

```
constexpr in Practice
FINDINGS FROM OUR CONFERENCE TALK “constexpr ALL THE THINGS!”
```

**Contents**

1 **Introduction**  

2 **Representing JSON values**  
   2.1 A compile-time `vector` type  
   2.2 Compile-time associative containers  
   2.3 Algorithms  
   2.4 Iterators  

3 **Parsing JSON values**  
   3.1 Multiple passes over the literal  
   3.2 `std::string_view`  

4 **Miscellany**  
   4.1 Oversights in the standard?  
   4.1.1 `std::swap` and `std::exchange`  
   4.1.2 `std::array`  
   4.2 Debugging  
   4.3 Other work  

5 **Conclusion**  

6 **References**

1 **Introduction**

This paper surveys part of the `constexpr` landscape as of late 2017. In January 2017 we began work on a presentation for [C++Now], subsequently also presented at [CppCon]. The thesis for this talk was an exploration of `constexpr` techniques through the construction of a compile-time user-defined literal for JSON values, as exemplified by the following hypothetical code:

```cpp
```
```cpp
constexpr auto jsv = R"({
  "feature-x-enabled": true,
  "value-of-y": 1729,
  "z-options": {
    "a": null,
    "b": "220 and 284",
    "c": [6, 28, 496]
  }
})"_json;

if constexpr (jsv["feature-x-enabled"]) {
    // code for feature x
} else {
    // code when feature x turned off
}
```

Although we achieved our goals, along the way we found some things we consider to be omissions and/or defects in the current standard (both in the library and in the language), as well as implementation bugs in standard libraries and varying behaviour between compilers. It is our hope that by highlighting current shortcomings and giving insight into various use cases for constexpr, this paper will provide guidance for future constexpr standardization and helpful considerations for implementers.

2 Representing JSON values

A JSON value is a discriminated union of six possibilities: null, a boolean, a number, a string, an array of values, or an object (which is a map of strings to values). Of these, nulls, booleans and numbers are immediately usable as literal types. Strings, arrays and objects require some implementation as compile-time containers.

2.1 A compile-time vector type

We implemented a simple bounded compile-time vector type with a size member of type `std::size_t` and a data member of type `std::array<T, N>`.

```cpp
namespace cx {

  template <typename Value, std::size_t Size = 5>
  class vector {
    using storage_t = std::array<Value, Size>;

  public:
    template<typename Itr>
    constexpr vector(Itr begin, const Itr &end) {
      while (begin != end) {
        push_back(*begin);
        ++begin;
      }
    }

  }

```
constexpr vector(std::initializer_list<Value> init) 
  : vector(init.begin(), init.end())
{
}

constexpr vector() = default;

// some functions omitted for the sake of brevity
// and initial exposition
// ...

constexpr auto capacity() const { return Size; }
constexpr auto size() const { return m_size; }
constexpr auto empty() const { return m_size == 0; }

constexpr void clear() { m_size = 0; }

constexpr const Value* data() const {
  return m_data.data();
}

private:
  storage_t m_data{};
  std::size_t m_size{0};
};

Our first surprise came when we tried to implement begin() and end() for this vector type in the obvious way:

template <typename Value, std::size_t Size = 5>
class vector
{
  // ...

  constexpr auto begin() const { return m_data.begin(); }
  constexpr auto begin() { return m_data.begin(); }

  constexpr auto end() const { return std::next(m_data.begin(), m_size); }
  constexpr auto end() { return std::next(m_data.begin(), m_size); }

  // ...
};

The implementation of std::next in GCC 7.2 fails when used in a constexpr context because of an internal function not being marked constexpr, as can be seen at https://godbolt.org/g/4YBTN7. The functions std::advance and std::distance fail in the same way.

This issue appears fixed in a trunk build of GCC as of September 2017. We don’t take the issue as a failing of library authors, but rather as an indication that constexpr functions require extensive testing.

If one is writing a constexpr function, adequate testing practically requires a doubling-up of tests to ensure that everything works both when invoked at runtime and at compile time.
2.2 Compile-time associative containers

We used the constexpr vector type for JSON strings (cx::vector<char>) and for JSON arrays (cx::vector<JSON_Value>). JSON objects are maps from strings to values, so we used an array of pairs as the underlying storage for our map type.

```cpp
template <typename Key, typename Value, std::size_t Size = 5,
          typename Compare = std::equal_to<Key>>
class map {
public:
    // ...
private:
    std::array<cx::pair<Key, Value>, Size> m_data{};
    std::size_t m_size{};
};
```

Note the use of cx::pair rather than std::pair. We were unable to use std::pair because

**std::pair’s assignment operator is not constexpr.**

This appears to be a defect in the standard, since if one has constexpr-friendly types inside a std::pair, one can clearly assign first and second in a constexpr context. The same is true of std::tuple.

2.3 Algorithms

In building a constexpr map type as an array of pairs, providing operator[] required a constexpr implementation of std::find_if. This we provided by simply adding the keyword constexpr to a reference implementation found at http://en.cppreference.com/w/cpp/algorithm/find.

```cpp
template <class InputIt, class UnaryPredicate>
constexpr InputIt find_if(InputIt first, InputIt last, UnaryPredicate p) {
    for (; first != last; ++first) {
        if (p(*first)) {
            return first;
        }
    }
    return last;
}
```

In the course of implementing our talk, we found constexpr uses for several other algorithms, e.g. std::equal and std::mismatch, which we provided similarly.

It is our belief that all of the C++14-era (pre-parallel) algorithms may be made constexpr, in most cases just by adding constexpr.

Requirements for the three algorithms that may allocate memory – std::inplace_merge, std::stable_partition, and std::stable_sort – are written such that complexity guarantees are
relaxed if there is not enough memory available, so we believe these could also be made constexpr with relaxed complexity.

2.4 Iterators

Using constexpr algorithms naturally requires constexpr iterators. We have a constexpr vector type with a constexpr push_back operation.

No functions on std::back_insert_iterator are yet constexpr-friendly.

Having a constexpr-capable back_insert_iterator was useful to us, so we wrote an equivalent. We see no reason why most iterators and their member functions could not be marked constexpr, so that they may work with an appropriate constexpr-friendly container.

3 Parsing JSON values

Our basic approach to parsing the JSON user-defined literal at compile time was to build a parser combinator library using constexpr lambda expressions and, from this, to build a JSON parser. The details of this approach can be found in our conference talks and in the code at [constexpr_all_the_things].

3.1 Multiple passes over the literal

In order to precompute size information, we needed to make more than one pass over the literal to be parsed. To this end, we used the templated form of a string literal operator proposed in [N3599] and furthered in [P0424]. This is currently implemented as an extension in GCC and Clang in the form originally proposed.

In order to achieve right-sizing of the JSON value, the return value of this operator is dependent on the output size required, which is computed from the literal itself.

In order to achieve right-sizing of the JSON value, the return value of this operator is dependent on the output size required, which is computed from the literal itself.

Multiple passes on the input, with later passes dependent on earlier ones, would have been impossible without a templated user-defined literal operator.
On line 4 above, we use a `std::initializer_list` to expand the template arguments. It doesn’t need to be `constexpr` in this case, but naturally, being in a “`constexpr ALL the things!” mindset, by default we typed `constexpr` in front of it, only to find that

```
constexpr std::initializer_list is not currently possible.
```

Although we are not sure of the possibilities or requirements here, this was at least surprising, and the source of an error in our slides and an issue logged against our github repository.

On line 5 above, the possibility of using structured bindings to decompose `S.num_objects` and `S.string_size` occurred to us. Once again, we were surprised to find that

```
structured bindings cannot currently be applied to `constexpr` declarations.
```

### 3.2 `std::string_view`

In parsing JSON literals, we made extensive use of `std::string_view`. However, we found a few operations where the particular library implementation available at the time (GCC 7.2) was not `constexpr`-capable, although the standard does specify `constexpr`:

- constructing a `std::string_view` from a `const char*`
- `std::string_view`'s `operator==`
- member functions `remove_prefix` and `remove_suffix`

Like the issue with using `std::next`, these issues have been fixed in GCC trunk, but they serve to underline the requirement for `constexpr` tests.

### 4 Miscellany

#### 4.1 Oversights in the standard?

In the course of development, we found a few other things that seem to have missed the `constexpr` train for no good reason.

##### 4.1.1 `std::swap` and `std::exchange`

Neither of these functions is currently marked `constexpr`, and by our thinking, they would be useful at compile-time. If `std::swap` were `constexpr`, we would naturally expect that `swap` implemented as a member function on library types would also be `constexpr` where that makes sense (e.g. `std::array::swap`).
4.1.2 std::array

The “trivial” container operations on std::array are all constexpr. However, swap and fill are not, and neither are the equality and comparison operators. To us, this seems like an omission.

The standard seems to have several inconsistencies in applying constexpr; in particular, there seem to be oversights for mutating operations and container operations which may be constexpr-friendly on literal types.

4.2 Debugging

Writing the code was not easy. In our experience, building up anything more than trivial constexpr constructs requires painstaking piecemeal work, frequently using primitive tools like throw to force errors in constexpr evaluation.

In the case of building up parser combinators, the nesting depth of lambda expressions was such that compiler diagnostics were extremely large and obscure, even for experienced template metaprogrammers.

Additionally, when the C++ was working, but the JSON string to be parsed was ill-formed, there was no easy way to communicate the location and nature of the parse error. This prevented us from writing anything more than very primitive error handling and from setting up credible tests for intentionally failing parses.

The ability to provide custom compiler diagnostics would have helped tremendously in the development, use, and testing of our code.

[P0596] is of particular interest to us as a potential facility for compile-time debugging and testing.

Although the focus of our presentation was entirely on compile-time computation, the mixing of runtime and compile-time calculations is currently problematic, given the inability to control or detect when the evaluation is occurring. This did lead to a bug in our slides; it’s easy to accidentally omit a constexpr keyword and get runtime results where purely compile-time was intended. A facility like [P0595] would be very helpful here.

4.3 Other work

Much of our work was in discovering currently usable techniques for creating and manipulating structures at compile time. At the moment we are limited to fixed-size structures, and this greatly increases the complexity of solutions. If dynamic structures were available at compile time, this work would be a lot simpler.

For this reason, we are very interested in the evolution of ideas in [P0597] and [P0639].
5 Conclusion

Even with the inability to create dynamic structures at compile time, creating relatively complex structures is currently possible.

The parser combinator approach offers, we think, an interesting technique for embedding all kinds of literals in C++ code. It is relatively easy to construct new parsers with this technique. JSON values are just one example; other possible applications are compile-time regular expressions, URL decompositions, and more.

Compile-time computation in relatively “normal” constexpr C++, rather than template metaprogramming, is likely to become more widespread, and the development of the standard to better support use cases like ours is exciting.

6 References

References

[C++Now] Ben Deane and Jason Turner. constexpr ALL the things! presented at C++Now 2017 https://www.youtube.com/watch?v=HMB9oXFobJc

[CppCon] Ben Deane and Jason Turner, constexpr ALL the things! presented at CppCon 2017 https://www.youtube.com/watch?v=PJwd4JLYJJY

[constexpr_all_the_things] Ben Deane and Jason Turner, constexpr_all_the_things git repository https://github.com/lefticus/constexpr_all_the_things


[P0596] Daveed Vandevoorde, std::constexpr_trace and std::constexpr_assert http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0596r0.html


[P0597] Daveed Vandevoorde, std::constexpr_vector<T> http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0597r0.html