A Semi Compile/Run-time Map with (Nearly) Zero Overhead Lookup

Fabian Renn-Giles

Watch video of the talk here:

https://www.youtube.com/watch?v=qNAbGpV1ZkU

fabian.renn@gmail.com
@hogliux

1. Introduction

Introduction

Usage example: C++ ↔ Java Bridge

```
class InputStream // java.io.InputStream
{
public:
  inline void close()           { jclass.getMethod("close").invoke(); }
  inline long skip(long bytes) { return jclass.getMethod("skip").invoke(bytes); }
  inline int read()            { return jclass.getMethod("read").invoke(); }
  .
  .
  .
};
```

called in tight-loop

Ensure getMethod is as fast as possible

Auto-generated C++ code for every Java class

Introduction

Use a cache!

```
JavaMethod getMethod(const std::string& methodName)
{
  static std::unordered_map<std::string, JavaMethod> cache;
  return cache.try_emplace(methodName, methodName).first->second;
}
```

- `try_emplace` needs to calculate the hash of `methodName`
- Collisions expected as thousands of methods in cache

➔ **Can we do better?**

Introduction

Use a cache!

```
class InputStream // java.io.InputStream
{
public:
  void close()              { jclass.getMethod("close").invoke(); }
  long skip(long bytes) { return jclass.getMethod("skip").invoke(bytes); }
  int read()               { return jclass.getMethod("read").invoke(); }
 };
```

String literals

- `methodName` is known at compile-time
- it should be possible to calculate the cache slot at compile-time
- Fallback to run-time hash if `methodName` is not a string literal

➔ **Somebody must have done this already, right?**

Introduction

Ben Deane's and Jason Turner's "`constepxr` all the things"[1,2]

```
cx::map<Key, Value>
```

- map is either completely compile-time or completely run-time
- If lookup is compile-time then all values must also be known at compile-time
- `JavaMethod` can only be resolved at run-time

✘ Compile-time lookup of run-time value not possible

[1] https://www.youtube.com/watch?v=PJwd4JLYJJY

[2] there are many more compile-time only maps (for example boost::hana::map)

Introduction

Louis Dionne's C++ meta-programming talk at meeting C++[1]

```
event_system events{{"foo", "bar", "baz"}}
```

- ✔ Lookup at compile-time and value-storage at runtime
- ✔ Can fallback to run-time lookup if key is not string literal
- ✘ Need to know all possible keys in advance

[1]https://goo.gl/M4vhqc from slide 73 onwards

Introduction

Let's roll our own!

```
semi::map<Key, Value>
```

Requirements

- Lookup at compile-time and value-storage at runtime
- Can fallback to run-time lookup if key is not string literal
- Keys need not to be known ahead of time

2. The basic principle

The basic principle

The basic principle is embarrassingly simple!

```
template <int> std::string map;
```

✔ map with `int` keys and `std::string` values

✔ Lookup at compile-time and value-storage at runtime

✔ Keys need not be known ahead of time

➔ **Compile-time lookup, run-time storage map in a single line of code**

The basic principle

What keys can we use?

```
template <Key> Value map;
```

- Any non-type template parameter types
- essentially `bool`, integral types and `nullptr_t`
- **C++20:** Class types as non-type template parameters!
    - Any compile-time literals, user-defined literals, ...

➔ **It's 2018: no compiler supports class types as template parameters**

3. Supporting more key types in 2018

Supporting more key types in 2018

Use `typenames` instead

```
template <typename> Value map;
```

- We need a “function” which maps any key to a unique typename:

```
UniqueReturnType key2type(Key key);
```

```
std::cout << map<decltype(key2type("foo"))>;
```

➔ **Is it possible to write `key2type` in C++?**

Supporting more key types in 2018

Implementing `key2type` for integral key types

```
template <auto...> struct dummy_t {};

template <typename Key>
constexpr auto key2type(Key key)
{
  return dummy_t<key>{};
}
```

```
std::cout << map<decltype(key2type(5))>;
```

- Returns instance of dummy_t<5> for key “5”, and so on…

✘ Compiler-error: “key” not a constant expression

Supporting more key types in 2018

Implementing `key2type` for integral key types

```
template <auto...> struct dummy_t {};

template <typename Lambda>[1]
constexpr auto key2type(Lambda lambda)
{
  return dummy_t<lambda()>{};
}
```

```
std::cout << map<decltype(key2type([] () { return 5; }))>;[2]
```

[1] `enable_if` template parameter removed for brevity
[2] only shown to demonstrate usage: error due to lambda being used in an unevaluated context - will be resolved on slide 22

Supporting more key types in 2018

Implementing `key2type` for integral key types

```
template <auto...> struct dummy_t {};

template <typename Lambda>1
constexpr auto key2type(Lambda lambda)
{
  return dummy_t<lambda()>{};
}
#define ID(x)  []() constexpr { return x; }
```

```
std::cout << map<decltype(key2type(ID(5)))> << std::endl;2
```

- Still doesn't work with string literals etc.

[1] `enable_if` template parameter removed for brevity
[2] only shown to demonstrate usage: error due to lambda being used in an unevaluated context - will be resolved on slide 22

Supporting more key types in 2018

Supporting string literals:

```
template <typename Lambda, std::size_t... I>
constexpr auto str2type(Lambda lambda, std::index_sequence<I...>)
{
  return dummy_t<lambda()[I]...>{};
}

template <typename Lambda>¹
constexpr auto key2type(Lambda lambda)
{
  return array2type (lambda, std::make_index_sequence<strlen³(lambda())>{});
}
```

```
std::cout << map<decltype(key2type(ID("foo")))> << std::endl;²
```

- Maps "foo" onto dummy_t<'f','o','o'>

[1] `enable_if` template parameter removed for brevity
[2] only shown to demonstrate usage: error due to lambda being used in an unevaluated context - will be resolved on slide 22
[3] `strlen` is marked `constexpr` in gcc only: substitute implementation for other compilers is left as an exercise for the reader

4. Putting it all together

Putting it all together

```
template <typename> std::string map_value;

map_value<decltype(key2type(ID("conference")))> = "cppcon";
```

Putting it all together

```
template <typename Value, typename>
Value& static_map_get()
{
    static Value value;
    return value;
}

static_map_get<std::string, decltype(key2type(ID("conference")))>() = "cppcon";
```

Putting it all together

```
template <typename Key, typename Value>
class static_map
{








public:
  template <typename>
  static Value& get()
  {
    static Value value;
    return value;
  }
};

using map = static_map<std::string, std::string>;
map::get<decltype(key2type(ID("conference")))>() = "cppcon";
```

Putting it all together

```
template <typename Key, typename Value>
class static_map
{
public:
  template <typename Lambda>
  static Value& get(Lambda lambda)
  {

    return get_internal<decltype(key2type(lambda))>();
  }
private:
  template <typename>
  static Value& get_internal()
  {
    static Value value;
    return value;
  }
};

using map = static_map<std::string, std::string>;
map::get(ID("conference")) = "cppcon";1
```


[1]constexpr lambda is now being evaluated as a normal parameter which resolves the issue mentioned in the footnote of slides 15,16,17

Putting it all together

```
template <typename Key, typename Value>
class static_map
{
public:
  template <typename Lambda>
  static Value& get(Lambda lambda)
  {
    static_assert(std::is_convertible_v<decltype(lambda()), Key>);
    return get_internal<decltype(key2type(lambda))>();
  }
private:
  template <typename>
  static Value& get_internal()
  {
    static Value value;
    return value;
  }
};

using map = static_map<std::string, std::string>;
map::get(ID("conference")) = "cppcon";1
```


[1]constexpr lambda is now being evaluated as a normal parameter which resolves the issue mentioned in the footnote of slides 15,16,17

Putting it all together

What's the performance?

```cpp
int& getAge()
{
  using map =
    semi::static_map<std::string, int>;

  return map::get(ID("age"));
}
```

```asm
mov eax, OFFSET FLAT:dummy_tlvalue
ret
```

Compiler Explorer: https://gcc.godbolt.org/z/6CRZ9W

➔ **Just as fast as accessing a global variable**

5. Optional run-time lookup

Optional run-time lookup

Impossible to write run-time version of key2type...

```
UniqueReturnType key2type(Key key);
```

...also impossible to somehow call `get` with run-time key

```
template <typename Lambda> get(Lambda lambda);
```

We can write a big switch-like statement:

```
if      (key == “food”)  return get(ID(“food”));
else if (key == “drink”) return get(ID(“drink”));
...
```

...but then all keys need to be known in advance.

Optional run-time lookup

Solution: maintain run-time map of pointers to static locals

```
std::unordered_map<Key, Value*> runtime_map;
```

Add to map when value is accessed via compile-time literal:

```
template <typename>
static Value& get_internal (const Key& key)
{
  static Value value;

  runtime_map[key] = &value;
  return value;
}
```

Slow: Executed every time `get` is called

➔ **Only add to map when get_internal is executed for the first time?**

Optional run-time lookup

Closer look at the static local variable:

```
template <typename>
static Value& get_internal()
{
  static Value value;


  return value;
}
```

Optional run-time lookup

Closer look at the static local variable:

```
template <typename>
static Value& get_internal()
{
  static Value value;

  return value;
}
```

1. Creates storage
2. Calls constructor

Optional run-time lookup

Closer look at the static local variable:

```
struct ConstructorInvoker
{
    ConstructorInvoker(char* mem) { new (mem) Value; }
};

template <typename>
static Value& get_internal()
{
  alignas (Value) static char storage[sizeof(Value)];
  static ConstructorInvoker invoker(storage);

  return *reinterpret_cast<Value*> (storage);
}
```

➔ **This change has no effect on the generated assembly**

Optional run-time lookup

Undefined behaviour?!?

```
struct ConstructorInvoker
{
    ConstructorInvoker(char* mem) { new
};

template <typename>
static Value& get_internal()
{
  alignas (Value) static char storage[sizeof(Value)];
  static ConstructorInvoker invoker(storage);

  return *reinterpret_cast<Value*> (storage);
}
```

- No alignment issues
- No aliasing

➔ **No excuses: standard says `reinterpret_cast` to `Value` is UB!**

Optional run-time lookup

Undefined behaviour?!?

```
struct ConstructorInvoker
{
    ConstructorInvoker(char* mem) { new (mem) Value; }
};

template <typename>
static Value& get_internal()
{
  alignas (Value) static char storage[sizeof(Value)];
  static ConstructorInvoker invoker(storage);

  return *std::launder(reinterpret_cast<Value*> (storage));
}
```

Meet the most obscure C++ function: `std::launder`

- Can be used to avoid UB in these types of cases
- cppreference.com shows exactly this example

Optional run-time lookup

Controlling the construction with an init flag

```
struct ConstructorInvoker
{
    ConstructorInvoker(char* mem) { new (mem) Value; }
};

template <typename>
static Value& get_internal(const Key& key)
{
  alignas (Value) static char storage[sizeof(Value)];
  static ConstructorInvoker invoker(storage);

  return *reinterpret_cast<Value*> (storage);
}
```

Optional run-time lookup

Controlling the construction with a needs_init flag

```
void initialise(const Key& key, char* mem, bool& needs_init);

template <typename>
static Value& get_internal(const Key& key)
{
  alignas (Value) static char storage[sizeof(Value)];
  static bool needs_init = true;

  if (needs_init) {
    initialise(key, storage, needs_init); needs_init = false;
  }

  return *reinterpret_cast<Value*> (storage);
}
```

No more thread-safety

- Generated code is faster: no more locks
- Just like STL containers, we do not claim that semi::map is thread-safe

Optional run-time lookup

Controlling the construction with a needs_init flag

```
void initialise(const Key& key, char* mem, bool& needs_init);

template <typename>
static Value& get_internal(const Key& key)
{
  alignas (Value) static char storage[sizeof(Value)];
  static bool needs_init = true;

  if (__builtin_expect(needs_init, false)) {
    initialise(key, storage, needs_init); needs_init = false;
  }

  return *reinterpret_cast<Value*> (storage);
}
```

No more thread-safety

- Generated code is faster: no more locks
- Just like STL containers, we do not claim that semi::map is thread-safe

Optional run-time lookup

Closer look at the `initialise` function

```
void initialise(const Key& key, char* mem, bool& needs_init)
{
                          new (mem) Value                                ;
}
```

Optional run-time lookup

Closer look at the `initialise` function

```
void initialise(const Key& key, char* mem, bool& needs_init)
{
  runtime_map.try_emplace(key, new (mem) Value                           );
}






static std::unorederd_map<Key, std::unique_ptr<Value              >> runtime_map;
```

- Add `Value` to `runtime_map`

Optional run-time lookup

Closer look at the `initialise` function

```
void initialise(const Key& key, char* mem, bool& needs_init)
{
  runtime_map.try_emplace(new (mem) Value, ValueDeleter{needs_init});
}

struct ValueDeleter
{
  void operator()(Value* v) { v->~Value();                    }
  bool& needs_init;
};

static std::unorederd_map<Key, std::unique_ptr<Value, ValueDeleter>> runtime_map;
```

- Add `Value` to `runtime_map`
- Ensure that destructor is called but that storage is **not** deallocated

Optional run-time lookup

Closer look at the `initialise` function

```
void initialise(const Key& key, char* mem, bool& needs_init)
{
  runtime_map.try_emplace(new (mem) Value, ValueDeleter{needs_init});
}

struct ValueDeleter
{
  void operator()(Value* v) { v->~Value(); needs_init = true; }
  bool& needs_init;
};

static std::unorederd_map<Key, std::unique_ptr<Value, ValueDeleter>> runtime_map;
```

- Add `Value` to `runtime_map`
- Ensure that destructor is called but that storage is **not** deallocated

Optional run-time lookup

Getting a value from a runtime key is now trivial!

```
static Value& get(const Key& key)
{
  return *runtime_map[key];
}

using map = semi::static_map<std::string, std::string>

std::string key;
std::cin >> key;
map::get(key);
```

- We also need to account for the case where a value is accessed via run-time key first and **then** via a compile-time key literal
- Not very interesting: see full-code for details on how this is done

Optional run-time lookup

Additional benefit: we can now also remove values from our map at run-time

```
static void clear()
{
  runtime_map.clear();
}

template <typename Lambda>
static void erase(Lambda lambda)
{
  runtime_map.erase(lambda());
}

static std::unordered_map<Key, std::unique_ptr<Value, ValueDeleter>> runtime_map;
```

➔ **Removes both from runtime_map and compile-time map**

Optional run-time lookup

And what about performance?

```
int& getAge()
{
  using map =
    semi::static_map<std::string, int>;

  return map::get(ID("age"));
}
```

```
cmpb $0, map::needs_init
jne initialise
movl map::storage, %eax
ret
.initialise:
.
Lots of code
.
```

Only executed the first time value is accessed

Compiler Explorer: https://gcc.godbolt.org/z/g7Rb4Z

6. But everything is static?

But everything is static?

```
template <typename Key, typename Value>
class static_map
{
public:
  static_map() = delete;

  template <typename Lambda>
  static Value& get(Lambda lambda);

  template <typename Lambda>
  static void erase(Lamnda lambda);

  static void clear();
  ...
};
```

- Storage needs to be static!
- So all methods declared static as well
- No per-instance storage

But everything is static?

Do you really need a non-static map?

- Caches are often used in singletons anyway

```
JavaMethod getMethod(const std::string& methodName)
{
  using cache = semi::static_map<std::string, JavaMethod    > cache;

  return cache::get(methodName);
}
```

Collision if semi-map with same Key/Value types exists anywhere else in the code

- Life-time can still be managed with `clear` method

But everything is static?

Do you really need a non-static map?

- Caches are often used in singletons anyway

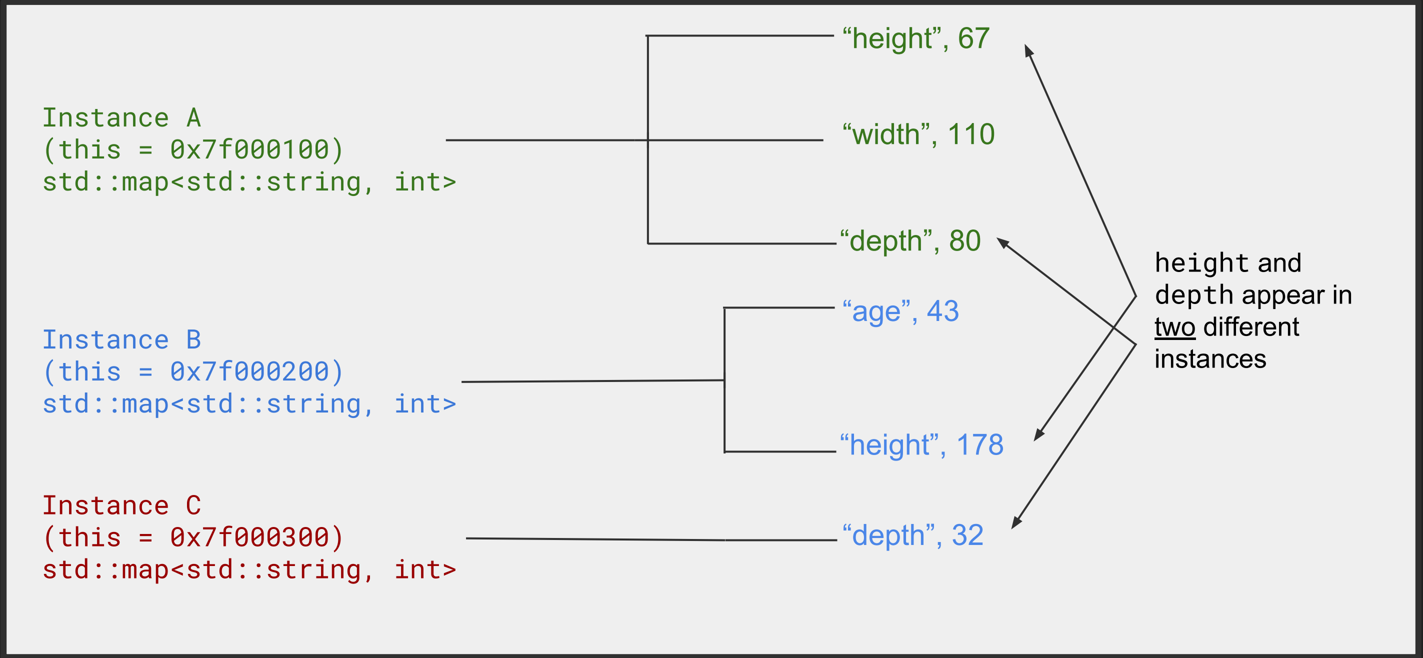
```
JavaMethod getMethod(const std::string& methodName)
{
  struct Tag {};
  using cache = semi::static_map<std::string, JavaMethod, Tag> cache;

  return cache::get(methodName);
}
```

- Life-time can still be managed with `clear` method

➔ **But I still really need a non-static map - is this possible?**

But everything is static?

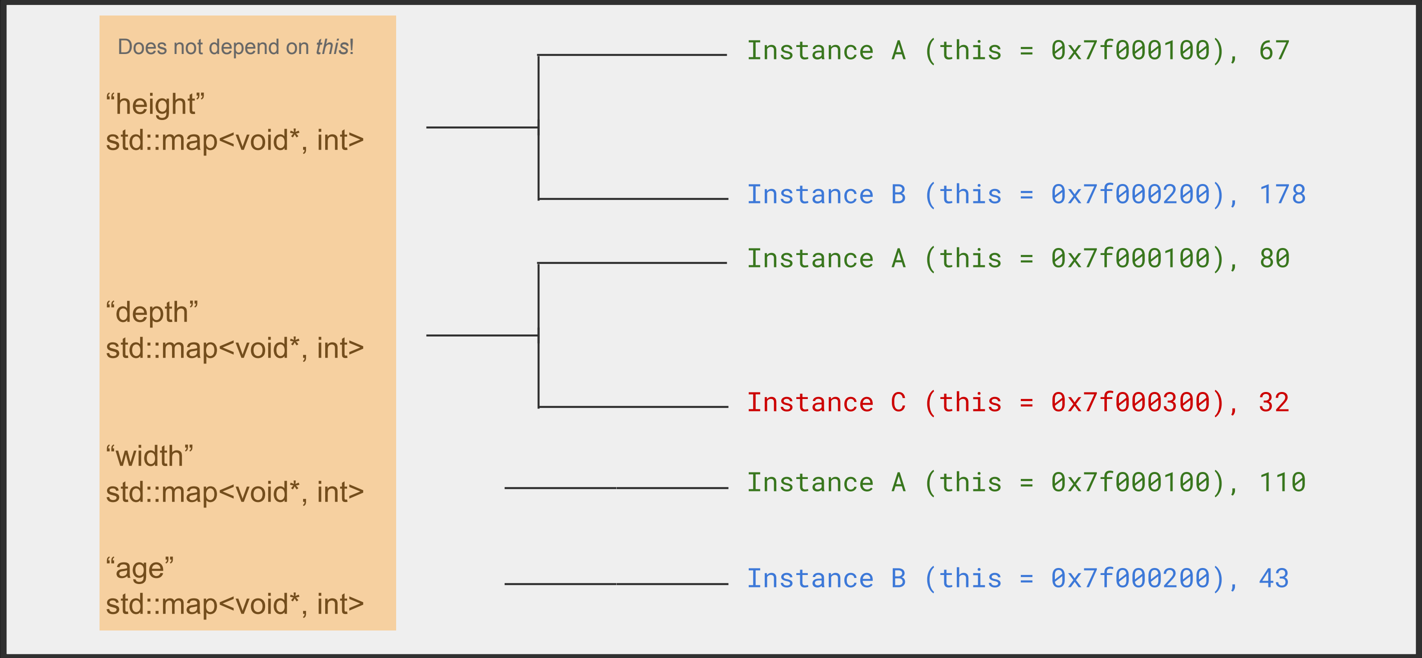
Normal runtime-maps: **each `this` pointer points to a map of keys**



Instances Key/Value pairs

But everything is static?

Invert the map: **each key points to a map of this pointers!**



Instances

Key/Value pairs

But everything is static?

Solution: **each key points to a map of this pointers!**

```
using map = semi::static_map<Key, Value>;
```

But everything is static?

Solution: **each key points to a map of this pointers!**

```
using map = semi::static_map<Key, std::map<void*, Value>>;
```

But everything is static?

Solution: **each key points to a map of this pointers!**

```
namespace semi
{
template <typename Key, typename Value>
class map
{
public:
  template <typename Lambda>
  Value& get(Lambda lambda)
  {
    return instance_map::get(lambda)[this];
  }
private:
  using instance_map = semi::static_map<Key, flat_map<void*, Value>>;
};
} // namespace semi

semi::map<std::string, std::string> m;
m.get(ID("conference")) = "cppcon";
```

Not static

But everything is static?

Invert the map: **each key points to a map of this pointers!**

```
namespace semi
{
template <typename Key, typename Value>
class map
{
public:
  template <typename Lambda>
  Value& get(Lambda lambda)
  {
    return instance_map::get(lambda)[this];
  }
private:
  using instance_map = semi::static_map<Key, std::map<void*, Value>>;
};
} // namespace semi

semi::map<std::string, std::string> m;
m.get(ID("conference")) = "cppcon";
```

Uses `static_map`

But everything is static?

Invert the map: **each key points to a map of this pointers!**

```
namespace semi
{
template <typename Key, typename Value>
class map
{
public:
  template <typename Lambda>
  Value& get(Lambda lambda)
  {
    return instance_map::get(lambda)[this];
  }
private:
  using instance_map = semi::static_map<Key, std::map<void*, Value>>;
};
} // namespace semi

semi::map<std::string, std::string> m;
m.get(ID("conference")) = "cppcon";
```

Use this parameter as the key

But everything is static?

But wait, doesn't that defeat the object?

- Calculating the hash of a memory address is often computationally more efficient compared to hashing other key types
- You normally have a large number of keys compared to the number of instances (especially true for caches)

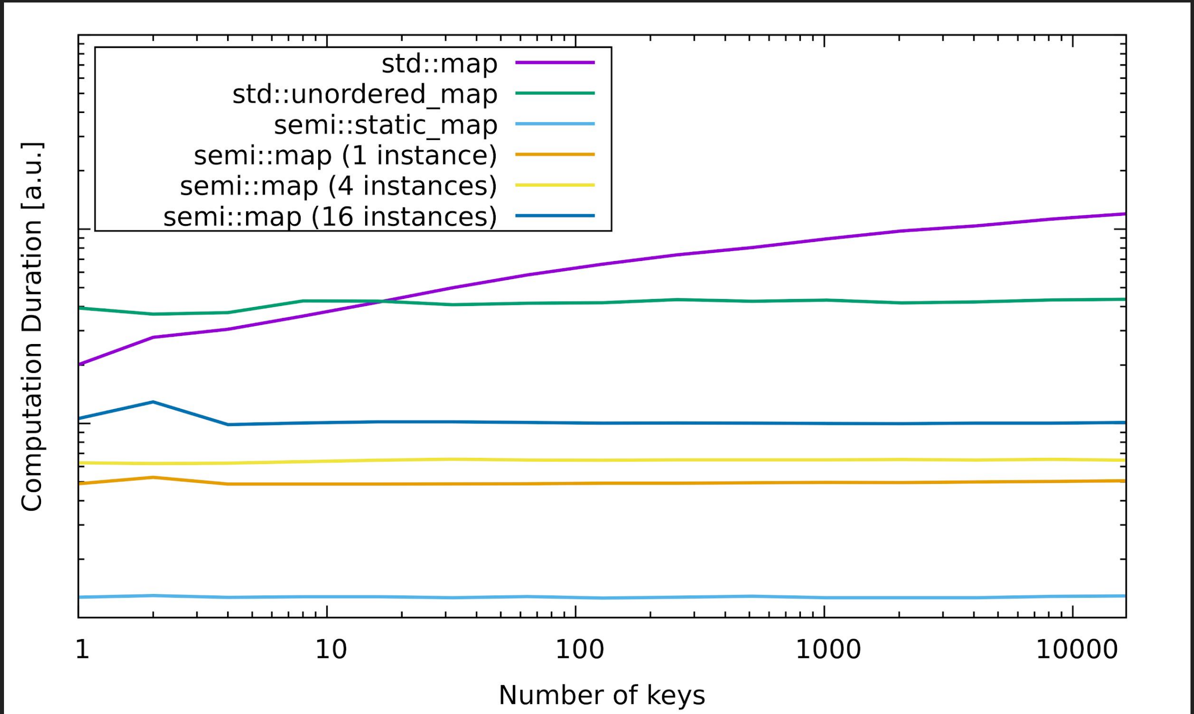
```
using map = semi::static_map<Key, std::map<void*, Value>[1]>;
```

Often only has 1-3 entries

[1]even better performance is achieved by using a flat map container type here which is used in the final benchmarked code

But everything is static?

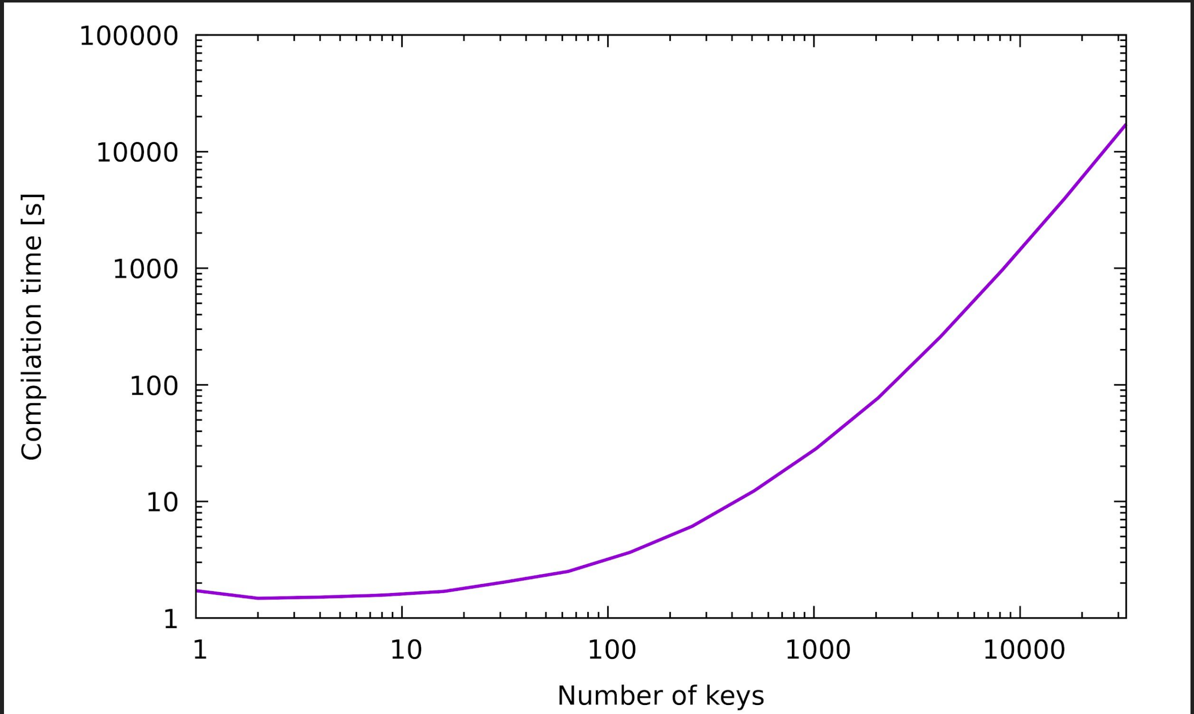
Benchmarks



Using: `clang++ -Ofast -march=native`

But everything is static?

Benchmarks



Thank you!

Code: https://github.com/hogliux/semimap
Video: https://www.youtube.com/watch?v=qNAbGpV1ZkU

@hogliux
fabian.renn@gmail.com

Optional run-time lookup

Only add `Value` pointer the first time `get_internal` is called

```
struct ValueWrapper
{
  ValueWrapper (const Key& key) { runtime_map[key] = &v; }
  Value v;
};
```

Faster: Only executed the first time get_internal is called!

```
template <typename>
static Value& get_internal (const Key& key)
{
  static ValueWrapper value(key);

  return value.v;
}
```

Optional run-time lookup

Getting a value from a runtime key is now trivial!

```
static Value& get(const Key& key)
{
  auto it = runtime_map.find(key);
  if (it != runtime_map.end())
    return it->second;

  return runtime_map.emplace_hint(it, key, {new Value, ValueDeleter{nullptr}})->second;
}
```

Same as before!

- Allocate value on heap if not already in runtime_map
- We don’t have an `init_flat` so pass `nullptr`

Managing key/value life-time

Closer look at the `initialise` function

```
void initialise(const Key& key, char* mem, bool& needs_init)
{
  auto it = runtime_map.find(key);

  if (it == runtime_map.end())
    runtime_map.try_emplace(key, new (mem) Value, ValueDeleter{&needs_init});
  else
    it->second
      = std::unique_ptr<Value, ValueDeleter> (new (mem) Value (std::move(it->second)),
                                               ValueDeleter{&needs_init});
}

struct ValueDeleter
{
  void operator()(Value* v)
  { if (needs_init != nullptr) { v->~Value(); *needs_init = true; }
    else                       { delete v; }
  }
  bool* needs_init = nullptr;
};
```

Putting it all together

What's the performance?

```cpp
std::string& getName()
{
  using map =
    semi::static_map<std::string, std::string>;

  return map::get(ID("name"));
}
```

```
movzx eax, BYTE PTR guard variable
test al, al
je .L15
mov eax, OFFSET FLAT:dummy_tlvalue
ret
.L15:
sub rsp, 8
mov edi, OFFSET FLAT:guard variable
call __cxa_guard_acquire
.
.
.
```

Only executed the first time value is accessed

Compiler Explorer: https://gcc.godbolt.org/z/KI8Ynk

➔ **Like any static local variable with non-default constructor, the compiler will generate lock guards**