BEN DEANE / bdeane@blizzard.com / @ben_deane

JASON TURNER / jason@emptycrate.com / @lefticus

C++NOW / TUESDAY 16TH MAY 2017

DON'T BURY THE LEAD

We want to do this:

```
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
}
```

THE PROMISE OF `constexpr`

- Runtime efficiency
- Clearer code, fewer magic numbers
- Less cross-platform pain

`constexpr` HISTORY 101

A Short, Incomplete (and Mostly Wrong?) History of `constexpr`

`constexpr`: THE FIRST AGE

- One (`return`) expression per function was allowed
- `constexpr` math functions explored
- `throw` link error trick discovered
- Recursive `constexpr` FNV1 string hash discovered

constexpr: END OF THE FIRST AGE

```cpp
// 1. Fall of Gondolin
// 2. Balrogs destroyed
// 3. Morgoth defeated and cast into the Timeless Void
// 4. constexpr string hashing discovered

constexpr uint64_t fnv1(uint64_t h, const char* s)
{
  return (*s == 0) ? h :
    fnv1((h * 1099511628211ull) ^ static_cast<uint64_t>(*s), s+1);
}
```

`constexpr`: THE SECOND AGE

- Generalized `constexpr` supported by Visual C++
- Compile-time computation/optimization popularized by *C++ Weekly* et al.
- Generalized `constexpr` string hashing (e.g. Murmur3) discovered
- `constexpr` libraries start to appear

`constexpr`: END OF THE SECOND AGE

Murmur hash as constexpr (C++14) (szelei.me)
53 submitted 1 year ago by sztomi rpclib
comment share save hide give gold report

(Also: Last Alliance of Elves & Men, Isildur takes up the hilt of Narsil and cuts the One Ring from Sauron's hand.)

`constexpr`: THE THIRD AGE?

- `constexpr` lambdas
- `if constexpr`
- `constexpr` STL proliferation?
- (coming soon, no doubt) `constexpr` cryptographic hashes

A PROBLEM WITH `constexpr`

[–] **SeanMiddleditch** Game Developer 6 points 3 days ago

A big problem for compile-time string types is the inability to use a different algorithm at run-time than compile-time. The implementation of `strlen` you want at run-time isn't legal in `constexpr` evaluation, for instance, and the `constexpr` version is quite a bit slower than the smart run-time version even with today's best compiler optimizations applied.

permalink embed save parent report give gold reply

The inability to use different runtime and compile time algorithms.

`constexpr` SFINAE?

A partial solution, perhaps.

```
constexpr auto str_view = "Computers are useless."sv;
constexpr auto cx_hash = fnv1(str_view);

auto str = "They can only give you answers."s;
auto rt_hash = fnv1(str);
```

We would *like* the second call to `fnv1` to use an efficient runtime algorithm.

`constexpr` SFINAE?

You *can* do something like this…

```
template <typename T>
using constexpr_construct_t = std::integral_constant<bool, (T{}, true)>;
```

And then feed it to the detection idiom.

`constexpr` SFINAE?

```
template <typename StringType>
constexpr auto fnv1(const StringType& s)
{
  if constexpr(is_detected_v<constexpr_construct_t, StringType>) {
    return cx::fnv1(s);
  } else {
    return runtime::fnv1(s);
  }
}
```

Caveats: many.

- "`constexpr` constructible" doesn't mean "`constexpr` hashable"
- "`constexpr` capable" doesn't mean "in a `constexpr` context"
- verbosity, compile time?

`constexpr` SFINAE?

Another possible (better) solution.

P0595

To: EWG

From: Daveed Vandevoorde (`daveed@edg.com`)

Date: 2017-02-02

The `constexpr` Operator

BUILDING constexpr JSON VALUES

```
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;
```

Two problems to solve:

1. How to represent JSON values
2. How to parse JSON values

REPRESENTING JSON VALUES

A JSON Value is a discriminated union of:

- null
- boolean
- number
- string
- array of values
- object (map of string -> value)

Clearly this is going to need some sort of recursion. And we are going to need `constexpr` ways to deal with a string, a vector and a map.

`constexpr` STRINGS

First, `std::string_view` is great, and *mostly* `constexpr`
(depending on how up-to-date your library implementation is).

Of course, `std::string_view` only really handles literal values:
it doesn't deal with building strings, and is not intended for *storing* strings.

`constexpr` STRINGS

We need a way to pass, store, and in general, work with character string literals.

While `std::string_view` would technically work for this it kind of mixes metaphors, since it is not intended for storing and comparison - just viewing.

For this, we built up the `static_string` class.

constexpr STRINGS

```cpp
struct static_string
{
  template <std::size_t N>
  constexpr static_string(const char (&str)[N])
    : m_size(N-1), m_data(&str[0])
  {}

  // constructor for substrings of string literals
  constexpr static_string(const char* str, std::size_t s)
    : m_size(s), m_data(str)
  {}

  constexpr static_string() = default;

  constexpr size_t size() const { return m_size; }
  constexpr const char *c_str() const { return m_data; }

  std::size_t m_size{0};
  const char *m_data = nullptr;
};
```

constexpr VECTORS

```
template <typename Value, std::size_t Size = 5>
class vector
{
  using storage_t = std::array<Value, Size>;
  storage_t m_data{};
  std::size_t m_size{0};
  ...

  // iterators, push_back, operator[] etc
  // are pretty easy to write
}
```

constexpr VECTORS

```
using iterator = typename storage_t::iterator;
using const_iterator = typename storage_t::const_iterator;

constexpr auto begin() const { return m_data.begin(); }
constexpr auto end() const { return m_data.begin() + m_size; }
// and similarly for other iterator functions...

constexpr void push_back(Value t_v)
{
  if (m_size >= Size) {
    throw std::range_error("Index past end of vector");
  } else {
    m_data[m_size++] = std::move(t_v);
  }
}
```

We were not able to use `std::next()` here, seems to be a bug in the implementation…

WHY NOT `std::next`?

In GCC's implementation: internal `__iterator_category` is not `constexpr` constructible.

```cpp
#include <array>
#include <iterator>

using namespace std;

constexpr array<int, 5> foo = {1,2,3,4,5};

constexpr auto third_of_foo()
{
  return next(foo.cbegin(), 3);
}

int main()
{
  constexpr auto i = *third_of_foo();
}

```

x86-64 gcc (snapshot) -std=c++1z -O3 -Wall -Wextra

11010 .LX0: .text // Intel

```
<Compilation failed>
```

```
In file included from /opt/compiler-explorer/gcc-trunk-20170
                 from /opt/compiler-explorer/gcc-trunk-20170
                 from /opt/compiler-explorer/gcc-trunk-20170
                 from /opt/compiler-explorer/gcc-trunk-20170
                 from /opt/compiler-explorer/gcc-trunk-20170
                 from <source>:1:
/opt/compiler-explorer/gcc-trunk-20170521/include/c++/8.0.0/
<source>:15:35:   in constexpr expansion of 'third_of_foo()'
<source>:10:14:   in constexpr expansion of 'std::next<const
/opt/compiler-explorer/gcc-trunk-20170521/include/c++/8.0.0/
       std::advance(__x, __n);
       ~~~~~~~~~~~~^~~~~~~~~~
/opt/compiler-explorer/gcc-trunk-20170521/include/c++/8.0.0/
     advance(_InputIterator& __i, _Distance __n)
     ^~~~~~~
```

Edit on C++ Compiler Explorer

`constexpr` VECTORS

This allows for natural use of the `vector` type

```
vector<int> vec;
vec.push_back(15);
```

constexpr VECTORS

Or put into a constexpr context

```
constexpr auto get_vector() {
  vector<int> vec;
  vec.push_back(15);
  return vec;
}

int main() {
  constexpr auto a_vector = get_vector();
  static_assert(a_vector.size() == 1);
}
```

MUTABLE `constexpr` STRINGS

And now we can build a mutable `constexpr` string by inheriting from our `vector`

MUTABLE constexpr STRINGS

```
template <typename CharType, size_t Size>
struct basic_string : vector<CharType, Size>
{
  constexpr basic_string(const static_string &s)
    : vector<CharType, Size>(s.begin(), s.end())
  {}
  constexpr basic_string(const std::string_view &s)
    : vector<CharType, Size>(s.cbegin(), s.cend())
  {}
  // ...
};
```

This relies on:

- constexpr data members must be initialized, so our base vector is all `0`
- We have not provided any methods for shrinking our data structures, but that is possible

constexpr MAPS

```cpp
template <typename Key, typename Value, std::size_t Size = 5>
class map
{
  using storage_t = std::array<cx::pair<Key, Value>, Size>;
  storage_t m_data{};
  std::size_t m_size{0};
  ...

  // iterators are the same as for arrays
  // operator[] needs a constexpr find
  // data grows in the same way that vector does
}
```

constexpr **MAPS**

```
constexpr auto get_colors() {
  cx::map<cx::static_string, std::uint32_t> colors;
  colors["red"] = 0xFF0000;
  colors["green"] = 0x00FF00;
  return colors;
}

int main() {
  constexpr auto colors = get_colors();
  colors["red"]; // returns 0xFF0000
  colors["blue"]; // compile-time error
}
```

WHY NOT `std::pair`?

Standard library definition does not have `constexpr operator=` for `std::pair`

This is the only aspect of `std::pair` that is not `constexpr`

```cpp
#include <utility>

using namespace std;

struct P
{
  std::pair<const char*, int> pr;
};

constexpr auto pair_test()
{
  P p;
  p.pr = make_pair("taxicab", 1729);
  return p;
}

int main()
{
  constexpr auto p = pair_test();
}
```

x86-64 gcc (snapshot) -std=c++1z -O3 -Wall -Wextra

11010 .LX0: .text // Intel

```
1 <Compilation failed>
```

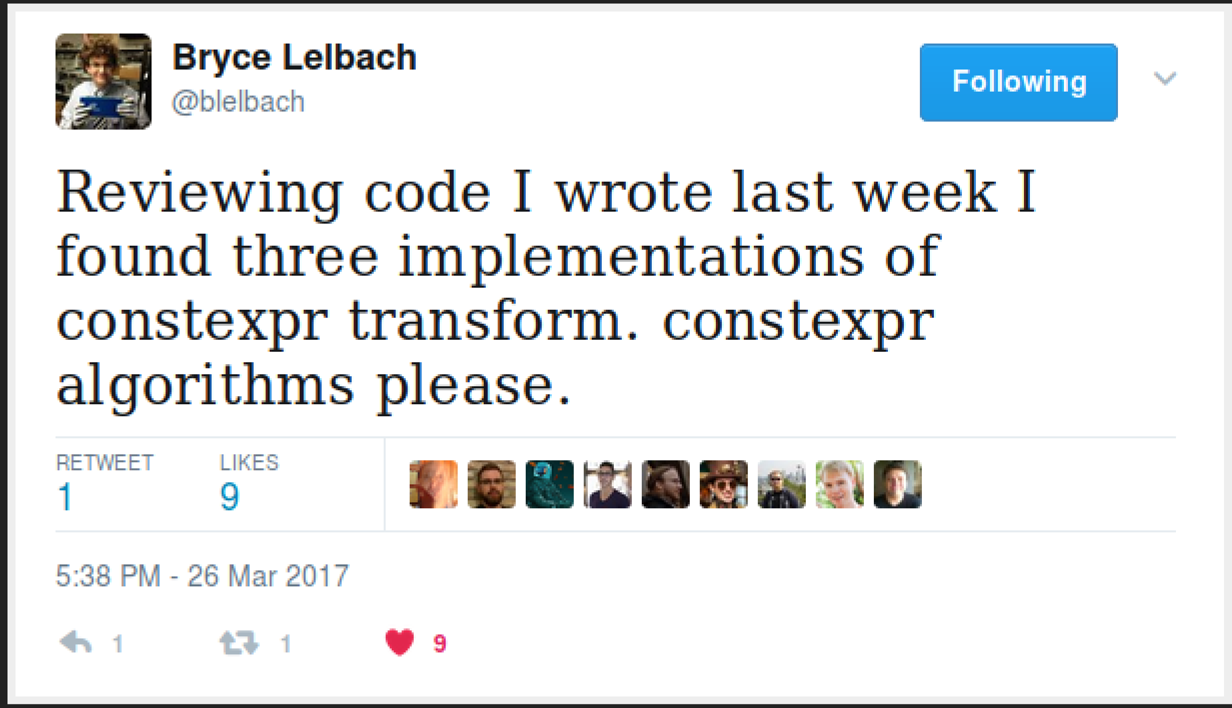
```
<source>: In function 'constexpr auto pair_test()':
<source>:13:35: error: call to non-constexpr function 'std::pair<_T1, _T2>
   p.pr = make_pair("taxicab", 1729);
                                   ^
<source>: In function 'int main()':
<source>:19:32: error: 'constexpr auto pair_test()' called in a constant e
   constexpr auto p = pair_test();
                                ^
Compiler exited with result code 1
```

Edit on C++ Compiler Explorer

constexpr find_if

```
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;
}
```

LET'S MAKE THEM ALL `constexpr` ALREADY



OTHER ALGORITHMS WE MADE `constexpr`

- `mismatch`
- `equal`
- `copy`

In the course of implementing this talk, we found uses for several `constexpr` algorithms.

JSON VALUE: FIRST ATTEMPT

```
template <size_t Depth=5>
struct JSON_Value
{
  static constexpr size_t max_vector_size{6};
  static constexpr size_t max_map_size{6};

  struct Data
  {
    bool boolean{false};
    double number{0};
    cx::static_string string;
    cx::vector<JSON_Value<Depth-1>, max_vector_size> array;
    cx::map<cx::static_string, JSON_Value<Depth-1>, max_map_size> object;
  };
  enum struct Type { Null, Boolean, Number, String, Array, Object };

  Type type = Type::Null;
  Data data;
  ...
};

template <> struct JSON_Value<0> {};
```

JSON VALUE: FIRST ATTEMPT

```
struct JSON_Value
{
  constexpr void assert_type(Type t) const
  {
    if (type != t) throw std::runtime_error("Incorrect type");
  }

  // For Array, and similarly for the other types
  constexpr decltype(auto) to_Array() const
  {
    assert_type(Type::Array);
    return (data.array);
  }
  constexpr decltype(auto) to_Array()
  {
    if (type != Type::Array) {
      type = Type::Array;
      data.array = {};
    }
    return (data.array);
  }
};
```

JSON VALUE: FIRST ATTEMPT

```
cx::JSON_Value j{};
j["a"].to_Number() = 15;
j["b"].to_String() = "Hello World";
j["d"].to_Array();
j["c"]["a"]["b"].to_Array().push_back(10.0);
j["c"]["a"]["c"] = cx::static_string("Hello World");
j["c"]["a"]["d"].to_Array().push_back(5.2);
```

WHY NOT `std::variant` ?

Similarly to `std::pair`, `std::variant` is missing some key `constexpr` support.

- `std::variant(const std::variant &)`
- `std::variant(std::variant &&)`
- `std::variant &operator=(const std::variant &)`
- `std::variant &operator=(std::variant &&)`

REQUIREMENTS FOR COMPILE-TIME TYPES

Huge list! Are you ready?!

- `constexpr` constructor
- `std::is_trivially_destructible`

Nothing else is required if it does not get invoked.

STL SHORTCOMINGS

- `array`
- `string`
- `string_view`
- `pair`
- `optional`
- `variant`
- `swap`

LIMITATIONS OF OUR CONTAINERS

- Fixed maximum size
- (Currently) cannot shrink
- Requires types that are default constructible

HOW TO IMPROVE OUR CONTAINERS

- We could wrap objects in `std::optional` to allow for objects that are not default constructible
- It should be possible to templatize on `constexpr` enabled allocator, making these containers optionally `constexpr`

`constexpr` ALLOCATOR?

From cppreference.com

```
template <class T>
struct SimpleAllocator {
  typedef T value_type;
  SimpleAllocator(/*ctor args*/);
  template <class U> SimpleAllocator(const SimpleAllocator<U>& other);
  T* allocate(std::size_t n);
  void deallocate(T* p, std::size_t n);
};
template <class T, class U>
bool operator==(const SimpleAllocator<T>&, const SimpleAllocator<U>&);
template <class T, class U>
bool operator!=(const SimpleAllocator<T>&, const SimpleAllocator<U>&);
```

constexpr ALLOCATOR?

```
template <class T, size_t Size>
struct ConstexprAllocator {
  typedef T value_type;
  consstexpr ConstexprAllocator(/*ctor args*/);
  template <class U>
  constexpr ConstexprAllocator(const ConstexprAllocator<U>& other);
  constexpr T* allocate(std::size_t n);
  constexpr  void deallocate(T* p, std::size_t n);
  std::array<std::pair<bool, value_type>, Size> data; // bool for free flag
};
```

Implementation left as an exercise to the reader.

PARSING JSON VALUE LITERALS

Because we need some way to actually turn a string literal into our JSON representation.

WHAT IS A PARSER?

```
Parser a :: String -> [(a, String)]
```

"A parser for things is a function from strings to lists of pairs of things and strings."

– Dr Seuss on parsers

Or in our case something like:

```
template <typename T>
using parser = auto (*)(string) -> list<pair<T, string>>;
```

PARSERS

```
template <typename T>
using parser = auto (*)(string) -> list<pair<T, string>>;
```

Of course, we don't really mean quite this…

- `string` -> `string_view` (compile-time stringlike thing)
- `list` -> `optional` (simpler)
- "function" -> "something invocable"

A SIMPLE PARSER

Let's have a couple of aliases that will make life simpler.

```
using parse_input_t = std::string_view;

template <typename T>
using parse_result_t = cx::optional<cx::pair<T, parse_input_t>>;
```

And let's make a parser that matches a single `char` that we give it.

```
constexpr auto match_char(parse_input_t s, char c) -> parse_result_t<char>
{
  if (s.empty() || s[0] != c) return std::nullopt;
  return parse_result_t<char>(
    cx::make_pair(c, parse_input_t(s.data()+1, s.size()-1)));
}
```

A SIMPLE PARSER

```
// Ceci n'est pas une parser.
constexpr auto match_char(parse_input_t s, char c) -> parse_result_t<char>;
```

`match_char` isn't actually a parser, because it has the wrong signature.

```
// This is the signature of a parser.
template <typename T>
using parser = auto (*)(parse_input_t s) -> parse_result_t<T>;
```

But now that we have `constexpr` lambdas, we can write a function that returns a parser.

A SIMPLE PARSER

```
constexpr auto make_char_parser(char c)
{
  return [=] (parse_input_t s) -> parse_result_t<char> {
    if (s.empty() || s[0] != c) return std::nullopt;
    return parse_result_t<char>(
        cx::make_pair(c, parse_input_t(s.data()+1, s.size()-1)));
  };
}
```

The lambda returned from `make_char_parser` is a parser that will match the given `char`.

MORE USEFUL PRIMITIVE PARSERS

So far we can match one char. Because fundamentally parsing works on "strings", there are a couple of other parsers that will be useful.

```
// parse one of a set of chars
constexpr auto one_of(std::string_view chars)
{
  return [=] (parse_input_t s) -> parse_result_t<char> {
    if (s.empty()) return std::nullopt;
    // basic_string_view::find is supposed to be constexpr, but no...
    auto j = cx::find(chars.cbegin(), chars.cend(), s[0]);
    if (j != chars.cend()) {
      return parse_result_t<char>(
          cx::make_pair(s[0], parse_input_t(s.data()+1, s.size()-1)));
    }
    return std::nullopt;
  };
}
```

MORE USEFUL PRIMITIVE PARSERS

And you can imagine how to write these.

```
// the opposite of one_of: match a char that isn't any of the given set
constexpr auto none_of(std::string_view chars)
{
  return [=] (parse_input_t s) -> parse_result_t<char> {
    ...
  };
}

// match a given string
constexpr auto make_string_parser(std::string_view str)
{
  return [=] (parse_input_t s) -> parse_result_t<std::string_view> {
    // here we could use a constexpr version of std::mismatch...
    ...
  };
}
```

BUILDING UP

So far we have a few primitive parsers.

In order to simply build up more complex parsers, we need to be able to **combine** parsers in various ways.

BUILDING UP

Some basic things we will want to do:

- Change the result type of a parser (`fmap`)
- Run one parser, then a second one based on what the first returned (`bind`)
- Run one parser, and if it fails run another (`operator|`)
- Run two parsers in succession and combine the outputs (`combine`)

(Pick your functional pattern: functor, monad, monoid, applicative…)

CHANGING THE RESULT TYPE (fmap)

```
fmap :: (a -> b) -> Parser a -> Parser b
```

```
template <typename F, typename P>
constexpr auto fmap(F&& f, P&& p)
{
  using R = parse_result_t<std::result_of_t<F(parse_t<P>)>>;
  return [f = std::forward<F>(f),
          p = std::forward<P>(p)] (parse_input_t i) -> R {
           const auto r = p(i);
           if (!r) return std::nullopt;
           return R(cx::make_pair(f(r->first), r->second));
         };
}
```

ALTERNATION (operator|)

```
operator| :: Parser a -> Parser a -> Parser a
```

```
template <typename P1, typename P2,
          std::enable_if_t<std::is_same_v<parse_t<P1>, parse_t<P2>>, int> = 0>
constexpr auto operator|(P1&& p1, P2&& p2) {
  return [=] (parse_input_t i) {
    const auto r1 = p1(i);
    if (r1) return r1;
    return p2(i);
  };
}
```

```
template <typename T>
constexpr auto fail(T) {
  return [=] (parse_input_t) -> parse_result_t<T> {
    return std::nullopt;
  };
}
```

CONJUNCTION (combine)

```
combine :: Parser a -> Parser b -> (a -> b -> c) -> Parser c
```

```
template <typename P1, typename P2, typename F,
          typename R = std::result_of_t<F(parse_t<P1>, parse_t<P2>)>>
constexpr auto combine(P1&& p1, P2&& p2, F&& f) {
  return [=] (parse_input_t i) -> parse_result_t<R> {
           const auto r1 = p1(i);
           if (!r1) return std::nullopt;
           const auto r2 = p2(r1->second);
           if (!r2) return std::nullopt;
           return parse_result_t<R>(
               cx::make_pair(f(r1->first, r2->first), r2->second));
         };
}
```

USEFUL `combine` PATTERNS

```
operator> :: Parser a -> Parser b -> Parser a
operator< :: Parser a -> Parser b -> Parser b
```

```
template <typename P1, typename P2,
          typename = parse_t<P1>, typename = parse_t<P2>>
constexpr auto operator<(P1&& p1, P2&& p2) {
  return combine(std::forward<P1>(p1),
                 std::forward<P2>(p2),
                 [] (auto, const auto& r) { return r; });
}
```

These operators are useful for throwing away the left or right hand side of `combine`.

ACCUMULATING COMBINATORS

And now you begin to see where this is heading…

```
many :: Parser a -> b -> (b -> a -> b)  -> Parser b
many1 :: Parser a -> b -> (b -> a -> b)  -> Parser b
exactly_n :: Parser a -> int -> b -> (b -> a -> b) -> Parser b
separated_by :: Parser a -> Parser x -> b -> (b -> a -> b) -> Parser b
```

These are starting to look like building blocks we can use to parse real things.

SOME SIMPLE EXAMPLES

This parser eats whitespace.

```
constexpr auto skip_whitespace()
{
  constexpr auto ws_parser =
    make_char_parser(' ')
    | make_char_parser('\t')
    | make_char_parser('\n')
    | make_char_parser('\r');
  return many(ws_parser, std::monostate{}, [] (auto m, auto) { return m; });
}
```

SOME SIMPLE EXAMPLES

This parses a decimal integer.

```
constexpr auto int_parser()
{
  return bind(one_of("123456789"sv),
              [] (char x, parse_input_t rest) {
                return many(one_of("0123456789"sv),
                            static_cast<int>(x - '0'),
                            [] (int acc, char c) { return (acc*10) + (c-'0'); })(rest);
              });
}
```

First any non-zero digit, then zero or more digits, building up the integer in the obvious way.

SOME SIMPLE EXAMPLES

This (very simply) parses a string.

```
constexpr auto string_parser(parse_input_t s)
{
  constexpr auto quote_parser = make_char_parser('"');
  const auto str_parser =
    many(none_of("\""sv),
         std::string_view(s.data()+1, 0),
         [] (const auto& acc, auto) {
           return std::string_view(acc.data(), acc.size()+1);
         });
  return (quote_parser < str_parser > quote_parser)(s);
}
```

GETTING TO JSON

We now have a toolkit for building parsers.

```
template <size_t Depth=5>
struct JSON_Value
{
  ...
  struct Data
  {
    bool boolean{false};
    double number{0};
    cx::static_string string;
    cx::vector<JSON_Value<Depth-1>, max_vector_size> array;
    cx::map<cx::static_string, JSON_Value<Depth-1>, max_map_size> object;
  };
  ...
};
```

To parse our JSON value, a reasonable approach is to use alternation on parsers for each type of value.

RECURSIVE PARSING STRUCTURE

```
struct recur
{
  template <std::size_t Depth = max_parse_depth>
  static constexpr auto value_parser()
  {
    constexpr auto p =
      fmap([] (std::string_view) { return JSON_Value<Depth>(std::monostate{}); },
           make_string_parser("null"sv))
      | fmap([] (std::string_view) { return JSON_Value<Depth>(true); },
             make_string_parser("true"sv))
      | fmap([] (std::string_view) { return JSON_Value<Depth>(false); },
             make_string_parser("false"sv))
      | fmap([] (auto n) { return JSON_Value<Depth>(n); },
             number_parser())
      | fmap([] (auto str) { return JSON_Value<Depth>(str); },
             string_parser())
      | array_parser<Depth>()
      | object_parser<Depth>();
    return skip_whitespace() < p;
  }
  ...
```

RECURSIVE PARSING STRUCTURE

```
  ...
  template <std::size_t Depth = max_parse_depth>
  static constexpr auto array_parser() { ... }

  template <std::size_t Depth = max_parse_depth>
  static constexpr auto key_value_parser() { ... }

  template <std::size_t Depth = max_parse_depth>
  static constexpr auto object_parser() { ... }
};

template <>
constexpr auto recur::value_parser<0>() {
  return fail(JSON_Value<0>{});
}

constexpr auto operator "" _json(const char* str, std::size_t len) {
  return recur::value_parser<>()(std::string_view{str, len});
}
```

ERROR MESSAGES

```
template <typename T, typename ErrorFn>
constexpr auto fail(T, ErrorFn f) {
  return [=] (parse_input_t) -> parse_result_t<T> {
    f();
    return std::nullopt;
  };
}

static constexpr auto array_parser() {
  return ...
    > (make_char_parser(']') | fail(']', [] { throw "expected ]"; }));
}
```

It's not a very good story.

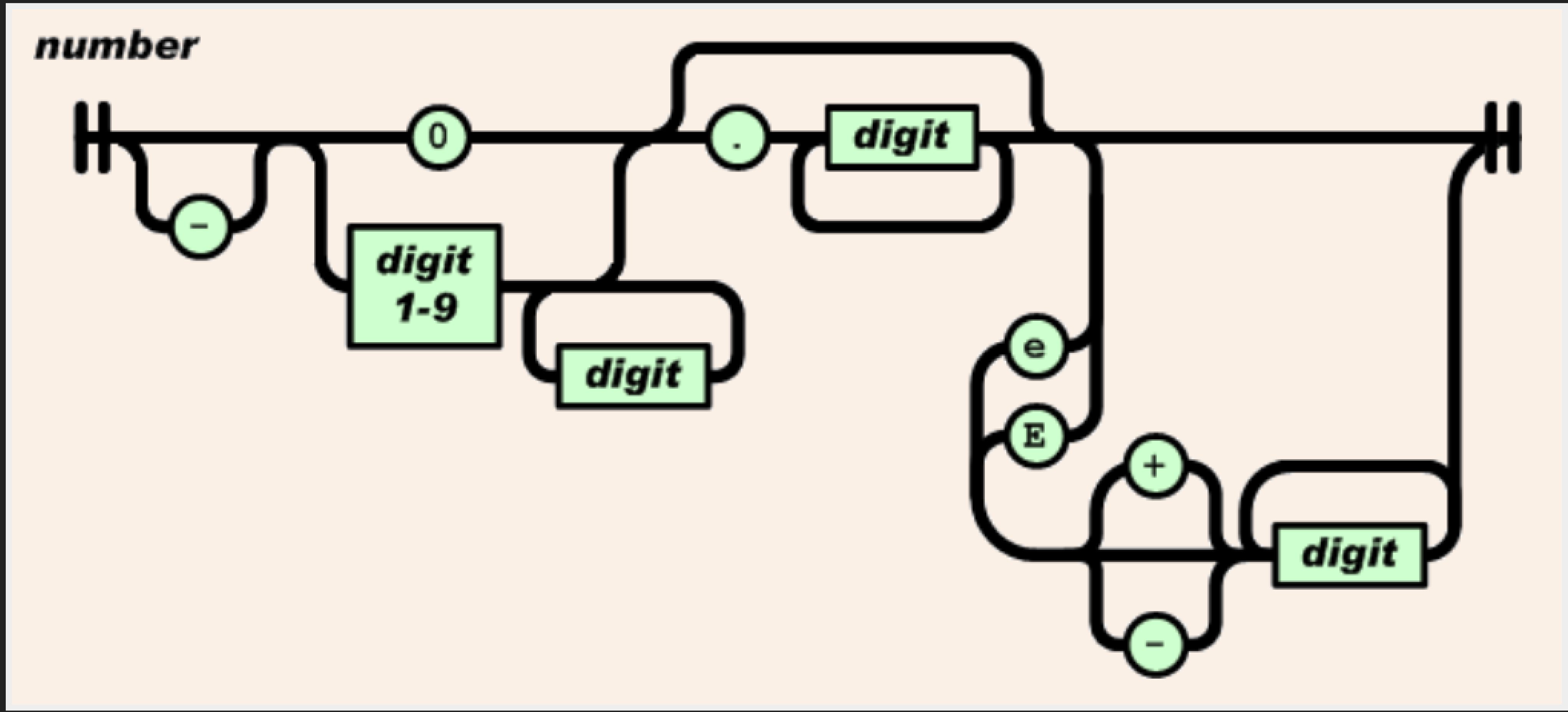
PARSING JSON VALUE LITERALS (BETTER)

What we have so far is the simplest proof-of-concept.

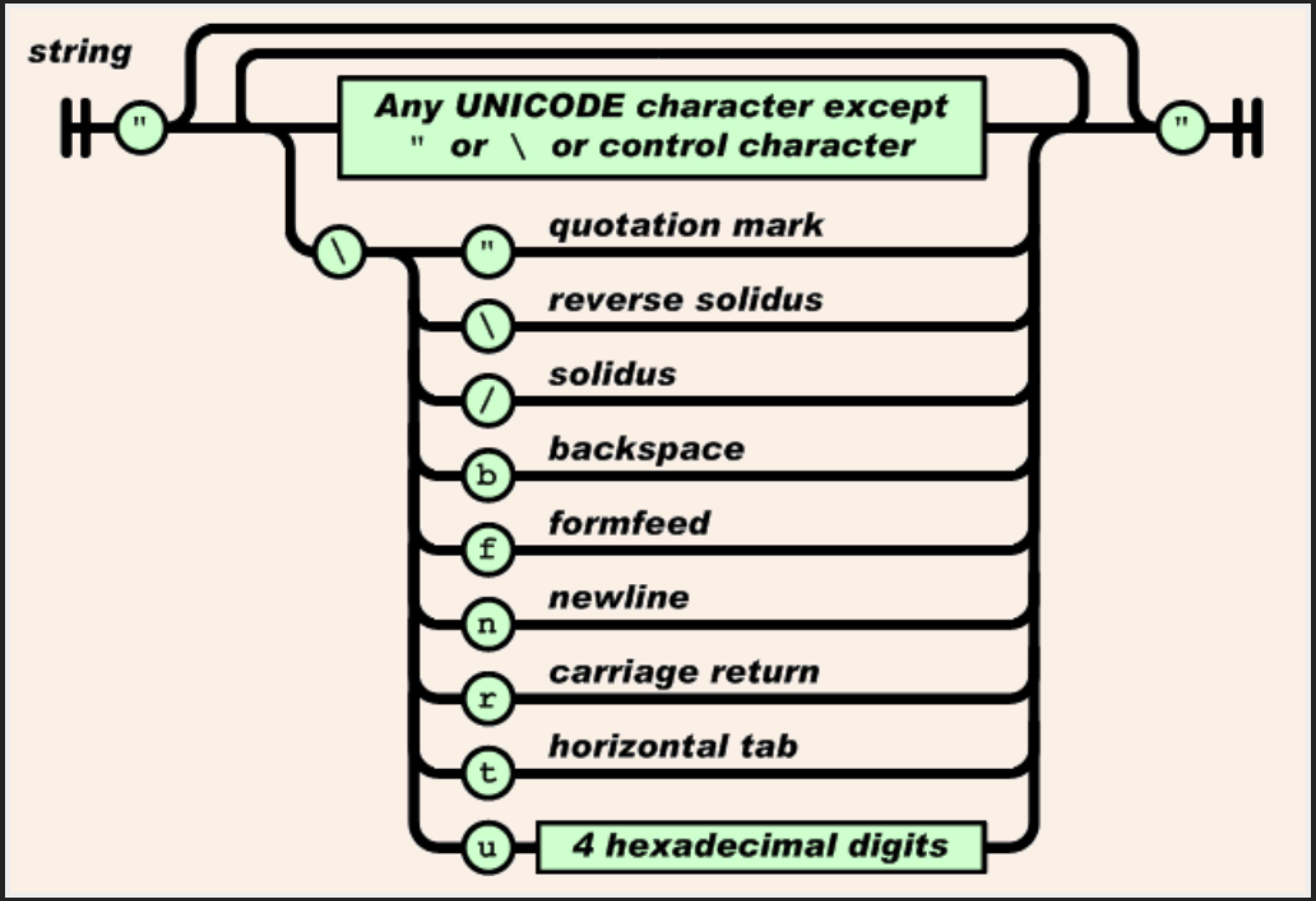
It works (for suitable values of "works").

It's a good starting point, but there are a few problems we need to address.

PROBLEM 1: A JSON NUMBER ISN'T AN `int`



PROBLEM 2: A JSON STRING ISN'T A `string_view`



PROBLEM 3: TEMPLATE INSTANTIATION

```
$ time make
[ 50%] Building CXX object main.cpp.o
[100%] Linking CXX executable constexpr-all-the-things
[100%] Build target constexpr-all-the-things

real    A BLOODY...
user     LONG...
sys       TIME
$
```

PROBLEM 4: ARBITRARY LIMITS

```
constexpr inline std::size_t max_parse_depth{3};

static constexpr size_t max_vector_size{6};
static constexpr size_t max_map_size{6};

namespace cx
{
  using string = basic_string<char, 32>;
}
```

GETTING RID OF TEMPLATE SLOWNESS

All this recursive templatery is a problem.

```
template <size_t Depth=5>
struct JSON_Value
{
  struct Data
  {
    ...
    cx::vector<JSON_Value<Depth-1>, max_vector_size> array;
    cx::map<cx::static_string, JSON_Value<Depth-1>, max_map_size> object;
  };
  ...
};
```

SOLUTION: MORE PARSING!

What we have is a parser for JSON values.

But we could create more parsers…

How about a parser for the **number** of JSON values required?

NUMBER-OF-VALUES PARSER

We can write a parser that computes the number of values in a literal:

- Array -> 1 + number of values in children
- Object -> 1 + number of values in children
- Everything else -> 1

We can reuse some structural components of our value parser, and a number-of-values parser is simpler in many places.

NUMBER-OF-VALUES PARSER

Take the recursive function templates out of our value parser: instead, the struct itself is a template containing the right-sized array of values.

```
template <std::size_t N>
struct recur
{
  using V = cx::vector<JSON_Value, N>;
  V vec{};

  constexpr recur(parse_input_t s) {
    value_parser(vec)(s);
  }

  static constexpr auto value_parser(V& v);
  ...
};
```

NON-TEMPLATED `JSON_Value`

Now we can have a `JSON_Value` that isn't a template.

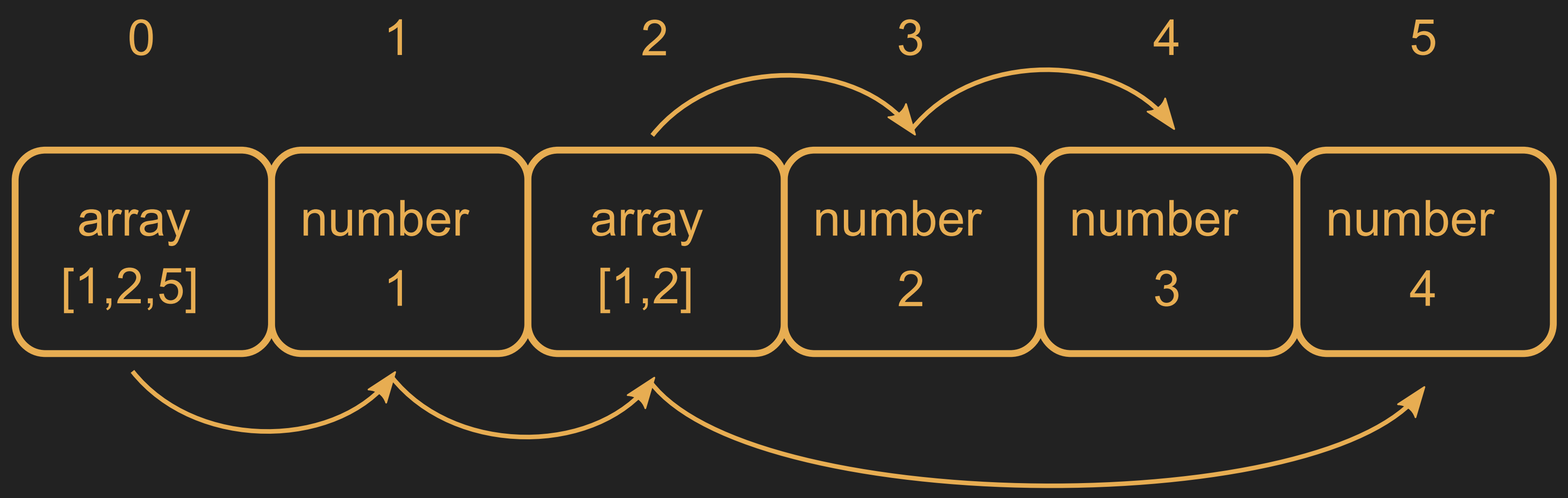
```
struct JSON_Value
{
  struct Data
  {
    ...
    cx::vector<std::size_t, max_vector_size> array;
    cx::map<cx::static_string, std::size_t, max_map_size> object;
  };
  ...
};
```

The array and object values store offsets into the externalized array.

EXAMPLE PARSE

```
constexpr auto jsval = "[1, [2, 3], 4]"_json;
```

Number of values: 6 (2 arrays, 4 numbers)



DRIVING THE PARSE

```
template <char... Cs>
constexpr auto numobjects()
{
  std::initializer_list<char> il{Cs...};
  return numobjects_recur<>::value_parser()(
      std::string_view(il.begin(), il.size()))->first;
}

template <typename T, T... Ts>
constexpr auto operator "" _json()
{
  constexpr std::initializer_list<T> il{Ts...};
  return recur<numobjects<Ts...>()>(
      std::string_view(il.begin(), il.size())).vec;
}
```

PROBLEM 3: SOLVED

Cost: an extra pass

Benefits:

- quicker compilation (no recursive templates!)
- no arbitrary hardcoded limit to depth

```
constexpr auto jsval = "([[[[[[[[[[[[[1]]]]]]]]]]]]])"_json;
static_assert(jsval[0][0][0][0][0][0][0][0][0][0][0][0][0].to_Number() == 1);
```

PROBLEM 4: ARBITRARY LIMITS

We still have limits on:

- string size
- array size
- object (map) size

Can we use the same strategy of precomputing size to combat these?

REMOVING STRING SIZE RESTRICTION

We **can** use the same technique:

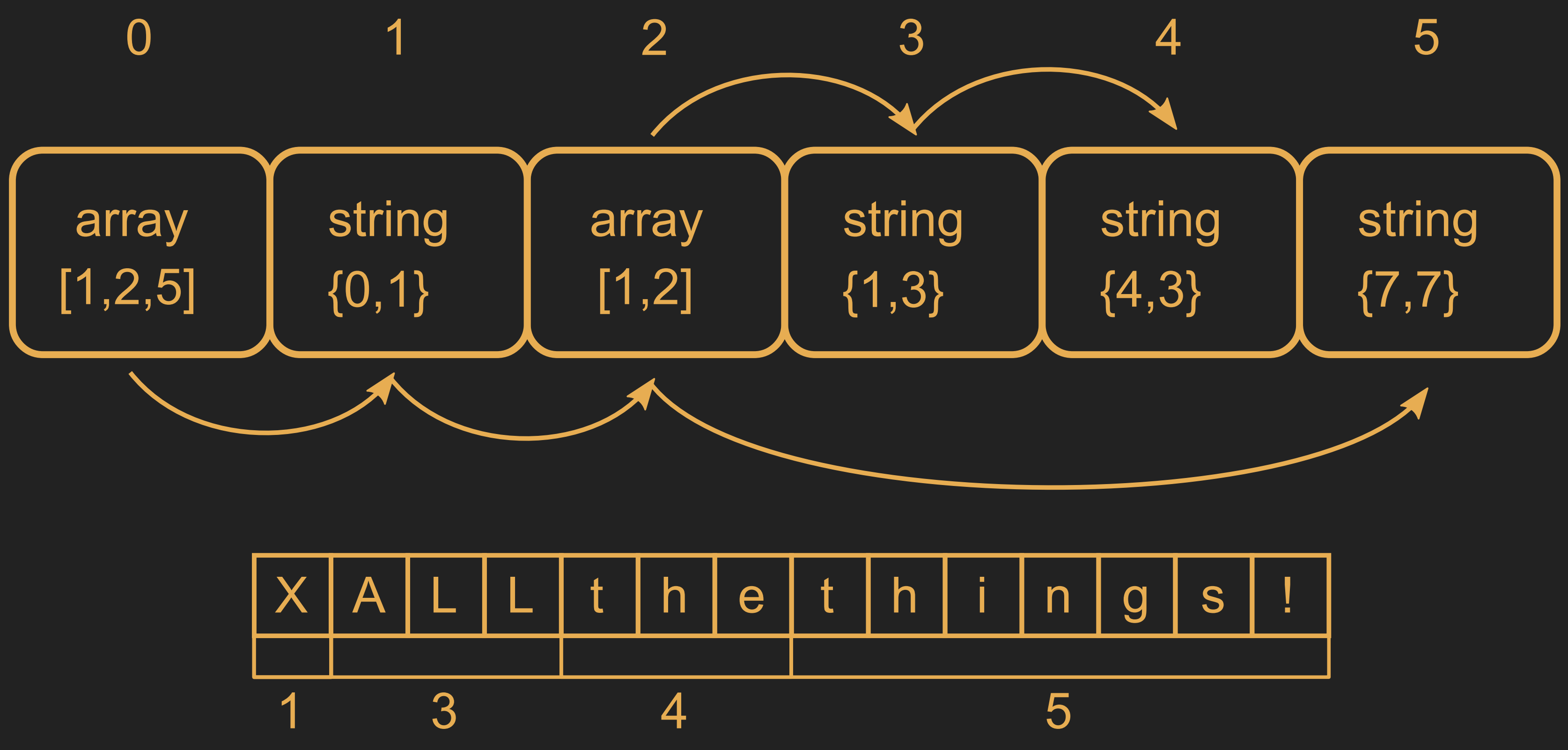
- precompute the total string size for the value
- rightsize a char buffer
- store `{offset, extent}` in the string `JSON_Value` as we parse

We can do the number-of-values and total-string-size computation in a single pass (that returns the pair of sizes).

STRING SIZE LIMIT REMOVED

```
constexpr auto jsval = R"(["X", ["ALL", "the"], "things!"])"_json;
```

Number of values: 6 (2 arrays, 4 strings)
Total string size: 14 (1 + 3 + 3 + 7)



REMAINING LIMITS

We still have limits on:

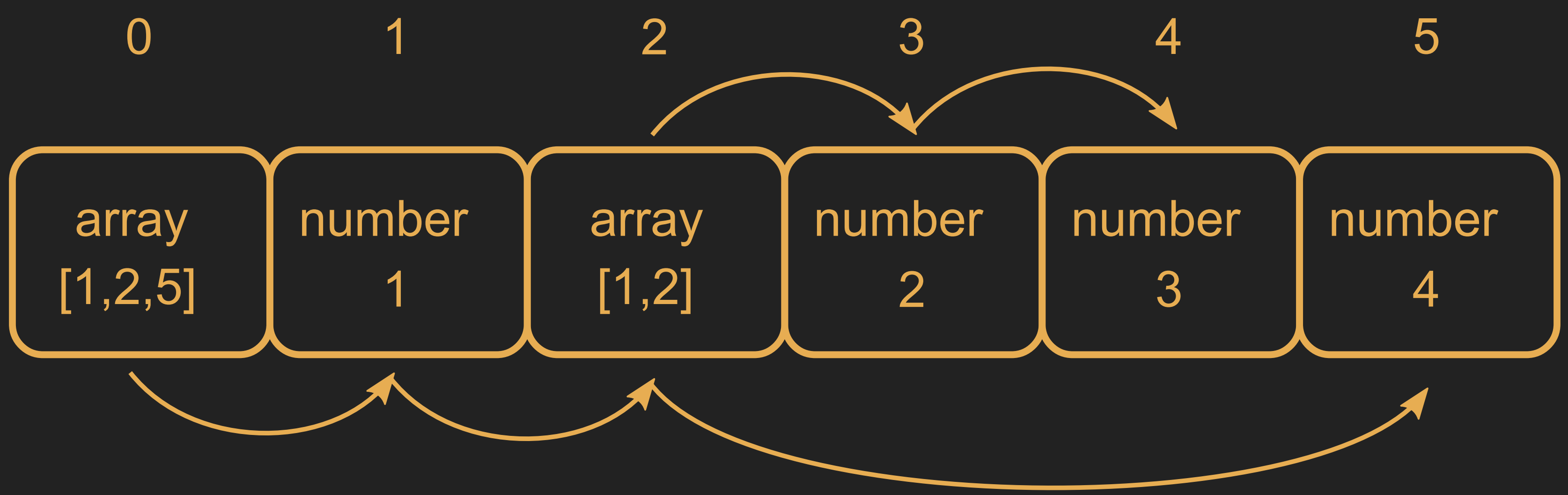
- array size
- object (map) size

We can't naively do the same thing we did with strings, because values within arrays/objects aren't contiguous.

ARRAYS/OBJECTS AREN'T CONTIGUOUS

As we saw before, because of arbitrary nesting.

```
constexpr auto jsval = "[1, [2, 3], 4]"_json;
```



ADD ANOTHER PASS

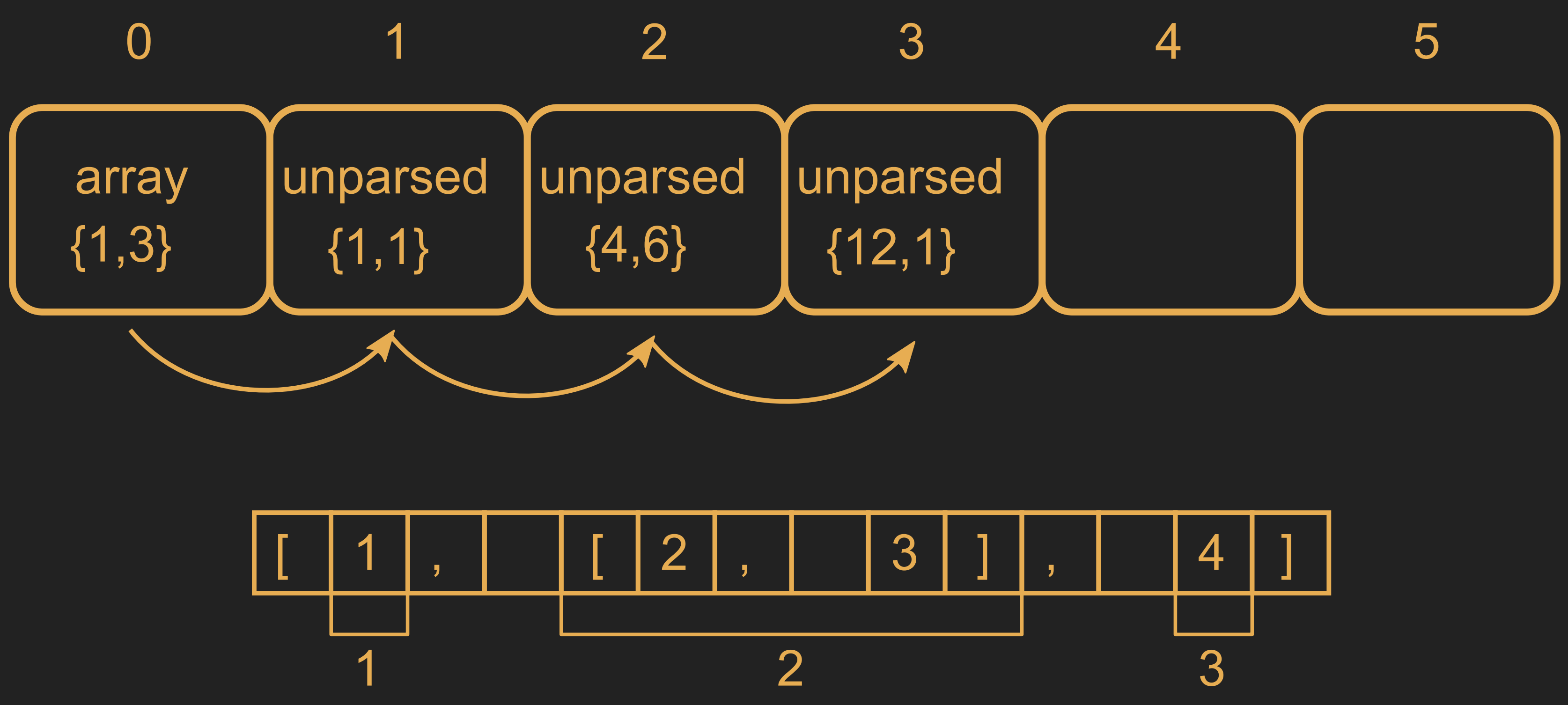
Add a pass to make the parser "breadth-first".

```
struct JSON_Value
{
  union Data
  {
    std::string_view unparsed;
    bool boolean;
    double number;
    ...
  };
  ...
};
```

"BREADTH-FIRST" PARSING

Now the array is parsed contiguously.

```
constexpr auto jsval = "[1, [2, 3], 4]"_json;
```



AS ARRAYS, SO OBJECTS

Arrays are now `{offset, extent}`, so there is no limit on array size.

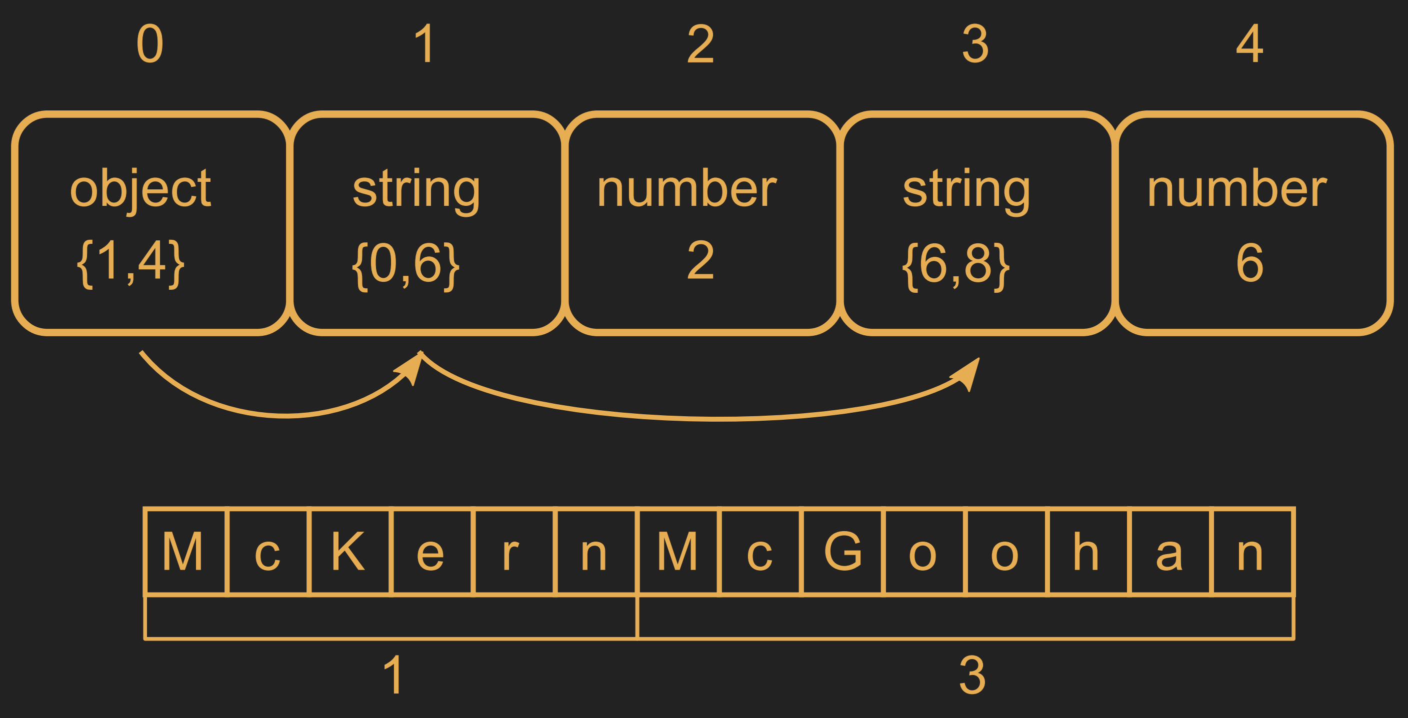
Objects could be arrays of (string, value).

We just need to deal with object keys.

OBJECT STORAGE

Objects are alternating strings and arbitrary values.

```
constexpr auto jsval = R"({"McKern":2,  "McGoohan":6})"_json;
```



FINALLY, NO LIMITS!

```cpp
struct JSON_Value
{
  struct ExternalView {
    std::size_t offset;
    std::size_t extent;
  };

  union Data {
    std::string_view unparsed;
    bool boolean;
    double number;
    ExternalView external_string;
    ExternalView external_array;
    ExternalView external_object;
  };
  ...
};
```

PARSING: CONCLUSION

- `constexpr` lambdas enable composable compile-time parsing
- parser combinators enable more complex literals
- multiple passes can be used thanks to template UDL operators and `string_view`
- adding extra passes can solve almost any problem…
- could parsing be helped by a (good?) C++ concept?

THE FUTURE

THE DESTRUCTOR PROBLEM

Currently any type with a non-trivial destructor cannot be used in `constexpr` context.

trivially destructible quiz time!

is_trivially_destructible

Does this `static_assert` succeed?

```
struct S {
};

static_assert(std::is_trivially_destructible_v<S>);
```

is_trivially_destructible

Does this `static_assert` succeed?

```
struct S {
  int i;
};

static_assert(std::is_trivially_destructible_v<S>);
```

is_trivially_destructible

Does this `static_assert` succeed?

```
struct S {
  std::unique_ptr<int> i;
};

static_assert(std::is_trivially_destructible_v<S>);
```

is_trivially_destructible

Does this `static_assert` succeed?

```
struct S {
  ~S() {}
};

static_assert(std::is_trivially_destructible_v<S>);
```

WHY IS THIS A PROBLEM?

It's easy to build a `constexpr` enabled type that can grow at runtime, or fail to compile if it gets too big in `constexpr` context.

```
struct Container {
  std::array<int, 10> data{};
  std::size_t length = 0;
  int *extra_data = nullptr;
  void push_back(const int i) {
    if (length >= data.size()) {
      if (!extra_data) {
        extra_data = new int[100];
      }
      extra_data[(length++) - data.size()] = i;
    } else {
      data[length++] = i;
    }
  }
};
```

WHY IS THIS A PROBLEM?

But: as soon as we add a destructor, the class is no longer usable in a `constexpr` context.

So we can build this type, but we are required to leak memory
if it grows beyond the static size!

SOLUTIONS TO THE `constexpr` DESTRUCTOR PROBLEM

```
struct Container {
  ~Container() {
    // this proposal allows for an empty destructor to be allowed
    if constexpr(something) {
      // do something
    }
  }
};
```

SOLUTIONS TO THE constexpr DESTRUCTOR PROBLEM

```
struct Container {
  ~Container() {
    // but why not treat it like any other constexpr code?
    // allow it as long as only constexpr allowed actions
    // happen at compile time?
    if (extra_data) {
      delete [] extra_data;
    }
  }
};
```

THE DEBUGGING PROBLEM

On which line does GCC report an error?

```
 1: constexpr int do_something()
 2: {
 3:   int val[1]{};
 4:   return val[1];
 5: }
 6:
 7: int main()
 8: {
 9:   constexpr auto val = do_something();
10: }
```

THE DEBUGGING PROBLEM

Several times during debugging we had to take the code from compile time context to runtime context to allow for actual debugging.

THE DEBUGGING PROBLEM

This proposal adds debugging capability at compile time.

P0596

To: EWG

From: Daveed Vandevoorde (`daveed@edg.com`)

Date: 2017-02-02

`std::constexpr_trace` and `std::constexpr_assert`

constexpr_vector

This other proposal from the same author allows for a special type of `constexpr_vector` that is allowed to grow and shrink at compile time only, requiring compiler support.

P0597

To: EWG

From: Daveed Vandevoorde (`daveed@edg.com`)

Date: 2017-02-02

```
std::constexpr_vector<T>
```

`constexpr` STL POSSIBILITIES

ALGORITHMS

Weakened complexity guarantees on `stable_sort`, `inplace_merge`, `stable_partition`?
(They make use of temporary buffers to improve complexity.)

Are there others that might need to have
weakened complexity guarantees for compile time use?

ITERATORS

If you have a `constexpr` container, you want the iterators to all be `constexpr`.

Many iterators could be `constexpr` and usable in a `constexpr` context if the operations on the corresponding containers are.

e.g. if you have `constexpr push_back` on your `constexpr vector` type, `back_insert_iterator` could easily be `constexpr`.

THE COST

COGNITIVE COST

- Flat data structures are easy to reason about
- `constexpr` code forces you to consider what your code is doing and the lifetime of objects (in a good way).
- Tree-like data structures are difficult to reason about
- Selecting data structure sizes can be difficult
- Error messages from heavily composed lambdas are… challenging to deal with
- Debugging often currently means "go back and think about the types"

COMPILE-TIME COST - DEBUG BUILD

- 6GB RAM!
- >2 Minutes Build Time
- 338K Binary
- Tweaking debug level can have a great effect. This might be related to symbol sizes.

COMPILE-TIME COST - RELEASE BUILD

- 328MB RAM
- 5s Build Time
- 9K Binary

COMPILE-TIME COST - COMPARISON

Using the same nightly build of GCC, how long does this take to compile?

```
#include <regex>

int main()
{
  std::regex attribute(R"(\s+(\S+)\s*=\s*('|")(.*?)\2)");
}
```

5s Debug, 7.5s Release

CONCLUSION

- All but 3 standard algorithms can (easily?) be made `constexpr`
- There are holes around assignment operations in the STL
- Many iterator operations could be made `constexpr` for use with `constexpr` containers
- Some interaction with C, ie `<cmath>` may hold back some operations
- `constexpr` lambdas unlock the potential for complex UDLs
- `constexpr` allocators and `constexpr` destructors would make it possible to unify `constexpr` containers with regular ones

Thanks!

https://github.com/lefticus/constexpr_all_the_things