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` – an 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	`tom@rstl-w.org`
Availability	I am available from the 30th of May till the 30th of August. 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 of 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 am 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 am a Physics major, I am also passionate about programming. I plan to obtain a Bachelors in Computer Science while doing a Masters program in Physics. So far, I have completed 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 have 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 math majors; in 2016 — `Python 3`.

Lately, I have been writing code (as part of my Bachelor thesis) for distributed memory systems to perform physical calculations. I work with quantum mechanical 2D fractal structures that don't exhibit translational symmetry, which makes calculations on them quite challenging. I have 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/`.

In my spare time I have read some books about programming and `C++` in particular, e.g. Scott Meyers' "Effective Modern C++", Andrei Alexandrescu's "Modern C++ Design". 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 am 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.

I would very much like to continue my work in this area even after succeeding with this project, contribute to `Boost` and become a part of the community.

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

`C++ 98/03` ≈ 3;

`C++ 11/14` ≈ 4;

`C++ Standard Library` ≈ 4;

`Boost C++ Libraries` ≈ 3;

`Git` ≈ 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 `Makefile`s to compile it. For documentation, I have been using `Doxygen`. But it is not the best choice when metaprogramming is involved. I am open to learning `DocBook` as this is what `Boost` is using.

3 Project Proposal

Motivation There is 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 are 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 is inefficient in terms of accessing the data by key. Common associate 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 this further.

```cpp
1   #include <iostream>
    #include <initializer_list>
3   #include <map>
    #include <unordered_map>
5   #include <experimental/string_view>
    using std::experimental::string_view;
7
    enum class weekday { sunday, monday, tuesday, wednesday
9                         , thursday, friday, saturday };
11  #define STRING_VIEW(str) string_view{ str, sizeof(str)-1 }
    constexpr std::initializer_list<std::pair< const string_view
13                                      , weekday>> string_to_weekday
        { { STRING_VIEW("sunday"),     weekday::sunday    }
15      , { STRING_VIEW("monday"),     weekday::monday    }
        , { STRING_VIEW("tuesday"),    weekday::tuesday   }
17      , { STRING_VIEW("wednesday"),  weekday::wednesday }
        , { STRING_VIEW("thursday"),   weekday::thursday  }
19      , { STRING_VIEW("friday"),     weekday::friday    }
        , { STRING_VIEW("saturday"),   weekday::saturday  }
21      };
23  int main(void)
    {
25      {
            // Calls malloc() at least 7 times
27          static const std::map<string_view, weekday>
                to_weekday1 = string_to_weekday;
29          std::cout << "'monday' maps to "
                      << static_cast<int>(to_weekday1.at("monday"))
31                    << std::endl;
            std::cout << "'friday' maps to "
33                    << static_cast<int>(to_weekday1.at("friday"))
                      << std::endl;
35          // Calls free() at least 7 times
        }
37      {
            // Calls malloc() at least 8 times
39          static const std::unordered_map<string_view, weekday>
                to_weekday2 = string_to_weekday;
41          std::cout << "'monday' maps to "
                      << static_cast<int>(to_weekday2.at("monday"))
43                    << std::endl;
            std::cout << "'friday' maps to "
45                    << static_cast<int>(to_weekday2.at("friday"))
                      << std::endl;
47          // Calls free() at least 8 times
        }
49      return 0;
    }
```

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). I 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 I 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:

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

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

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

// 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];

// Challenging: again, should only generate code loading from memory.
auto cmap2 = make_static_map(map_data);
auto& at_0 = map_data[0];
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 in *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.[1]

- 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.

---

[1]In theory, one may even make the implementation `C++11` compatible by making heavy use of template recursion. This, however, requires many functions to be implemented twice, and thus will not be pursued in this project.

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

29th May — 2nd June 5th June — 9th June			Consult Boost Developer's mailing list and do some background reading to come up with a suitable design for `static_map` 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 — 16th June 19th June — 23rd June 26th June — 30th June			Implement `static_map` 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 — 7th July 10th July — 14th July 17th July — 21th July 24th July — 28th July			Complete the implementation of `static_map` 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 — 4th August 7th 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 Test

The initial objective was to reimplement the toy `static_map` provided at `https://goo.gl/e07ooa` 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 were `constexpr`, and if they were, 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 was actually a constant expression (as it was 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:

```cpp
constexpr auto factorial(std::size_t const n) -> std::size_t
{
  if (n == 0 || n == 1) return 1;
  return n * factorial(n - 1);
}

constexpr auto foo(std::size_t const n)
{
  return __builtin_constant_p(factorial(n));
}

int main(void)
{
  static_assert(__builtin_constant_p(factorial(10)), "");
  // This fails!
  static_assert(foo(10), "");
}
```

I have 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 have 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()
   {
3      auto x = compute_something();
       if (good_answer(x)) return x;
5      throw std::runtime_error{""};
   }

7
   constexpr auto foo2()
9  {
       auto x = compute_something();
11     return good_answer(x)
           ? x
13         : (throw std::runtime_error{""}, x);
   }
```

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:

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

I have 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&
   {
3      size_t n = 0;
       while (n < N && _values[n].second.first != k) {
5          ++n;
       }
7      return n != N
           ? _values[n].second.second
9          : (throw key_not_found_error{}, _values[0].second.second);
   }
```

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 have learned that the test has outgrown its initial requirements and a complete toy implementation of the proposal is 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 goes 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 is this case. A 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 have decided to follow the "convention" used in `libstdc++` and created an inner storage class that allows the `static_map` to be implemented independent of the underlying storage type. Addition of this inner class turned out to be crutial 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

```cpp
#include <utility>
#include <cassert>
#include <stdexcept>
#include <type_traits>
#include <experimental/string_view>

struct key_not_found_error : std::exception {
  auto what() const noexcept -> char const* override
  { return _msg; }
private:
  static constexpr char const* _msg = "Key not found.";
};

constexpr char const* key_not_found_error::_msg;


namespace {

template < class _Static_map
         , bool _is_const
         >
struct map_iterator {
  using type             = map_iterator<_Static_map, _is_const>;
  using difference_type  = typename _Static_map::difference_type;
  using value_type       =
    std::conditional_t< _is_const
                      , std::add_const_t<typename _Static_map::value_type>
                      , typename _Static_map::value_type >;
  using iterator_category = std::random_access_iterator_tag;
  using reference        =
    std::conditional_t< _is_const
                      , typename _Static_map::const_reference
                      , typename _Static_map::reference >;
  using pointer          = std::add_pointer_t<value_type>;
  using size_type        = typename _Static_map::size_type;

private:
  using _const_iterator  = map_iterator<_Static_map, /*is_const =*/ true>;
  using _storage_type    =
    std::conditional_t< _is_const
                      , std::add_const_t<typename _Static_map::storage_type>
                      , typename _Static_map::storage_type >;

  static constexpr size_type _size = _Static_map::size();
  _storage_type* _data;
  size_type      _i;

  constexpr auto _to_same_object( map_iterator const& x
                                , map_iterator const& y ) const noexcept -> bool
  { return x._data == y._data; }

public:
  constexpr map_iterator() noexcept : _data{ nullptr }, _i{ 0 } {}

  constexpr map_iterator(_storage_type& data, size_type const pos) noexcept
    : _data{ &data }, _i{ pos }
  {}

  constexpr map_iterator(map_iterator const& other) noexcept = default;
  constexpr map_iterator& operator=(map_iterator const& other) noexcept = default;

  __attribute__((always_inline))
  constexpr auto operator*() const -> reference
  {
    return _data != nullptr
      ? ( _i < _size
            ? _data->value(_i)
            : (throw std::out_of_range{"Iterator not dereferenceable."}, _data->value(0))
        )
      : (throw std::invalid_argument{"Iterator not dereferenceable."}, _data->value(0));
  }

  constexpr auto operator->() const -> pointer { return &(*(*this)); }

  constexpr auto operator++() noexcept -> map_iterator& { ++_i; return *this; }

  constexpr auto operator++(int) noexcept -> map_iterator
  {
    map_iterator saved{ *this };
    ++(*this);
    return saved;
  }

  constexpr auto operator-(map_iterator const& other) const -> difference_type
  {
    return _to_same_object(*this, other)
      ? (static_cast<difference_type>(_i) - static_cast<difference_type>(other._i))
      : (throw std::invalid_argument{"Iterators not comparable."}, 0);
  }
```

```
       constexpr auto operator==(map_iterator const& other) const noexcept -> bool
92     { 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
98     { 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
104    { return (*this - other) >  0; }

106    constexpr auto operator>=(map_iterator const& other) const -> bool
       { return (*this - other) >= 0; }
108
       constexpr auto operator[](difference_type const n) const -> reference
110    { return *(*this + n); }

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

114    friend
       constexpr auto swap(map_iterator& lhs, map_iterator& rhs) noexcept -> void
116    {
         auto const _temp_idx = lhs._i;
118      lhs._i = rhs._i;
         rhs._i = _temp_idx;
120
         auto const _temp_data = lhs._data;
122      lhs._data = rhs._data;
         rhs._data = _temp_data;
124    }

126    friend
       constexpr auto operator+ ( map_iterator     const& x
128                              , 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
134    { return x + n; }

136    friend
       constexpr auto operator- ( map_iterator     const& x
138                              , difference_type const  n ) noexcept -> map_iterator
       { return {*(x._data), x._i - n}; }
140
       friend
142    constexpr auto operator+=( map_iterator          & x
                                 , difference_type const  n ) noexcept -> map_iterator&
144    { return x = x + n; }

146    friend
       constexpr auto operator-=( map_iterator          & x
148                              , difference_type const  n ) noexcept -> map_iterator&
       { return x = x - n; }
150 };

152 template <class _Static_map, bool _is_const>
    constexpr typename _Static_map::size_type map_iterator<_Static_map, _is_const>::_size;
154
    } // unnamed namespace
156

158 struct equal_c {
       template <class _Char>
160    constexpr auto operator()(_Char const* const a, _Char const* const b) const
         noexcept( noexcept(std::declval<_Char>() == std::declval<_Char>())
162           && noexcept(std::declval<_Char>() != std::declval<_Char>()) )
       {
164      std::size_t i = 0;
         while (a[i] != _Char{} && a[i] == b[i]) {
166        ++i;
         }
168      return a[i] == _Char{} && b[i] == _Char{};
       }
170 };


172
    namespace {
174
    template <class _Char>
176 inline constexpr auto length_c(_Char const* const str) noexcept -> std::size_t
    {
178    std::size_t i = 0;
       while (str[i] != _Char{}) {
180      ++i;
       }
182    return (i == 0) ? 0 : (i - 1);
    }
184
```

```cpp
      template <class _Char>
186   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
        for (std::size_t j = 0; j < l; ++j) {
196       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()(_T 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()(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>
      struct hash_c<std::experimental::string_view, _N> {
246     constexpr auto operator()(std::experimental::string_view const& x) const noexcept
        { return hash_c<char const*, _N>{}(x.data()); }
248   };


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
      template < class _Key
278           , class _Tp
```

```cpp
            , size_t _N
            , class _Pred   = std::equal_to<_Key>
            , class _Hasher = hash_c<_Key, _N>
            >
    class static_map {

    public:
      using type              = static_map<_Key, _Tp, _N, _Pred, _Hasher>;
      using key_type          = _Key;
      using mapped_type       = _Tp;
      using key_equal         = _Pred;
      using hasher            = _Hasher;
      using value_type        = std::pair<key_type const, mapped_type>;
      using reference         = value_type &;
      using const_reference   = value_type const&;
      using iterator          = map_iterator<type, /*is_const =*/ false>;
      using const_iterator    = map_iterator<type, /*is_const =*/ true>;
      using difference_type   = std::ptrdiff_t;
      using size_type         = std::size_t;

    private:
      struct _Storage {
      private:
        using _actual_value_type = value_type;
        _actual_value_type _vs[_N];

        static constexpr auto make_node(const_reference value)
          noexcept( std::is_nothrow_copy_constructible<key_type>::value
                 && std::is_nothrow_copy_constructible<mapped_type>::value )
        { return std::make_pair(value.first, value.second); }


      public:
        template <class... _Pair>
        constexpr _Storage(_Pair const&... values)
          noexcept(_all<noexcept(make_node(std::declval<_Pair const&>()))...>::value)
          : _vs{ make_node(values)... }
        {}

        constexpr auto value(size_type const i) const noexcept -> const_reference
        { return _vs[i]; }

        constexpr auto value(size_type const i)       noexcept -> reference
        { return _vs[i]; }
      };

    public:
      using storage_type = _Storage;

    private:
      key_equal      const _eq;
      hasher         const _hf;
      storage_type         _data;

    private:
      __attribute__((always_inline))
      constexpr auto _lookup(key_type const& k) const
        noexcept( noexcept(std::declval<hasher>()(std::declval<key_type>()))
               && noexcept(std::declval<key_equal>()( std::declval<key_type>()
                                                    , std::declval<key_type>())) )
      {
        auto const guess = _hf(k);
        auto       i     = guess;

        while (i < _N && !_eq(_data.value(i).first, k)) { ++i; }
        if (i == _N) {
          i = 0;
          while (i < guess && !_eq(_data.value(i).first, k)) { ++i; }
          if (i == guess) return _N;
        }
        return i;
      }

    public:
      template <class... _K, class... _V>
      constexpr static_map(std::pair<_K, _V> const&... values)
        noexcept( std::is_nothrow_constructible< storage_type
                                               , std::pair<_K, _V> const&...>::value
               && std::is_nothrow_default_constructible<key_equal>::value
               && std::is_nothrow_default_constructible<hasher>::value )
        : _eq{}, _hf{}, _data{ values... }
      {}

      template <class... _K, class... _V>
      constexpr static_map( key_equal equal, hasher hash_function
                          , std::pair<_K, _V> const&... values )
        noexcept( std::is_nothrow_constructible< storage_type
                                               , std::pair<_K, _V> const&...>::value
               && std::is_nothrow_move_constructible<key_equal>::value
               && std::is_nothrow_move_constructible<hasher>::value )
        : _eq{ std::move(equal) }
        , _hf{ std::move(hash_function) }
        , _data{ values... }
      {}
```

```cpp
374    static constexpr auto size() noexcept -> size_type
       { return _N; }

       constexpr auto begin() noexcept -> iterator
378    { return {_data, 0}; }

380    constexpr auto   end() noexcept -> iterator
       { return {_data, size()}; }

       constexpr auto cbegin() const noexcept -> const_iterator
384    { return {_data, 0}; }

386    constexpr auto   cend() const noexcept -> const_iterator
       { return {_data, size()}; }

       __attribute__((always_inline))
390    constexpr auto find(key_type const& k) const
         noexcept(noexcept(std::declval<static_map>()._lookup(std::declval<key_type>())))
392      -> const_iterator
       {
394      auto i = _lookup(k);
         return i == _N
396        ? cend()
           : const_iterator{_data, i};
398    }

400    constexpr auto count(key_type const& k) const
         noexcept(noexcept(std::declval<static_map>().find(std::declval<key_type>())))
402      -> size_type
       { return find(k) == cend() ? 0 : 1; }

       __attribute__((always_inline))
406    constexpr auto at(key_type const& k) const -> mapped_type const&
       {
408      auto i = _lookup(k);
         return i != _N
410        ? _data.value(i).second
           : (throw key_not_found_error{}, _data.value(0).second);
412    }

414    __attribute__((always_inline))
       constexpr auto at(key_type const& k) -> mapped_type&
416    {
         auto i = _lookup(k);
418      return i != _N
           ? _data.value(i).second
420        : (throw key_not_found_error{}, _data.value(0).second);
       }

       constexpr auto operator[](key_type const& k) const -> mapped_type const&
424    { return at(k); }

426    constexpr auto operator[](key_type const& k) -> mapped_type &
       { return at(k); }
428  };


432  namespace {

434  template <std::size_t _N>
     __attribute__((always_inline))
436  inline constexpr auto _insert( std::size_t const index, std::size_t const guess
                                   , std::size_t (&indices)[_N] ) noexcept
438  {
       auto i = guess;
440    while (i < _N && indices[i] != _N) { ++i; }
       if (i == _N) {
442      i = 0;
         while (i < guess && indices[i] != _N) { ++i; }
444    }
       indices[i] = index;
446  }

448  template <std::size_t _N>
     __attribute__((always_inline))
450  inline constexpr auto _init_impl( std::size_t (&indices)[_N]
                                     , std::pair<std::size_t, std::size_t> const& x ) noexcept
452  { _insert(x.first, x.second, indices); }

454  template <std::size_t _N, class... _I>
     __attribute__((always_inline))
456  inline constexpr auto _init_impl( std::size_t (&indices)[_N]
                                     , std::pair<std::size_t, std::size_t> const& x
458                                   , std::pair<_I, _I> const&... xs ) noexcept
     {
460    _insert(x.first, x.second, indices);
       _init_impl(indices, xs...);
462  }


464  template <std::size_t, class _T>
```

```cpp
465 using _transform = _T;
466
467 template < class _K
468           , class _V
469           , class _Pred
470           , class _Hasher
471           , std::size_t _N
472           , std::size_t... _Is
473     >
474 __attribute__((always_inline))
475 inline constexpr auto _initialise( std::pair<_K const, _V> const (&il)[_N]
476                                   , _Pred&& equal, _Hasher&& hf
477                                   , std::index_sequence<_Is...> )
478   noexcept( noexcept(std::declval<_Hasher>()(std::declval<_K>())))
479     && noexcept(std::is_nothrow_constructible
480                    < static_map<_K, _V, _N, _Pred, _Hasher>
481                    , _Pred, _Hasher, _transform<_Is, std::pair<_K const, _V> const&>...
482                    >::value) )
483 {
484   std::size_t indices[_N] = {((void)_Is, _N)...};
485   _init_impl(indices, std::make_pair(_Is, hf(il[_Is].first))...);
486
487   return static_map<_K, _V, _N, _Pred, _Hasher>{ std::forward<_Pred>(equal)
488                                                , std::forward<_Hasher>(hf)
489                                                , il[indices[_Is]]... };
490 }
491
492 } // unnamed namespace
493
494 template < class _K
495           , class _V
496           , std::size_t _N
497           , class _Pred = std::equal_to<_K>
498           , class _Hasher = hash_c<_K, _N>
499     >
500 __attribute__((always_inline))
501 inline
502 constexpr
503 auto make_static_map( std::pair<_K const, _V> const (&il)[_N]
504                     , _Pred&& equal = _Pred{}
505                     , _Hasher&& hf = _Hasher{} )
506   noexcept( noexcept(_initialise( std::declval<decltype(il)>()
507                                 , std::declval<_Pred>()
508                                 , std::declval<_Hasher>()
509                   , std::make_index_sequence<_N>{})) )
510 {
511   return _initialise( il
512                     , std::forward<_Pred>(equal)
513                     , std::forward<_Hasher>(hf)
514                     , std::make_index_sequence<_N>{} );
515 }
516
517
518 enum class weekday { sunday
519                    , monday
520                    , tuesday
521                    , wednesday
522                    , thursday
523                    , friday
524                    , saturday };
525
526 using std::experimental::string_view;
527 #define STRING_VIEW(str) std::experimental::string_view{str, sizeof(str)-1}
528
529 struct equal_string_view {
530   constexpr
531   auto operator()( string_view const& a
532                  , string_view const& b ) const noexcept -> bool
533   { return equal_c{}(a.data(), b.data()); }
534 };
535
536 int main(void)
537 {
538   {
539     // Compile-time stuff
540     constexpr std::pair<int const, const char *> map_data[] =
541       { { 5, "apple"  }
542       , { 8, "pear"   }
543       , { 0, "banana" }
544       };
545     // Initialisation
546     constexpr auto cmap = make_static_map(map_data);
547
548     // No abort call in assembly
549     if (!cmap[8]) abort();
550
551     // These operations use hashing
552     static_assert(equal_c{}(cmap[5], "apple"), "");
553     static_assert(equal_c{}(cmap[8], "pear"), "");
554     static_assert(equal_c{}(cmap[0], "banana"), "");
555
556     // This fails to compile -- as expected.
557     // constexpr auto foo = cmap[-1];
558
```

```
560        // We need this because GCC does not support constexpr lambdas, yet.
           struct is_eight {
562          constexpr auto operator()(std::pair<int, char const*> const& x)
               const noexcept
564          { return x.first == 8; }
           };

566
           // Iterators
568        static_assert( find_if_c(cmap.cbegin(), cmap.cend(), is_eight{})
                          != cmap.cend(), "" );
570    }
       {
572        // Run-time stuff
           constexpr std::pair<int const, const char *> map_data[] =
574          { { 5, "apple"  }
           , { 8, "pear"   }
576        , { 0, "banana" }
           };
578        auto cmap = make_static_map(map_data);

580        // No abort in assembly
            if (!cmap.count(5)) abort();

582
           // Values are mutable !!
584        auto& i = cmap.at(8);
           i = "orange";
586        assert(equal_c{}(i, "orange"));

588        // Iterators
           auto const it1 = cmap.find(8);
590        assert(it1->first == 8);
       }
592    {
           // Working with string_view
594        constexpr std::pair<const std::experimental::string_view, weekday>
             string_to_weekday[]
596            { { STRING_VIEW("sunday"),    weekday::sunday }
               , { STRING_VIEW("monday"),    weekday::monday }
598            , { STRING_VIEW("tuesday"),   weekday::tuesday }
               , { STRING_VIEW("wednesday"), weekday::wednesday }
600            , { STRING_VIEW("thursday"),  weekday::thursday }
               , { STRING_VIEW("friday"),    weekday::friday }
602            , { STRING_VIEW("saturday"),  weekday::saturday }
               };
604        constexpr auto to_weekday = make_static_map( string_to_weekday
                                                     , equal_string_view{} );

606
           static_assert(to_weekday[STRING_VIEW("sunday")] == weekday::sunday, "");
608        static_assert(to_weekday[STRING_VIEW("monday")] == weekday::monday, "");
           static_assert(to_weekday[STRING_VIEW("tuesday")] == weekday::tuesday, "");
610        static_assert(to_weekday[STRING_VIEW("wednesday")] == weekday::wednesday, "");
           static_assert(to_weekday[STRING_VIEW("thursday")] == weekday::thursday, "");
612        static_assert(to_weekday[STRING_VIEW("friday")] == weekday::friday, "");
           static_assert(to_weekday[STRING_VIEW("saturday")] == weekday::saturday, "");

614
           if (!to_weekday.count(STRING_VIEW("sunday"))) abort();

616
           struct is_friday {
618          constexpr auto operator()(std::pair< string_view const
                                                , weekday > const& x)
620            const noexcept -> bool
           { return equal_c{}(x.first.data(), "friday"); }
622        };

           // Lookup
624        static_assert(to_weekday.find(STRING_VIEW("friday")) != to_weekday.cend(), "");
626        static_assert(to_weekday.find(STRING_VIEW("__friday__")) == to_weekday.cend(), "" );
           // Linear search
628        static_assert( find_if_c( to_weekday.cbegin(), to_weekday.cend()
                                   , is_friday{} ) != to_weekday.cend(), "" );
630    }
       {
632        // C-strings
           constexpr std::pair<char const* const, weekday>
634          string_to_weekday[]
               { { "sunday",    weekday::sunday }
636            , { "monday",    weekday::monday }
               , { "tuesday",   weekday::tuesday }
638            , { "wednesday", weekday::wednesday }
               , { "thursday",  weekday::thursday }
640            , { "friday",    weekday::friday }
               , { "saturday",  weekday::saturday }
642            };
           constexpr auto to_weekday = make_static_map( string_to_weekday
644                                                    , equal_c{} );
       }
646    return 0;
   }
```

static_map.cpp