

Google Summer of Code Proposal: Boost `static_map`

March 2017

Abstract

This document proposes an addition to **Boost C++ Libraries** – a *compile-time* hash table. There are multiple good implementations of unordered associate containers (e.g. `std::unordered_map`, Google's `sparsehash`). These implementations provide both lookup and insertion/deletion functionality. They are, however, not the perfect fit for the case when the contents of the container are fixed upon construction or even known at compile time. `std::vector` vs. `std::array` is a good analogy here. We propose a `static_map` associate container with focus on `constexpr` usage.

1 Personal Details

Name	Tom Westerhout
University	Radboud University of Nijmegen, The Netherlands
Degree Program	Bachelor in Physics
Email	<code>tom@rstdl-w.org</code>
Availability	I consider GSoC as a full-time job/internship, thus I plan to spend at least 40 hours per week working on the project. I have an exam in April and, in case I fail it, the retake is planned on the 24th of May. 29th (Monday) or 30th of May (the official start date) seems like a good starting point to me. My summer vacation officially starts only on the 1st of July, before that I have some courses to follow, exams to take and a thesis to defend. During this time I plan on working only 4 hours a day, including weekends. However, this does not seem like a big issue to me, because I have no other commitments in July and August and can easily compensate for the lost time. Working 8 hours a day Mon through Fri + 5 hours a day on Sat and Sun would result in more than 40 hours/week, if averaged over the whole work period. End August (the official end of coding) is a nice point to stop, because my summer vacation ends on the 1st of September.

2 Background Information

I'm currently in my third year of Bachelor in Physics program at Radboud University (The Netherlands) and plan on defending my thesis in June. Although I major in physics, I'm quite interested in programming, follow CS courses and plan on obtaining a Bachelor degree in CS within the next two years. By now, I've done the following project-related courses:

- *Imperative Programming 1 & 2* (Introduction to C++).
- *Hacking in C* (Memory layout, calling conventions and debugging).
- *Object Orientation*.
- *Algorithms & Datastructures*.
- *Languages and Automata*.
- *Functional Programming 1 & 2*.

For the last two years I've been working as a teaching assistant at Radboud University. I taught both physics and programming. In 2015 I taught C to first year physics and/or math majors; in 2016 — Python 3.

Lately, I've been writing code (as part of my Bachelor thesis) for distributed memory systems to perform some physical calculations. I work with quantum mechanical 2D fractal structures that don't exhibit translational symmetry, which makes calculations on them quite challenging. I've been calculating plasmonic properties of some materials using a newly developed method that some people believed to be impossible (too computationally expensive). We plan to publish a paper about it soon. Some code and documentation (both in early state of development) can be found on <https://github.com/twesterhout/plasmon-cpp/>.

Theoretical physicists are usually thinkers rather than doers. Thus in my spare time I've read some books about programming and C++ in particular, e.g. Scott Meyers' "Effective Modern C++", Andrei Alexandrescu's "Modern C++ Design", etc. This got me interested in the modern part of C++, metaprogramming and library development. Boost seems a perfect place to apply my knowledge and learn from more experienced people in this area, and the `static_map` project, i.e. "compile-time stuff" + "modern C++", is *exactly* what I'm most interested in at the moment. Although I haven't done any work related to this project, I believe myself to be a fast learner. Being highly motivated, I hope to pick up on the background quite soon.

In case I succeed with this project, I'd very much like to continue working in this area, contributing to Boost and become a part of the community.

As for my knowledge of the listed languages/technologies/tools, I'd rate it as follows:

C++ 98/03	≈ 2–3;
C++ 11/14	≈ 3–4;
C++ Standard Library	≈ 3–4;
Boost C++ Libraries	≈ 2–3;
Git	≈ 1–2.

I use Linux as my primary desktop operating system and have grown used to doing as much as possible via command line. Thus, usually I write code in Vim and use Makefiles to compile it. For documentation, I've been using Doxygen. But it's not the best choice when metaprogramming is involved. I'm open to learning DocBook as this is what Boost is using.

3 Project Proposal

Motivation There's often need for associative containers. C++ Standard Library provides `std::map` and `std::unordered_map`. Google has `dense_hash_map`. There are many more implementations. These all focus on both insertion/deletion and lookup. However, sometimes we're only interested in the lookup capabilities. For example, when the container is initialised from constant static data or even `constexpr` data. While a C-style associative array may serve well as a storage for such data, it's inefficient in terms of accessing the data by key. Common associative containers, on the other hand, implement efficient lookup, but use dynamic memory allocation for initialisation which may be undesired in some cases. The following example illustrates it further.

```
1  #include <iostream>
2  #include <initializer_list>
3  #include <map>
4  #include <unordered_map>
5  #include <experimental/string_view>
6  using std::experimental::string_view;
7
8  enum class weekday { sunday, monday, tuesday, wednesday
9                      , thursday, friday, saturday };
10
11 #define STRING_VIEW(str) string_view{ str, sizeof(str)-1 }
12 constexpr std::initializer_list<std::pair< const string_view
13         , weekday>> string_to_weekday
14     { { STRING_VIEW("sunday"),    weekday::sunday    }
15     , { STRING_VIEW("monday"),    weekday::monday    }
16     , { STRING_VIEW("tuesday"),   weekday::tuesday   }
17     , { STRING_VIEW("wednesday"), weekday::wednesday }
18     , { STRING_VIEW("thursday"),  weekday::thursday  }
19     , { STRING_VIEW("friday"),    weekday::friday    }
20     , { STRING_VIEW("saturday"),  weekday::saturday  }
21     };
22
23 int main(void)
24 {
25     {
26         // Calls malloc() at least 7 times
27         static const std::map<string_view, weekday>
28             to_weekday1 = string_to_weekday;
29         std::cout << "'monday' maps to "
30             << static_cast<int>(to_weekday1.at("monday"))
31             << std::endl;
32         std::cout << "'friday' maps to "
33             << static_cast<int>(to_weekday1.at("friday"))
34             << std::endl;
35         // Calls free() at least 7 times
36     }
37     {
38         // Calls malloc() at least 8 times
39         static const std::unordered_map<string_view, weekday>
40             to_weekday2 = string_to_weekday;
41         std::cout << "'monday' maps to "
42             << static_cast<int>(to_weekday2.at("monday"))
43             << std::endl;
44         std::cout << "'friday' maps to "
45             << static_cast<int>(to_weekday2.at("friday"))
46             << std::endl;
47         // Calls free() at least 8 times
48     }
49     return 0;
50 }
```

Even though the `string_to_weekday` is `constexpr`, to make use of fast lookups ($\mathcal{O}(1)$ for `unordered_map` and $\mathcal{O}(\log(N))$ for `map` compared to $\mathcal{O}(N)$ of linear searching) we have to copy data at runtime, which involves dynamic memory allocation. And although one may implement stack allocator to prevent dynamic memory allocation for small datasets, initialisation and lookups will still happen at runtime, because they are not marked `constexpr`.

With C++14 relaxed constraints on `constexpr` functions it is possible to implement an associative container with `constexpr` (whenever possible) key lookups (as a proof, see the toy implementation of such a container in Appendix A). We think that a high quality implementation of such a container would be a worthwhile addition to Boost.

Proposal As part of the Google Summer of Code 2017 project we propose to do the following:

- To seek consensus from the Boost Developer’s mailing list on a suitable design of a `static_map` class with the following design features:
 1. The number of key–value pairs is fixed upon construction.
 2. All features can be used in constant expressions, i.e. all member functions are marked `constexpr`.
 3. The container can be statically initialised in the mind the compiler or global static storage.
 4. Values, though neither keys nor the number of items, are modifiable.
 5. The container performs `constexpr` key lookups to the “maximum possible extent”.

This list of requirements is harder to implement than it may seem at first sight. Consider the following example:

```

2  constexpr std::pair<int, char const*>
   map_data[] = { { 5, "apple" },
4               , { 8, "pear" },
               , { 0, "banana" }
6               };

6  // Generates no runtime code. Easy, because the standard requires this to be executed
8  // at compile-time.
   constexpr auto cmap = make_static_map(map_data);
10  constexpr char const* what_is_5 = cmap[5];

12  // Generates no runtime code. Easy, because 0 is a non-type template parameter.
   char const* what_is_0 = std::get<0>(cmap);
14

16  // Challenging: should only generate code loading immediately from a memory location.
   // It must NOT generate any additional runtime overhead like hashing or searching.
   char const* what_is_8 = cmap[8];
18

20  // Challenging: again, should only generate code loading from memory.
   auto cmap2 = make_static_map(map_data);
   auto& at_0 = map_data[0];
22  at_0 = "orange";

```

If all inputs for a `constexpr` operation are constant expressions, but the result is not used as a constant expression, then the compiler is *not required* to execute the operation at compile-time. This is what makes lines 19, 23 and 24 so challenging.

- To implement a `static_map` unordered associative container class which satisfies the above outlined requirements on top of the requirements in the standard. This includes the implementation of utility classes/functions for `constexpr` string comparison and hashing. The implementation should work on at least two major compilers in, hopefully, C++14 mode.
- To implement a comprehensive unit test suite for the `static_map` class with focus on ensuring no runtime overhead for the challenging cases mentioned above (i.e. when the result of a `constexpr` function is not used as a constant expression).
- To configure a per-commit continuous integration for the unit test suite on at least one of the major public open source CI services (e.g. Travis, Appveyor).
- To write documentation to Boost quality levels for the `static_map` container class. This includes time and space complexity guarantees and benchmarks, and exception guarantees for each function in the API and each use of each function.

Schedule We propose the following schedule to achieve the goals outlined above:

29th May — 5th June	2nd June — 9th June	Consult Boost Developer’s mailing list and do some background reading to come up with a suitable design for <code>static_map</code> class and all required utility classes. In the meantime configure the system for development, i.e. get compilers, CI service, doc tools, etc., working.
12th June — 19th June — 26th June	16th June — 23th June — 30th June	Implement <code>static_map</code> class for the simple case of keys being of integral type. Start writing tests and documentation. It would be nice to have a working implementation (for this simple case) before the first phase evaluation deadline.
3rd July — 10th July — 17th July — 24th July	7th July — 14th July — 21th July — 28th July	Complete the implementation of <code>static_map</code> class and utility classes. Continue working on tests and documentation. It would be nice to have a fully working (on at least two compilers) implementation before the second phase evaluation deadline.
31st July — 7th August	4th August — 11th August	Finish implementing tests. Finish the documentation. Maybe do some work to support other compilers.
14th August	18th August	Extra
21st August	25th August	Final touches, submitting results.

4 Programming Competency

The initial objective of was to reimplement the toy `static_map` provided at <https://goo.gl/e070oa> so that GCC or VS2017 would execute the non-`constexpr` assignment at compile time.

My first idea was to use SFINAE to detect whether parameters are `constexpr`, and if they are, store the result of the lookup in a `constexpr` variable, thus forcing the compiler to evaluate the operation at compile-time. Detecting `constexpr`ness of parameters, however, proved to be impossible. One could write a macro to detect whether an expression passed to it is actually a constant expression (as it is done in the following answer: <http://stackoverflow.com/a/40413051>). More or less what GCC’s `__builtin_constant_p` does. Unfortunately, this only works with one level of indirection:

```
constexpr auto factorial(std::size_t const n) -> std::size_t
2 {
3     if (n == 0 || n == 1) return 1;
4     return n * factorial(n - 1);
5 }
6
7 constexpr auto foo(std::size_t const n)
8 {
9     return __builtin_constant_p(factorial(n));
10 }
11
12 int main(void)
13 {
14     static_assert(__builtin_constant_p(factorial(10)), "");
15     // This fails!
16     static_assert(foo(10), "");
17 }
```

I’ve thus come to the conclusion that there is no standard way to force the compiler to evaluate the expression at compile time, we have to rely on compiler’s “kindness” to optimise it. After some trial and error I’ve figured that GCC does not like the following:

1. “Difficult” exceptions such as `std::out_of_range`, `std::runtime_error`, etc. I call these exceptions “difficult”, because (at least in GCC-6.3) the error message is stored in a `class/struct` with non-constexpr operations.
2. `throw` statements in the function body. I.e. in the following code snippet

```

1 constexpr auto foo1()
2 {
3     auto x = compute_something();
4     if (good_answer(x)) return x;
5     throw std::runtime_error{""};
6 }
7
8 constexpr auto foo2()
9 {
10    auto x = compute_something();
11    return good_answer(x)
12        ? x
13        : (throw std::runtime_error{""}, x);
14 }
```

GCC finds `foo2` easier to optimise. However, this is purely empirical, and compared to Item 1 I don’t have a theoretical explanation for it.

3. Multiple return statements. This works well in combination with Item 2. For example, consider the `static_map::at` function in the toy implementation that was provided:

```

1 constexpr auto at(key_type const& k) const -> mapped_type const&
2 {
3     for (size_t n = 0; n < N; n++)
4         if (_values[n].second.first == k)
5             return _values[n].second.second;
6     throw key_not_found_error{};
7 }
```

I’ve already replaced the “difficult” `std::out_of_range` exceptions with an “easy” one. This doesn’t help. GCC-6.3 still produces runtime code for this function. However, if we follow the rules of Items 2 and 3 and replace the original function with

```

1 constexpr auto at(key_type const& k) const -> mapped_type const&
2 {
3     size_t n = 0;
4     while (n < N && _values[n].second.first != k) {
5         ++n;
6     }
7     return n != N
8         ? _values[n].second.second
9         : (throw key_not_found_error{}, _values[0].second.second);
10 }
```

GCC-6.3 optimises the function completely, i.e. no runtime code is produced for it. To prove that Item 1 is useful, substitute `key_not_found_error{}` by `std::out_of_range{"Key not found."}` — runtime code is back.

It was a matter of following the described rules of thumb to satisfy the requirements of the competency test. From Niall Douglas I’ve learned that the test has outgrown its initial requirements and a complete toy implementation of the proposal was now desired.

Adding support for iterators and custom hash and equals functions is a matter of doing. The only challenging part is initialisation of the `static_map`. In the original solution initialisation went as follows: the user passes a C-style array to `make_static_map` function, it expands the array into a parameter pack and passes it to the `static_map` constructor which

initialises the underlying storage type (again a C-array) using list initialisation. To make use of hashing, elements in the underlying array must appear in a certain order. Sorting the array after initialisation seems like a bad solution, because keys are constant. Storing keys as non-const would make `static_map` inconsistent with other associate containers, and using `const_casts` seems like a dirty hack in this case. The good place to sort the array would be the `make_static_map` function. In the original implementation this function expands the array `il` using `std::index_sequence<_Is...>` as `il[_Is]...`. Thus sorting can be achieved by expanding `il` as `il[σ (_Is)]...`, where σ is some permutation. My implementation of this idea can be found in Appendix A.

Another interesting thing about the original implementation is that it uses an array of `std::pair<std::size_t, std::pair<key_type const, mapped_type>>` as underlying storage. I found the extra `std::size_t` unnecessary and decided to use an array of `std::pair<key_type const, mapped_type>` as the underlying storage. But then, as indicated by Niall, my solution stopped working with Clang. Thus I've decided to follow the "convention" used in `libstdc++` and created an inner storage class that allows the `static_map` be implemented independent of the type of underlying storage type. Addition of this inner class turned out to be crucial as now even Clang was happy to optimise code without the extra `std::size_t`.

My complete toy implementation of `static_map` can be found in Appendix A or online on https://github.com/twesterhout/GSoC17-proposal/blob/master/static_map.cpp (easier to copy-paste into Compiler Explorer). The "challenging cases" work with both GCC-6+ and Clang-3.9+.

A Code

```
2 #include <utility>
3 #include <cassert>
4 #include <stdexcept>
5 #include <type_traits>
6 #include <experimental/string_view>
7
8 struct key_not_found_error : std::exception {
9     auto what() const noexcept -> char const* override
10     { return _msg; }
11 private:
12     static constexpr char const* _msg = "Key not found.";
13 };
14
15 constexpr char const* key_not_found_error::_msg;
16
17 namespace {
18
19 template < class _Static_map
20             , bool _is_const
21             >
22 struct map_iterator {
23     using type = map_iterator<_Static_map, _is_const>;
24     using difference_type = typename _Static_map::difference_type;
25     using value_type =
26         std::conditional_t< _is_const
27                             , std::add_const_t<typename _Static_map::value_type>
28                             , typename _Static_map::value_type >;
29     using iterator_category = std::random_access_iterator_tag;
30     using reference =
31         std::conditional_t< _is_const
32                             , typename _Static_map::const_reference
33                             , typename _Static_map::reference >;
34     using pointer = std::add_pointer_t<value_type>;
35     using size_type = typename _Static_map::size_type;
36
37 private:
38     using _const_iterator = map_iterator<_Static_map, /*is_const =*/ true>;
39     using _storage_type =
40         std::conditional_t< _is_const
41                             , std::add_const_t<typename _Static_map::storage_type>
42                             , typename _Static_map::storage_type >;
43
44     static constexpr size_type _size = _Static_map::size();
45     _storage_type* _data;
46     size_type _i;
47
48     constexpr auto _to_same_object( map_iterator const& x
49                                     , map_iterator const& y ) const noexcept -> bool
50     { return x._data == y._data; }
51
52 public:
53     constexpr map_iterator() noexcept : _data{ nullptr }, _i{ 0 } {}
54
55     constexpr map_iterator(_storage_type& data, size_type const pos) noexcept
56     : _data{ &data }, _i{ pos }
57     {}
58
59     constexpr map_iterator(map_iterator const& other) noexcept = default;
60     constexpr map_iterator& operator=(map_iterator const& other) noexcept = default;
61
62     __attribute__((always_inline))
63     constexpr auto operator*() const -> reference
64     {
65         return _data != nullptr
66             ? ( _i < _size
67                 ? _data->value(_i)
68                 : (throw std::out_of_range{"Iterator not dereferenceable."}, _data->value(0))
69             )
70             : (throw std::invalid_argument{"Iterator not dereferenceable."}, _data->value(0));
71     }
72
73     constexpr auto operator->() const -> pointer { return &(*this); }
74
75     constexpr auto operator++() noexcept -> map_iterator& { ++_i; return *this; }
76
77     constexpr auto operator++(int) noexcept -> map_iterator
78     {
79         map_iterator saved{ *this };
80         ++(*this);
81         return saved;
82     }
83
84     constexpr auto operator-(map_iterator const& other) const -> difference_type
85     {
86         return _to_same_object(*this, other)
87             ? (static_cast<difference_type>>(_i) - static_cast<difference_type>(other._i))
88             : (throw std::invalid_argument{"Iterators not comparable."}, 0);
89     }
90 }
```



```

92 constexpr auto operator==(map_iterator const& other) const noexcept -> bool
{ return _to_same_object(*this, other) && (_i == other._i); }

94 constexpr auto operator!=(map_iterator const& other) const noexcept -> bool
{ return !(*this == other); }

96 constexpr auto operator< (map_iterator const& other) const -> bool
{ return (*this - other) < 0; }

98 constexpr auto operator<= (map_iterator const& other) const -> bool
{ return (*this - other) <= 0; }

100 constexpr auto operator> (map_iterator const& other) const -> bool
{ return (*this - other) > 0; }

102 constexpr auto operator>= (map_iterator const& other) const -> bool
{ return (*this - other) >= 0; }

104 constexpr auto operator[] (difference_type const n) const -> reference
{ return *(*this + n); }

106 constexpr operator _const_iterator() const noexcept { return {_data, _i}; }

108 friend
110 constexpr auto swap(map_iterator& lhs, map_iterator& rhs) noexcept -> void
{
112     auto const _temp_idx = lhs._i;
114     lhs._i = rhs._i;
116     rhs._i = _temp_idx;

118     auto const _temp_data = lhs._data;
120     lhs._data = rhs._data;
122     rhs._data = _temp_data;
124 }

126 friend
128 constexpr auto operator+ ( map_iterator const& x
, difference_type const n ) noexcept -> map_iterator
{ return {(x._data), x._i + n}; }

130 friend
132 constexpr auto operator+ ( difference_type const n
, map_iterator const& x ) noexcept -> map_iterator
{ return x + n; }

134 friend
136 constexpr auto operator- ( map_iterator const& x
, difference_type const n ) noexcept -> map_iterator
{ return {(x._data), x._i - n}; }

138 friend
140 constexpr auto operator+=( map_iterator & x
, difference_type const n ) noexcept -> map_iterator&
{ return x = x + n; }

142 friend
144 constexpr auto operator-=( map_iterator & x
, difference_type const n ) noexcept -> map_iterator&
{ return x = x - n; }
146 };

148 template <class _Static_map, bool _is_const>
150 constexpr typename _Static_map::size_type map_iterator<_Static_map, _is_const>::_size;
152 } // unnamed namespace
154
156 struct equal_c {
158     template <class _Char>
160     constexpr auto operator()(_Char const* const a, _Char const* const b) const
162     noexcept( noexcept(std::declval<_Char>() == std::declval<_Char>())
&& noexcept(std::declval<_Char>() != std::declval<_Char>()) )
    {
164         std::size_t i = 0;
166         while (a[i] != _Char{} && a[i] == b[i]) {
168             ++i;
170         }
172         return a[i] == _Char{} && b[i] == _Char{};
174     }
176 };

178 namespace {
180     template <class _Char>
182     inline constexpr auto length_c(_Char const* const str) noexcept -> std::size_t
184     {
186         std::size_t i = 0;
188         while (str[i] != _Char{}) {
190             ++i;
192         }
194         return (i == 0) ? 0 : (i - 1);
196     }
198 }

```

```

186 template <class _Char>
inline constexpr auto crc32_hash(_Char const* const str) noexcept -> std::uint32_t
{
188     constexpr std::uint32_t INITXOR = 0xFFFFFFFF;
constexpr std::uint32_t FINALXOR = 0xFFFFFFFF;
190     constexpr std::uint32_t CRCPOLY = 0xEDB88320;

192     auto const l = length_c(str);
auto crcreg = INITXOR;
194
196     for (std::size_t j = 0; j < l; ++j) {
auto b = static_cast<std::make_unsigned_t<_Char>>(str[j]);
        for (std::size_t i = 0; i < 8 * sizeof(_Char); ++i) {
198             if ((crcreg ^ b) & 1) {
                crcreg = (crcreg >> 1) ^ CRCPOLY;
200             }
            else {
202                 crcreg >>= 1;
            }
204             b >>= 1;
        }
206     }

208     return crcreg ^ FINALXOR;
}

210 template <class _Char>
212 inline constexpr auto simple_hash(_Char const* const str) noexcept
{
214     std::size_t hash = 0;
std::size_t i = 0;
216     while (str[i] != _Char{}) {
        hash = 37 * hash + str[i];
218         ++i;
    }
220     return hash;
}

222 } // unnamed namespace
224

226 template <class _T, std::size_t _N, class = void>
struct hash_c;

228 template <class _T, std::size_t _N>
230 struct hash_c<_T, _N, std::enable_if_t<std::is_integral<_T>::value>> {
    constexpr auto operator()(const _T const* const x) const noexcept -> std::size_t
232     { return static_cast<std::size_t>(x) % _N; }
};

234 template <std::size_t _N>
236 struct hash_c<char const*, _N> {
    constexpr auto operator()(const char const* const x) const noexcept -> std::size_t
238     {
        // return simple_hash(x) % _N;
240         return static_cast<std::size_t>(crc32_hash(x)) % _N;
    }
};

242 };

244 template <std::size_t _N>
246 struct hash_c<std::experimental::string_view, _N> {
    constexpr auto operator()(std::experimental::string_view const& x) const noexcept
248     { return hash_c<char const*, _N>{}(x.data()); }
};

250 template < class _RAIterator, class _Pred>
252 inline constexpr auto find_if_c(_RAIterator&& first, _RAIterator&& last, _Pred&& p )
noexcept( noexcept(std::declval<_RAIterator>() - std::declval<_RAIterator>())
254     && noexcept(std::declval<_RAIterator>() = std::declval<_RAIterator>())
&& noexcept(*std::declval<_RAIterator>())
256     && noexcept(++std::declval<_RAIterator>()) )
{
258     auto const _count = last - first;
auto i = first;
260     std::remove_const_t<decltype(_count)> n = 0;
while (n < _count && !p(*i)) {
262         ++n; ++i;
    }
264     return n == _count ? last : i;
}

266

268 namespace {
template <bool... _Bs> struct _all;
270 template <> struct _all<> : std::true_type {};

272 template <bool _B, bool... _Bs>
struct _all<_B, _Bs...> : std::conditional_t<_B, _all<_Bs...>, std::false_type> {};
274 } // unnamed namespace

276
278 template < class _Key
, class _Tp

```

```

280         , size_t _N
281         , class _Pred = std::equal_to<_Key>
282         , class _Hasher = hash_c<_Key, _N>
283     >
284     class static_map {
285     public:
286         using type = static_map<_Key, _Tp, _N, _Pred, _Hasher>;
287         using key_type = _Key;
288         using mapped_type = _Tp;
289         using key_equal = _Pred;
290         using hasher = _Hasher;
291         using value_type = std::pair<key_type const, mapped_type>;
292         using reference = value_type &;
293         using const_reference = value_type const&;
294         using iterator = map_iterator<type, /*is_const =*/ false>;
295         using const_iterator = map_iterator<type, /*is_const =*/ true>;
296         using difference_type = std::ptrdiff_t;
297         using size_type = std::size_t;
298     private:
299         struct _Storage {
300         private:
301             using _actual_value_type = value_type;
302             _actual_value_type _vs[_N];
303
304             static constexpr auto make_node(const_reference value)
305             noexcept( std::is_nothrow_copy_constructible<key_type>::value
306                       && std::is_nothrow_copy_constructible<mapped_type>::value )
307             { return std::make_pair(value.first, value.second); }
308
309         public:
310             template <class... _Pair>
311             constexpr _Storage(_Pair const&... values)
312             noexcept( _all<noexcept(make_node(std::declval<_Pair const&>()))...>::value )
313             : _vs{ make_node(values)... }
314             {}
315
316             constexpr auto value(size_type const i) const noexcept -> const_reference
317             { return _vs[i]; }
318
319             constexpr auto value(size_type const i) noexcept -> reference
320             { return _vs[i]; }
321         };
322     public:
323         using storage_type = _Storage;
324     private:
325         key_equal const _eq;
326         hasher const _hf;
327         storage_type _data;
328     private:
329         __attribute__((always_inline))
330         constexpr auto _lookup(key_type const& k) const
331         noexcept( noexcept(std::declval<hasher>()(std::declval<key_type>()))
332                 && noexcept(std::declval<key_equal>()(std::declval<key_type>())
333                           , std::declval<key_type>())) )
334         {
335             auto const guess = _hf(k);
336             auto i = guess;
337
338             while (i < _N && !_eq(_data.value(i).first, k)) { ++i; }
339             if (i == _N) {
340                 i = 0;
341                 while (i < guess && !_eq(_data.value(i).first, k)) { ++i; }
342                 if (i == guess) return _N;
343             }
344             return i;
345         }
346     public:
347         template <class... _K, class... _V>
348         constexpr static_map(std::pair<_K, _V> const&... values)
349         noexcept( std::is_nothrow_constructible< storage_type
350                 , std::pair<_K, _V> const&...>::value
351                 && std::is_nothrow_default_constructible<key_equal>::value
352                 && std::is_nothrow_default_constructible<hasher>::value )
353         : _eq{ }, _hf{ }, _data{ values... }
354         {}
355
356         template <class... _K, class... _V>
357         constexpr static_map( key_equal equal, hasher hash_function
358                               , std::pair<_K, _V> const&... values )
359         noexcept( std::is_nothrow_constructible< storage_type
360                 , std::pair<_K, _V> const&...>::value
361                 && std::is_nothrow_move_constructible<key_equal>::value
362                 && std::is_nothrow_move_constructible<hasher>::value )
363         : _eq{ std::move(equal) }
364         , _hf{ std::move(hash_function) }
365         , _data{ values... }
366         {}

```

```

374 static constexpr auto size() noexcept -> size_type
375 { return _N; }
376
377 constexpr auto begin() noexcept -> iterator
378 { return {_data, 0}; }
379
380 constexpr auto end() noexcept -> iterator
381 { return {_data, size()}; }
382
383 constexpr auto cbegin() const noexcept -> const_iterator
384 { return {_data, 0}; }
385
386 constexpr auto cend() const noexcept -> const_iterator
387 { return {_data, size()}; }
388
389 __attribute__((always_inline))
390 constexpr auto find(key_type const& k) const
391     noexcept(noexcept(std::declval<static_map>().lookup(std::declval<key_type>())))
392     -> const_iterator
393 {
394     auto i = lookup(k);
395     return i == _N
396         ? cend()
397         : const_iterator{_data, i};
398 }
399
400 constexpr auto count(key_type const& k) const
401     noexcept(noexcept(std::declval<static_map>().find(std::declval<key_type>())))
402     -> size_type
403 { return find(k) == cend() ? 0 : 1; }
404
405 __attribute__((always_inline))
406 constexpr auto at(key_type const& k) const -> mapped_type const&
407 {
408     auto i = lookup(k);
409     return i != _N
410         ? _data.value(i).second
411         : (throw key_not_found_error{}, _data.value(0).second);
412 }
413
414 __attribute__((always_inline))
415 constexpr auto at(key_type const& k) -> mapped_type&
416 {
417     auto i = lookup(k);
418     return i != _N
419         ? _data.value(i).second
420         : (throw key_not_found_error{}, _data.value(0).second);
421 }
422
423 constexpr auto operator[](key_type const& k) const -> mapped_type const&
424 { return at(k); }
425
426 constexpr auto operator[](key_type const& k) -> mapped_type &
427 { return at(k); }
428 };
429
430
431 namespace {
432
433 template <std::size_t _N>
434 __attribute__((always_inline))
435 inline constexpr auto _insert( std::size_t const index, std::size_t const guess
436                               , std::size_t (&indices)[_N] ) noexcept
437 {
438     auto i = guess;
439     while (i < _N && indices[i] != _N) { ++i; }
440     if (i == _N) {
441         i = 0;
442         while (i < guess && indices[i] != _N) { ++i; }
443     }
444     indices[i] = index;
445 }
446
447 template <std::size_t _N>
448 __attribute__((always_inline))
449 inline constexpr auto _init_impl( std::size_t (&indices)[_N]
450                                , std::pair<std::size_t, std::size_t> const& x ) noexcept
451 { _insert(x.first, x.second, indices); }
452
453 template <std::size_t _N, class... _I>
454 __attribute__((always_inline))
455 inline constexpr auto _init_impl( std::size_t (&indices)[_N]
456                                , std::pair<std::size_t, std::size_t> const& x
457                                , std::pair<_I, _I> const&... xs ) noexcept
458 {
459     _insert(x.first, x.second, indices);
460     _init_impl(indices, xs...);
461 }
462
463 template <std::size_t, class _T>

```

```

466 using _transform = _T;

468 template < class _K
         , class _V
470         , class _Pred
         , class _Hasher
472         , std::size_t _N
         , std::size_t... _Is
474         >
__attribute__((always_inline))
476 inline constexpr auto _initialise( std::pair<_K const, _V> const (&il)[_N]
                                     , _Pred&& equal, _Hasher&& hf
                                     , std::index_sequence<_Is...> )
478     noexcept( noexcept( std::declval<_Hasher>()(std::declval<_K>()) )
               && noexcept( std::is_nothrow_constructible
480                           < static_map<_K, _V, _N, _Pred, _Hasher>
                           , _Pred, _Hasher, _transform<_Is, std::pair<_K const, _V> const&>...
                           >::value) ) )
482 {
484     std::size_t indices[_N] = {((void)_Is, _N)...};
486     _init_impl(indices, std::make_pair(_Is, hf(il[_Is].first))...);

488     return static_map<_K, _V, _N, _Pred, _Hasher>{ std::forward<_Pred>(equal)
490                                                     , std::forward<_Hasher>(hf)
492                                                     , il[indices[_Is]]... };
494 }

496 // unnamed namespace
498 template < class _K
         , class _V
         , std::size_t _N
         , class _Pred = std::equal_to<_K>
         , class _Hasher = hash_c<_K, _N>
500         >
__attribute__((always_inline))
502 inline
constexpr
504 auto make_static_map( std::pair<_K const, _V> const (&il)[_N]
                      , _Pred&& equal = _Pred{}
                      , _Hasher&& hf = _Hasher{} )
506     noexcept( noexcept( _initialise( std::declval<decltype(il)>()
                      , std::declval<_Pred>()
                      , std::declval<_Hasher>()
508                      , std::make_index_sequence<_N>{} ) ) ) )
510 {
512     return _initialise( il
514                       , std::forward<_Pred>(equal)
516                       , std::forward<_Hasher>(hf)
518                       , std::make_index_sequence<_N>{} );
520 }

522 enum class weekday { sunday
524 , monday
526 , tuesday
528 , wednesday
530 , thursday
532 , friday
534 , saturday };

536 using std::experimental::string_view;
538 #define STRING_VIEW(str) std::experimental::string_view{str, sizeof(str)-1}

540 struct equal_string_view {
542     constexpr
544     auto operator()( string_view const& a
                      , string_view const& b ) const noexcept -> bool
546     { return equal_c{}(a.data(), b.data()); }
548 };

550 int main(void)
552 {
554     // Compile-time stuff
556     constexpr std::pair<int const, const char *> map_data[] =
558     { { 5, "apple" }
      , { 8, "pear" }
      , { 0, "banana" }
      };
    // Initialisation
    constexpr auto cmap = make_static_map(map_data);

    // No abort call in assembly
    if (!cmap[8]) abort();

    // These operations use hashing
    static_assert(equal_c{}(cmap[5], "apple"), "");
    static_assert(equal_c{}(cmap[8], "pear"), "");
    static_assert(equal_c{}(cmap[0], "banana"), "");

    // This fails to compile -- as expected.
    // constexpr auto foo = cmap[-1];

```

```

560 // We need this because GCC does not support constexpr lambdas, yet.
561 struct is_eight {
562     constexpr auto operator()(std::pair<int, char const*> const& x)
563     { const noexcept
564         { return x.first == 8; }
565     };
566
567     // Iterators
568     static_assert( find_if_c(cmap.cbegin(), cmap.cend(), is_eight{})
569         != cmap.cend(), "" );
570 }
571 {
572     // Run-time stuff
573     constexpr std::pair<int const, const char *> map_data[] =
574         { { 5, "apple" }
575         , { 8, "pear" }
576         , { 0, "banana" }
577         };
578     auto cmap = make_static_map(map_data);
579
580     // No abort in assembly
581     if (!cmap.count(5)) abort();
582
583     // Values are mutable !!
584     auto& i = cmap.at(8);
585     i = "orange";
586     assert(equal_c{}(i, "orange"));
587
588     // Iterators
589     auto const it1 = cmap.find(8);
590     assert(it1->first == 8);
591 }
592 {
593     // Working with string_view
594     constexpr std::pair<const std::experimental::string_view, weekday>
595         string_to_weekday[]
596         { { STRING_VIEW("sunday"),    weekday::sunday }
597         , { STRING_VIEW("monday"),    weekday::monday }
598         , { STRING_VIEW("tuesday"),   weekday::tuesday }
599         , { STRING_VIEW("wednesday"), weekday::wednesday }
600         , { STRING_VIEW("thursday"),  weekday::thursday }
601         , { STRING_VIEW("friday"),    weekday::friday }
602         , { STRING_VIEW("saturday"),  weekday::saturday }
603         };
604     constexpr auto to_weekday = make_static_map( string_to_weekday
605         , equal_string_view{} );
606
607     static_assert(to_weekday[STRING_VIEW("sunday")] == weekday::sunday, "");
608     static_assert(to_weekday[STRING_VIEW("monday")] == weekday::monday, "");
609     static_assert(to_weekday[STRING_VIEW("tuesday")] == weekday::tuesday, "");
610     static_assert(to_weekday[STRING_VIEW("wednesday")] == weekday::wednesday, "");
611     static_assert(to_weekday[STRING_VIEW("thursday")] == weekday::thursday, "");
612     static_assert(to_weekday[STRING_VIEW("friday")] == weekday::friday, "");
613     static_assert(to_weekday[STRING_VIEW("saturday")] == weekday::saturday, "");
614
615     if (!to_weekday.count(STRING_VIEW("sunday"))) abort();
616
617     struct is_friday {
618         constexpr auto operator()(std::pair< string_view const
619             , weekday > const& x)
620         { const noexcept -> bool
621             { return equal_c{}(x.first.data(), "friday"); }
622         };
623
624     // Lookup
625     static_assert(to_weekday.find(STRING_VIEW("friday")) != to_weekday.cend(), "");
626     static_assert(to_weekday.find(STRING_VIEW("__friday__")) == to_weekday.cend(), "" );
627     // Linear search
628     static_assert( find_if_c( to_weekday.cbegin(), to_weekday.cend()
629         , is_friday{} ) != to_weekday.cend(), "" );
630 }
631 {
632     // C-strings
633     constexpr std::pair<char const* const, weekday>
634         string_to_weekday[]
635         { { "sunday",    weekday::sunday }
636         , { "monday",    weekday::monday }
637         , { "tuesday",   weekday::tuesday }
638         , { "wednesday", weekday::wednesday }
639         , { "thursday",  weekday::thursday }
640         , { "friday",    weekday::friday }
641         , { "saturday",  weekday::saturday }
642         };
643     constexpr auto to_weekday = make_static_map( string_to_weekday
644         , equal_c{} );
645 }
646 return 0;
}

```

static_map.cpp