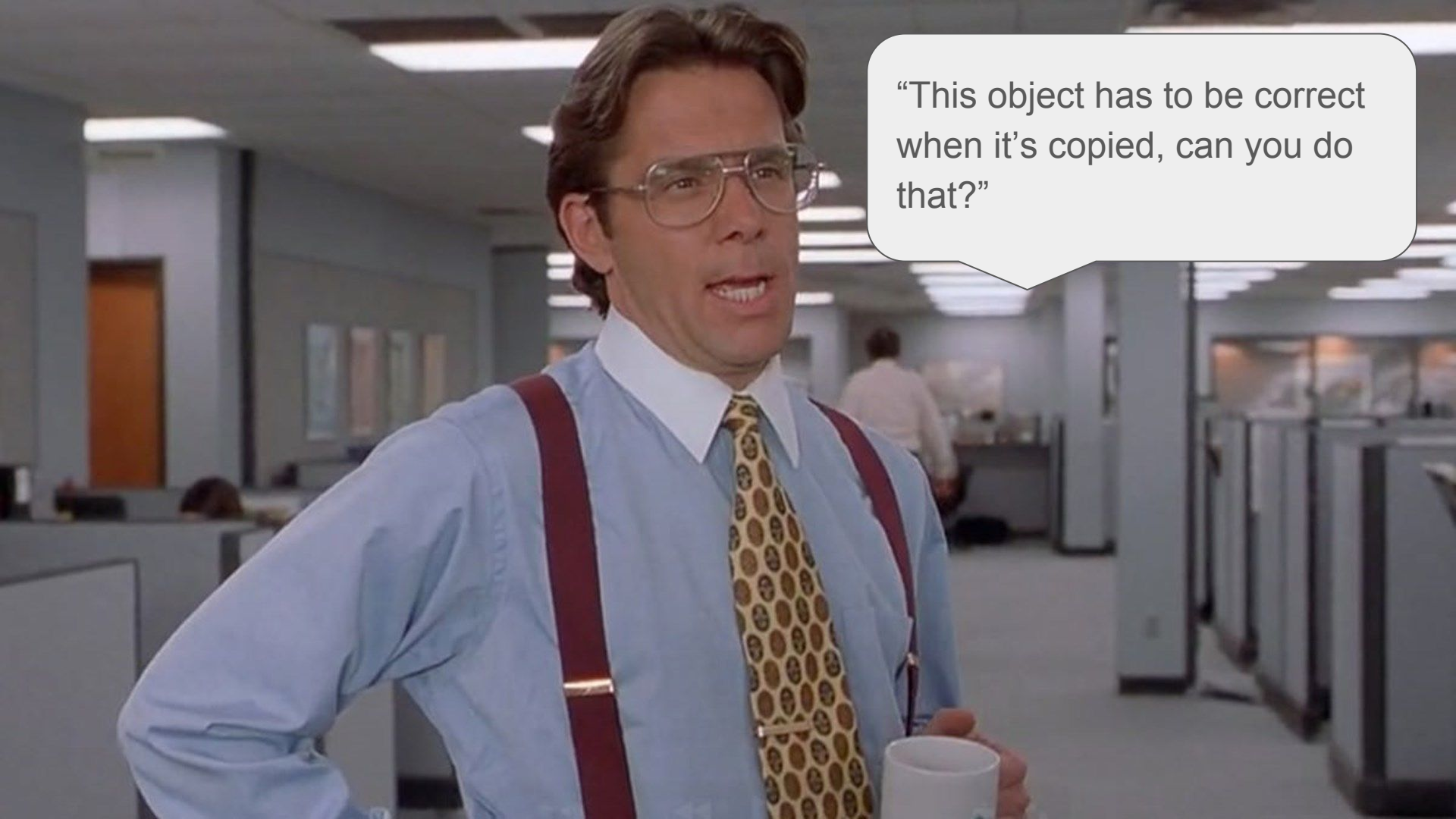
Zero is default? Right?

```
struct VeryImportant
{
    int myOne;
    int myTwo;
    VeryImportant(): myOne(0), myTwo(0) {}
};
```


“I want to be able to set the values when I create the object!”

We got this!

```
struct VeryImportant
{
    int myOne;
    int myTwo;
    VeryImportant(): myOne(0), myTwo(0) {}
    VeryImportant(int aOne, int aTwo)
        : myOne(aOne), myTwo(aTwo) {}
};
```


“This object has to be correct when it’s copied, can you do that?”

The rule of how many?

```
struct VeryImportant
{
     int myOne;
     int myTwo;
     VeryImportant(): myOne(0), myTwo(0) {}
     VeryImportant(int aOne, int aTwo): myOne(aOne), myTwo(aTwo) {}
     VeryImportant(const VeryImportant& aVery)
          : myOne(aVery.myOne), myTwo(aVery.myTwo) {}
     VeryImportant& operator=(const VeryImportant& aVery)
     {
          myOne = aVery.myOne;
          myTwo = aVery.myTwo;
          return *this;
     }
};
```


“We upgraded to the new compiler, it supports move semantics, can you make sure everything is movable?”

```
struct VeryImportant
{
     int myOne;
     int myTwo;
     VeryImportant(): myOne(0), myTwo(0) {}
     VeryImportant(int aOne, int aTwo): myOne(aOne), myTwo(aTwo) {}
     VeryImportant(const VeryImportant& aVery)
          : myOne(aVery.myOne), myTwo(aVery.myTwo) {}
     VeryImportant& operator=(const VeryImportant& aVery)
     {
          myOne = aVery.myOne;
          myTwo = aVery.myTwo;
          return *this;
     }
     VeryImportant(VeryImportant&& aVery)
          : myOne(aVery.myOne), myTwo(aVery.myTwo) {}
     VeryImportant& operator=(VeryImportant&& aVery)
     {
          myOne = aVery.myOne;
          myTwo = aVery.myTwo;
          return *this;
     }
};
```


“Why is our new VeryImportant struct slower than our old library?”

Searching starts

You look over your code, nothing

Searching starts

You look over your code, nothing

You look at all the call sites, nothing

Searching starts

You look over your code, nothing

You look at all the call sites, nothing

You start doing git blame diffs, nothing

Searching starts

You look over your code, nothing

You look at all the call sites, nothing

You start doing git blame diffs, nothing

You go into assembly...

```
404dc4          callq  403720 <memcpy@plt>
```

“Why is my code slow?”

You start googling something like that

Sprinkle in memcpy and structs

A certain word keeps popping up

“Why is my code slow?”

You start googling something like that

Sprinkle in memcpy and structs

A certain word keeps popping up

Trivial

“Objects of trivially-copyable types are the only C++ objects that may be safely copied with std::memcpy”

C++ named requirements: *TriviallyCopyable*

Requirements

- Every copy constructor is trivial or deleted
- Every move constructor is trivial or deleted
- Every copy assignment operator is trivial or deleted
- Every move assignment operator is trivial or deleted
- at least one copy constructor, move constructor, copy assignment operator, or move assignment operator is non-deleted
- Trivial non-deleted destructor

This implies that the class has no virtual functions or virtual base classes.

```
struct VeryImportant
{
     int myOne;
     int myTwo;
     VeryImportant(): myOne(0), myTwo(0) {}
     VeryImportant(int aOne, int aTwo): myOne(aOne), myTwo(aTwo) {}
     VeryImportant(const VeryImportant& aVery)
          : myOne(aVery.myOne), myTwo(aVery.myTwo) {}
     VeryImportant& operator=(const VeryImportant& aVery)
     {
          myOne = aVery.myOne;
          myTwo = aVery.myTwo;
          return *this;
     }
     VeryImportant(VeryImportant&& aVery)
          : myOne(aVery.myOne), myTwo(aVery.myTwo) {}
     VeryImportant& operator=(VeryImportant&& aVery)
     {
          myOne = aVery.myOne;
          myTwo = aVery.myTwo;
          return *this;
     }
};
```

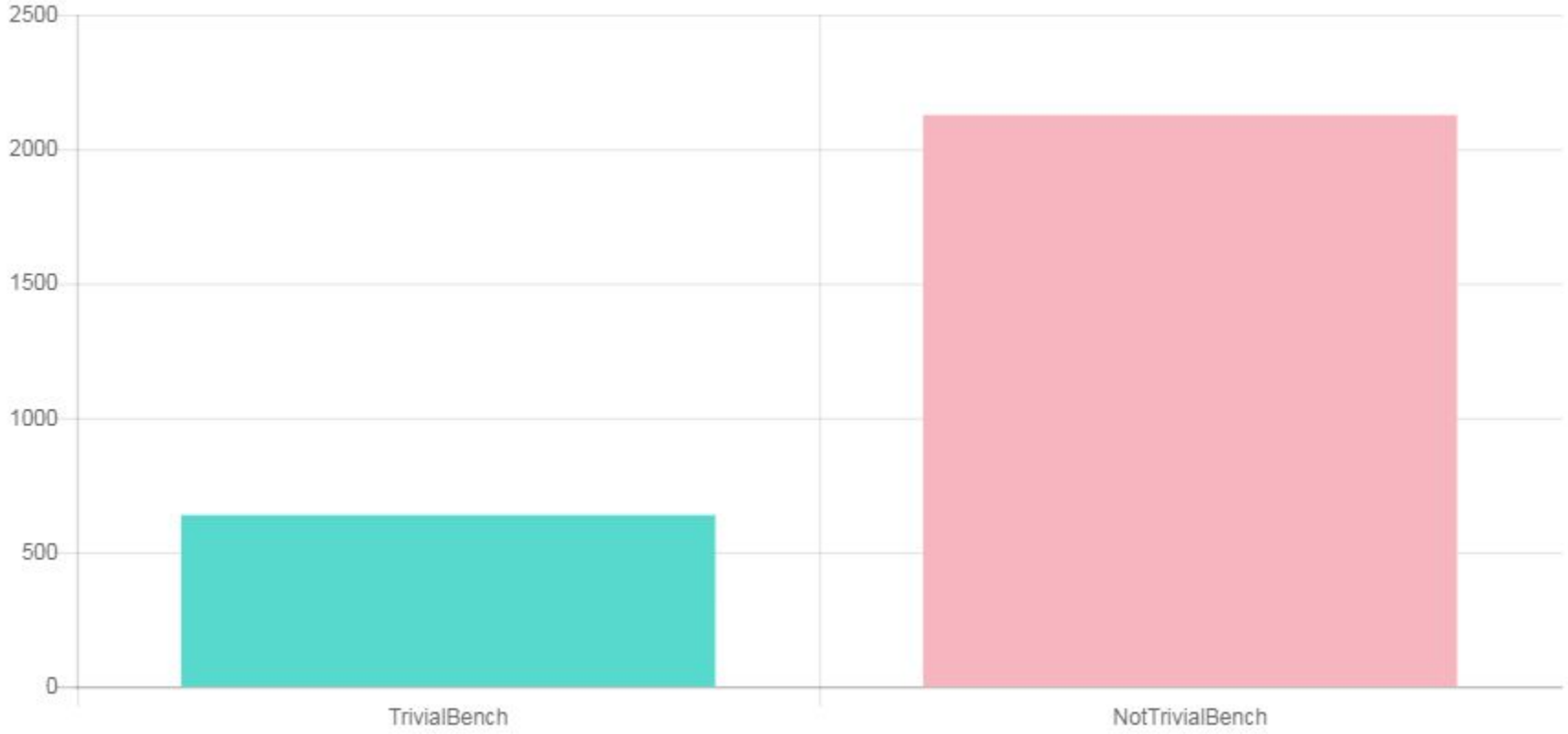
Trivial default constructor

The default constructor for class T is trivial (i.e. performs no action) if all of the following is true:

- The constructor is not user-provided (i.e., is implicitly-defined or defaulted on its first declaration)
- T has no virtual member functions
- T has no virtual base classes
- T has no non-static members with default initializers. (since C++11)
- Every direct base of T has a trivial default constructor
- Every non-static member of class type has a trivial default constructor

We’re back here

```
struct VeryImportant
{
    int myOne;
    int myTwo;
    VeryImportant(): myOne(0), myTwo(0) {}
    VeryImportant(int aOne, int aTwo): myOne(aOne), myTwo(aTwo) {}
};
```


2500
2000
1500
1000
500
0
TrivialBench
NotTrivialBench

The only healthy choice...

```
#define ENFORCE_TRIVIAL(t) \
static_assert(std::is_standard_layout_v<t>); \
static_assert(std::is_trivially_copyable_v<t>); \
static_assert(std::is_trivially_copy_assignable_v<t>); \
static_assert(std::is_trivially_copy_constructible_v<t>); \
static_assert(std::is_trivially_move_assignable_v<t>); \
static_assert(std::is_trivially_move_constructible_v<t>); \
static_assert(std::is_trivially_destructible_v<t>);

#define TRIVIAL_STRUCT(name, ...) \
struct name __VA_ARGS__; \
ENFORCE_TRIVIAL(name);
```

```
TRIVIAL_STRUCT(VeryImportant,
{
    int myOne;
    int myTwo;
    VeryImportant(): myOne(0), myTwo(0) {}
    VeryImportant(int aOne, int aTwo): myOne(aOne),
myTwo(aTwo) {}
});
```

My wish

```
struct [[bikeshed_trivially_copyable]] VeryImportant
{
    int myOne;
    int myTwo;
    VeryImportant(): myOne(0), myTwo(0) {}
    VeryImportant(int aOne, int aTwo)
        : myOne(aOne), myTwo(aTwo) {}
};
```

My wish

```
struct [[bikeshed_trivially_copyable]] VeryImportant
{
    int myOne;
    int myTwo;
    VeryImportant(): myOne(0), myTwo(0) {}
    VeryImportant(int aOne, int aTwo): myOne(aOne),
myTwo(aTwo) {}
    VeryImportant(const VeryImportant& aVery)
        : myOne(aVery.myOne), myTwo(aVery.myTwo) {}
    // Compile Error
};
```

Twitter:

@olafurw