Google Summer of Code Proposal: Boost.Static_views

March 2018

Abstract

This document proposes an addition to Boost `C++` Libraries — `Static_views` library. The library focuses on working with compile-time (i.e. `constexpr`) homogeneous data. Its applications range from converting bitmaps from 8-bit to 24-bit representations to implementing efficient enumeration to string conversions and custom error categories.

Personal Details

Name	Tom Westerhout
University	Radboud University of Nijmegen, the Netherlands
Degree Program	Masters in Physics
Email	`kot.tom97@gmail.com`
Availability	I am available from May till the end of August. I consider GSoC a full-time job/internship and thus plan to spend at least 40 hours per week working on the project. Monday, 14th of May (the official start date) seems like a good starting point to me. My summer vacation officially starts only in the beginning of July, before that I have some courses to follow, exams to take and a research project to work on. Although it seems much, last year's experience shows that I still find enough time to work on the project. Also, I will have more free time in July and August and can easily compensate for the lost time, if any. I might also decide to keep working full-time on the project till the end of August rather than the official end of GSoC.

Background Information

I am currently in my first year of Masters in Physics program at Radboud University (in the Netherlands). Although I am a Physics major, I am also passionate about programming. I follow many programming-related courses and am considering doing a Masters in Computer Science. So far, I have followed these project-related courses:

- *Imperative Programming 1 & 2* (Introduction to `C++`).
- *Hacking in C* (Memory layout, calling conventions and debugging).
- *Object Orientation.*
- *Algorithms & Datastructures* (Also worked as a teaching assistant for this course).
- *Languages and Automata.*
- *Functional Programming 1 & 2.*
- *Advanced Programming* (Generic programming, domain specific languages, etc.)
- *Compiler Construction.*

Also, in January 2018 I have participated in 3COWS (Composability, Comprehensibility, Correctness Winter School). It was an intensive programme for Ms. and PhD. students extending the CEFP (Central European Functional Programming) summer school.

For two and a half years I have been working as a teaching assistant at Radboud University. I have taught physics, math, and computer science courses.

As for the projects, as part of my Bachelors internship[6], I have written some `C++14` code[7] for distributed memory systems for doing some physical calculations. We have recently published a paper[9] about the results of these computations. Although the algorithms involved are not particularly difficult, as far as I know, this is still the only open source solution for calculating plasmonic properties of non-translational invariant quantum systems.

Currently, I am working on applying swarm intelligence to obtaining excited states of many-body quantum systems[8]. High level algorithms are written in `Haskell` and performance critical parts — in `C++17` (with a `C` wrapper in between). We use a shallow neural network (a Restricted Boltzmann Machine) as a variational ansatz by tracing out all hidden nodes. This part is written in `C++`. Monte-Carlo simulations which are used to estimate the fitness function are written in `C++` as well. We hope to present some first results at "Beyond Digital Computing" conference in Heidelberg near end of March 2018.

Finally, last summer I have successfully completed a Google Summer of Code working on the very same project I am proposing here. I believe `Static_views` library to be very useful. It is however not possible to create a Boost-quality library in just one summer. Having a relatively high study load (due to my following both Physics and CS courses) I don't have enough time to work on the project from September till May. I did however manage to follow all the major discussions on `boost-dev` mailing list. Also, I feel like I may need some guidance to complete this project. These two reasons are my main motivation for doing GSoC this year.

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

`C++ 98/03`	≈ 3;
`C++ 11/14/17`	≈ 4;
`C++ Standard Library`	≈ 4;
`Boost C++ Libraries`	≈ 4;
`Git`	≈ 3.

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 `Neovim` and use some build system to compile it. From the `C++`–related ones I am familiar with `Boost.Build` and `CMake`. For documentation, I have grown to like `Sphinx` (which `Static_views` currently uses). I also have experience in using `Doxygen` and have played around with `Standardese`.

Project Proposal

Ranges TS, Boost.Hana and Static_views Computations in C++ can be divided into *run-time* and *compile-time* ones.[1] If we talk about run-time manipulation of homogeneous sequences, then quite often STL algorithms[1] are enough. If one wants to do multiple things in one pass, Ranges TS[5] come into play. For run-time manipulation of heterogeneous sequences there is Boost.Fusion[2]. There is also Boost.Hana[3] which greatly simplifies working with compile-time heterogeneous sequences. But what about homogeneous compile-time data? One could naively expect that Boost.Hana can handle that as well. It turns out to be not the case.

Hana focuses on type-level information. This means that if one wants to, for example, find all floats in a constexpr C-style array which are smaller than some threshold value, one can't do that in Hana in a non-hacky way. This example illustrates that in C++ there is a gap between run-time computations (Ranges TS) and type-level computations (Boost.Hana). Static_views is meant to fill this gap. The library consists of two layers: low-level views and high-level data structures implemented on top of them.

Static Map static_views::static_map is an example (currently, the only one) of such high-level data structures. Simply put, it is a compile-time analogue of std::unordered_map. Consider the following toy example:

```
enum class weekday {
  sunday, monday, tuesday, wednesday, thursday, friday, saturday};

#define STRING_VIEW(str) string_view{str, sizeof(str)-1}
constexpr static std::pair<string_view const, weekday>
  string_to_weekday[] =
    {{ STRING_VIEW("sunday"),    weekday::sunday    },
     { STRING_VIEW("monday"),    weekday::monday    },
     { STRING_VIEW("tuesday"),   weekday::tuesday   },
     { STRING_VIEW("wednesday"), weekday::wednesday },
     { STRING_VIEW("thursday"),  weekday::thursday  },
     { STRING_VIEW("friday"),    weekday::friday    },
     { STRING_VIEW("saturday"),  weekday::saturday  }};

int main()
{
  {
    // Calls malloc() at least 7 times
    static const std::map<string_view, weekday> to_weekday1{
      std::begin(string_to_weekday), std::end(string_to_weekday)};
    std::cout << "'monday' maps to "
              << static_cast<int>(to_weekday1.at("monday"))
              << std::endl;
    std::cout << "'friday' maps to "
              << static_cast<int>(to_weekday1.at("friday"))
              << std::endl;
    // Calls free() at least 7 times
  }
  {
    // Calls malloc() at least 8 times
    static const std::unordered_map<string_view, weekday> to_weekday2{
      std::begin(string_to_weekday), std::end(string_to_weekday)};
    std::cout << "'monday' maps to "
              << static_cast<int>(to_weekday2.at("monday"))
              << std::endl;
    std::cout << "'friday' maps to "
              << static_cast<int>(to_weekday2.at("friday"))
              << std::endl;
    // Calls free() at least 8 times
  }
  return 0;
}
```

We have some data available to us at compile-time and we would like to perform lookups at both compile-time and run-time. Using std::unordered_map or std::map we get $\mathcal{O}(1)$ or $\mathcal{O}(\log N)$ lookups respectively, but at the cost of $\mathcal{O}(N)$ memory allocations. On top of that, even when

[1] Without loss of generality, we can say that all computations produce results, because if a computation results in nothing (void), we can always create a wrapper around it returning a dummy value. Then we can easily distinguish run- and compile-time computations by defining a computation as compile-time if its result can be stored in a constexpr variable.

keys we are looking for are known at compile-time, the lookups themselves only happen at run-time. Consider now the very same example written in terms of `static_views::static_map`:

```
static auto to_weekday = static_views::make_static_map(string_to_weekday);
std::cout << "'monday' maps to "
          << static_cast<int>(to_weekday1.at("monday"))
          << std::endl;
std::cout << "'friday' maps to "
          << static_cast<int>(to_weekday1.at("friday"))
          << std::endl;
```

Firstly, we get rid of memory allocations. Secondly, recent compilers are able to perform the lookups at compile-time. And finally, seeing as we have declared `string_to_weekday` as `constexpr`, we can make `to_weekday constexpr` as well which will completely eliminate the initialisation costs.

Better switch One can also use `static_map` as a `switch` statement:

```
enum class animal : unsigned {
  cat    = 0x11,
  dog    = 0x22,
  turtle = 0x33,
  ... a couple more ...
};

// Implementation using switch
auto operator<<(std::ostream& out, animal const x) -> std::ostream&
{
  switch (x) {
  case animal::cat:    return out << "cat";
  case animal::dog:    return out << "dog";
  case animal::turtle: return out << "turtle";
  default: throw std::invalid_argument{"What has just happened?"};
  };
}

// Implementation using static_map
auto operator<<(std::ostream& out, animal const x) -> std::ostream&
{
  constexpr static std::pair<animal, char const*> mapping[] = {
    {animal::cat,    "cat"},
    {animal::dog,    "dog"},
    {animal::turtle, "turtle"}
  };
  constexpr auto animal_to_string = static_views::make_static_map(mapping);
  return out << animal_to_string.at(x);
}
```

This very simple example also illustrates some advantages of `static_map` over a simple `switch`. Notice that `cat` and `dog` do not map to consecutive integers. This means that algorithmically, the time complexity of first implementation is linear in the number of animals while that of the second is (amortised) constant. This becomes important if we, for example, want to implement a custom `error_category`. Quite often, error codes are not just consecutive integers and if there are many of them (e.g. there are thousands of possible `NTSTATUS` values) using an $\mathcal{O}(1)$ implementation over an $\mathcal{O}(N)$ one starts to make sense.

What needs to be done The goal of this project is to bring `Static_views` to "review-ready" state, i.e. a state in which Boost developers find the library fit to be put into library review queue. It is very difficult to specify exactly what needs to be done, but here are the most important points:

- Ensure that the above examples compile and run on at least two major compilers (`Clang` and `MSVC`) and are thoroughly worked out in the tutorial section of the documentation.
- Complete the reference part of the documentation.
- Polish the implementation. In particular, the current messy way of specifying concepts should be improved.
- Add automatic code coverage checks and make sure the results are above 90%.

- Prove, by means of extensive benchmarking, that `static_views::static_map` outperforms `std::unordered_map` in both the speed of initialisation and the speed of lookups.

Also, here are a few things which need to be done but are optional rather than essential part of the project:

- Fix `GCC` so that it can compile `Static_views` code.
- Add `CMake` support.
- Add more data structures such as a flat sorted map.

Schedule

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

14th May	—	18th May	Add automatic code coverage tests and start polishing the implementation.
21st May 28th May	— —	25th May 1st June	Keep working on the implementation and make sure both `Clang` and `MSVC` are happy with it.
4th June	—	8th June	Keep working on the implementation and tests.
11th June	—	15th June	First evaluation. It would be nice to have enough unit tests and an implementation we are happy with.
18th June 25th June 2nd July	— — —	22nd June 29th June 6th July	Work out the "`enum`→string", "8-bit↔24-bit bitmaps", and "`std::error_category`" examples. Create a PR to Niall's `ntkernel-error-category`[4] library to make it work correctly with `Static_views`.
9th July	—	13th July	Start working on benchmarks and (possibly) optimising the library based on them.
16th July	—	20th July	Finish the benchmarks and ask for some initial feedback on `boost-dev` mailing list.
23th July	—	27th July	Have a go at fixing `GCC`.
30th July 6th August 13th August	— — —	3rd August 10th August 17th August	Work on other optional tasks and changes based on feedback from `boost-dev`.

References

[1] STL algorithms. http://en.cppreference.com/w/cpp/algorithm.

[2] Joel de Guzman, Dan Marsden, and Tobias Schwinger. Boost.Fusion. https://github.com/boostorg/fusion.

[3] Louis Dionne. Boost.Hana. https://github.com/boostorg/hana.

[4] Niall Douglas. `ntkernel-error-category`. https://github.com/ned14/ntkernel-error-category.

[5] Eric Niebler and Casey Carter. Working draft, C++ extensions for ranges. http://open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4671.pdf, June 2017.

[6] T. Westerhout. Plasmons in fractals. https://twesterhout.github.io/plasmon-cpp, 2017.

[7] T. Westerhout. Tools to calculate quantities related to plasmons in materials with no translational symmetry. https://github.com/twesterhout/plasmon-cpp, 2017.

[8] T. Westerhout. Code snippets for my maybe masters thesis. https://github.com/twesterhout/tcm-swarm, 2018.

[9] Tom Westerhout, Edo van Veen, Mikhail I Katsnelson, and Shengjun Yuan. Plasmon confinement in fractal quantum systems. *arXiv preprint arXiv:1801.06439*, 2018. https://arxiv.org/abs/1801.06439.