Constant Fun

Dietmar Kühl
Bloomberg LP

Copyright Notice

Overview

- constexpr motivation
- constexpr overview
- straight forward array implementation
- sorting at compile time

Constant Expressions

- values determined at compile-time:
  - integer, floating point, string literals
  - some meta information like sizeof(x)
  - computations using only constant expressions
  - since C++11: results from constexpr functions (when everything used is a constexpr)

Why constexprs? I

- save run-time: do computations at compile-time
- compute repeatedly used results just once (actually: once per translation unit where used)
- similar to hoisting computations out of a loop
- can be used by compiler for optimisations, e.g., no branch if condition always true/false

Why constexprs? II

- some language constructs require constants:
  - size of built-in arrays
  - non-type template arguments
  - values given to enumerators
  - values used for case labels

Why constexprs? III

- constexpr objects are initialised early
  - that's not necessarily true for const objects
- there is no concern about order of initialisation
- .. or multiple threads trying to initialise the object

constexpr Functions

- needed to allow abstractions to be constexpr
- e.g. bitmask enum without constexpr functions:
  - implicit bit operators have wrong type: int
  - ...and are not usable with class enums
  - user-defined operators not everywhere usable

Bitmask vs. constexpr

```
enum          bm { b0 = 0x1, b1 = 0x2, b2 = 0x4 };



int main() {
    int b = bm::b0 | bm::b1;
    switch (b) {
        case bm::b0 | bm::b1: /*…*/;
    }
}
```

Bitmask vs. constexpr

```
enum          bm { b0 = 0x1, b1 = 0x2, b2 = 0x4 };



int main() {
    bm b = bm::b0 | bm::b1; // bad conversion
    switch (b) {
        case bm::b0 | bm::b1: /*…*/;
    }
}
```

Bitmask vs. constexpr

```
enum          bm { b0 = 0x1, b1 = 0x2, b2 = 0x4 };




int main() {
    bm b = bm(bm::b0 | bm::b1);
    switch (b) {
        case bm::b0 | bm::b1: /*…*/;
    }
}
```

Bitmask vs. constexpr

```
enum class bm { b0 = 0x1, b1 = 0x2, b2 = 0x4 };



int main() {
    bm b = bm(bm::b0 | bm::b1); // no such op
    switch (b) {
        case bm::b0 | bm::b1: /*…*/;
    }
}
```

Bitmask vs. constexpr

```
enum class bm { b0 = 0x1, b1 = 0x2, b2 = 0x4 };

             bm operator| (bm v0, bm v1) {
   return bm(int(v0) | int(v1));
}
int main() {
   bm b = bm::b0 | bm::b1;
   switch (b) {
      case bm::b0 | bm::b1: /*…*/; // not constexpr
   }
}
```

Bitmask vs. constexpr

```
enum class bm { b0 = 0x1, b1 = 0x2, b2 = 0x4 };

constexpr bm operator| (bm v0, bm v1) {
    return bm(int(v0) | int(v1));
}
int main() {
    bm b = bm::b0 | bm::b1;
    switch (b) {
        case bm::b0 | bm::b1: /*…*/;
    }
}
```

constexpr Functions

- non-virtual; return and arguments: literal types
- C++11: ctors with empty body and single statement (return) functions
- C++14: functions can’t have asm, goto, label, try-block; variables must be of literal type, non-static, non-thread-local, initialised

Literal Types

- void, scalar types, references
- arrays of literal types
- classes with some restrictions:
  - trivial destructor (can be =default)
  - either aggregate or with constexpr ctor
  - members/bases are non-volatile literal types

Constant Expression

- doesn't use any of the following

this outside constexpr member	non-constexpr function
exceed implementation limits	invoke undefined behaviour
access non-constexpr data	conversion from void*
dynamic_cast/reinterpret_cast	pseudo dtor
new/delete expression	throw expression
unspecified relational/equality op	typeid on polymorphic object
odr use of this/local var in lambda	names not obeying constraints(*)

(*) allowed: constexpr & introduced during eval

Names in constexpr

- arbitrary names can’t be used
- use of [qualified] names is constrained:
  - names referring to constexprs are allowed
  - names introduced within constexpr can be used

Name Example

```
template <typename T>
constexpr size_t size(initializer_list<T> init) {
    constexpr size_t s = init.size();        // not OK
    return s;
}

int main() {
    constexpr size_t s = size({ 1, 2, 3 });
}
```

Name Example

```
template <typename T>
constexpr size_t size(initializer_list<T> init) {
                    size_t s = init.size();            // OK
    return s;
}

int main() {
    constexpr size_t s = size({ 1, 2, 3 });    // OK
}
```

Name Implication

- argument values cannot be used as constexpr
- initializer_list<T>::size() can't affect result type
- different result types imply at least one of
  - argument types differ somehow
  - the number of arguments differ

Objective

- create a constexpr map template:
  - map strings to enumeration values
  - map strings to factory functions
- constexpr => the map is readily initialised

(string in the general sense, not std::string)

Attempt: Use an Array

```
constexpr pair<string, int> map[] = {
    pair<string, int>(“one”, 1),
    make_pair(“two”, 2),
    { “three”, 3 }
};
template <typename P, size_t N, typename K>
auto constexpr access(P const (&a)[N], K k) {
  P const* it = find_if(begin(a), end(a), match1st(k));
  return it == end(a)? throw “not found”: it->second;
}
```

Attempt: Use an Array

```
constexpr pair<string, int> map[] = {
    pair<string, int>(“one”, 1),
    make_pair(“two”, 2),
    { “three”, 3 }
};
template <typename P, size_t N, typename K>
auto constexpr access(P const (&a)[N], K k) {
  P const* it = find_if(begin(a), end(a), match1st(k));
  return it == end(a)? throw “not found”: it->second;
}
```

Attempt: Use an Array

```
constexpr pair<string_view, int> map[] = { // C++17
    pair<string_view, int>("one", 1),       // C++17
    make_pair("two", 2),
    { "three", 3 }
};
template <typename P, size_t N, typename K>
auto constexpr access(P const (&a)[N], K k) {
   P const* it = find_if(begin(a), end(a), match1st(k));
   return it == end(a)? throw "not found": it->second;
}
```

Detour: string_view

- replacement for std::string when only reading
- converts from char const* and std::string
- provides read-only view of underlying sequence
- supports the corresponding string members
- can be changed itself (the subrange referenced)
- is a literal type

Detour: string_view

- replacement for std::string when only reading
- converts from char const* and std::string
- provides read-only view of underlying sequence
- supports the corresponding string members
- can be changed itself (the subrange referenced)
- is a literal type

Detour: string_view

- a simple implementation for constexpr strings:

```
class string_view {
    char const* b, * e;
public:
    constexpr string_view(char const* s)
        : b(s), e(find(s, unreachable(), '\0')) {}
    constexpr char const* begin() const { return b; }
    constexpr char const* end() const   { return e; }
    // …
};
```

Detour: string_view

```
struct unreachable {
    template <typename T> friend constexpr
    bool operator!= (T, unreachable) { return true; }
};

template <typename I, typename E, typename V>
constexpr I find(I it, E end, V value) {
    while (it != end && *it != value) ++it;
    return it;
}
```

Detour: string_view

```
struct unreachable {
    template <typename T> friend constexpr
    bool operator!= (T, unreachable) { return true; }
};

template <typename I, typename E, typename V>
constexpr I find(I it, E end, V value) {
    return it == end || *it == value
        ? it == find(it + 1, end, value);
}
```

Detour: string_view

- string_view user-defined literal (not in C++17)

```
namespace udl {
    constexpr string_view operator""_sv(
        char const* s, size_t) { return string_view(s); }
}
int main() {
    using namespace udl;
    f("one"_sv);
}
```

Attempt: Use an Array

```
constexpr pair<string_view, int> map[] = { // C++14
    pair<string_view, int>("one", 1),       // C++14
    make_pair("two", 2),                     // C++14
    { "three", 3 }
};
template <typename P, size_t N, typename K>
auto constexpr access(P const (&a)[N], K k) {
  P const* it = find_if(begin(a), end(a), match1st(k));
  return it == end(a)? throw "not found": it->second;
}
```

Attempt: Use an Array

```
constexpr pair<string, int> map[] = {
    pair<string, int>(“one”, 1),
    make_pair(“two”, 2),
    { “three”, 3 }
};
template <typename P, size_t N, typename K>
auto constexpr access(P const (&a)[N], K k) {
  P const* it = find_if(begin(a), end(a), match1st(k));
  return it == end(a)? throw “not found”: it->second;
}
```

Attempt: Use an Array

```
constexpr pair<string, int> map[] = {
    pair<string, int>(“one”, 1),
    make_pair(“two”, 2),
    { “three”, 3 }
};
template <typename P, size_t N, typename K>
auto constexpr access(P const (&a)[N], K k) {
  P const* it = find_if(begin(a), end(a), match1st(k));
  return it == end(a)? throw “not found”: it->second;
}
```

match1st()

```
template <typename T>
struct matcher1st {
    T d_value;
    template <typename P>
    constexpr bool operator()(P const& p) const {
        return this->d_value == p.first;
    }
};
template <typename T>
constexpr matcher1st<T> match1st(T value) {
    return matcher1st<T>{ value };
}
```

Attempt: Use an Array

```
constexpr pair<string, int> map[] = {
     pair<string, int>(“one”, 1),
     make_pair(“two”, 2),
     { “three”, 3 }
};
template <typename P, size_t N, typename K>
auto constexpr access(P const (&a)[N], K k) {
    P const* it = find_if(begin(a), end(a), match1st(k));
    return it == end(a)? throw “not found”: it->second;
}
```

Attempt: Use an Array

```
constexpr pair<string, int> map[] = {
    pair<string, int>(“one”, 1),
    make_pair(“two”, 2),
    { “three”, 3 }
};
template <typename P, size_t N, typename K>
auto constexpr access(P const (&a)[N], K k) {
  P const* it = find_if(begin(a), end(a), match1st(k));
  return it == end(a)? throw “not found”: it->second;
}
```

Use an Array

```
constexpr pair<string, int> map[] = { { "three", 3 } };
template <typename P, size_t N, typename K>
auto constexpr access(P const (&a)[N], K k) {
    P const* it = find_if(begin(a), end(a), match1st(k));
    return it == end(a)? throw "not found": it->second;
}
int main() {
    constexpr int three = access(map, "three"_sv);
                  int four = access(map, "four"_sv);
    constexpr int five = access(map, "five"_sv); // error
}
```

Validity of constexpr Fun

- constexpr functions can use all expressions!
- unless a prohibited expression is used when a constexpr is needed

```
constexpr bool fun(bool v) { return v? v: throw "bad";}
int main() {
                    bool b0 = fun(false); // throws
    constexpr bool b1 = fun(false); // compile fails
}
```

Validity of constexpr Fun

- constexpr functions can use all expressions!
- unless a prohibited expression is used when a constexpr is needed

```
constexpr bool fun(bool v) { return v? v: throw "bad";}
int main() {
                    bool b0 = fun(true); // OK
    constexpr bool b1 = fun(true); // OK
}
```

Use an Array

- it works, e.g. (yes, I’m aware that this is silly):

```
pair<string_view, int> m[] = {{“one”, 1} /*…*/ };
int main(int ac, char* av) {
    string_view key(av[ac != 1]);
    switch (map_access(m, key)) {
    case map_access(m, “one”_sv): cout << “one”;
    }
}
```

- not sorted, may have duplicates

constexpr Map Members

- initializer_list<T> won't work:
    - itself it is a temporary: can't be used directly
    - capturing the content would change the type
- specifying the size is a bit annoying
- deducing the size requires a factory function

Deducing Type

```
template <typename K, typename V, int N>
class map;

template <typename… P>
auto constexpr make_map(P… p) ->
    map<first_t<P>,
        second_t<P>,
        sizeof…(p)> {
    /*…*/
}
```

Deducing Type

```
template <typename K, typename V, int N>
class map;

template <typename… P>
auto constexpr make_map(P… p) ->
    map<first_t<common_type_t<P…>>,
           second_t<common_type_t<P…>>,
           sizeof…(p)> {
    /*…*/
}
```

Deducing Type

```
template <typename K, typename V, int N>
class map;

template <typename… P>
auto constexpr make_map(P… p) ->
    map<common_type_t<first_t<P>…>,
            common_type_t<second_t<P>…>,
            sizeof…(p)> {
    /*…*/
}
```

Deducing Type

```
template <typename K, typename V, int N>
class map;

template <typename… F, typename…S>
auto constexpr make_map(pair<F, S>… p) ->
    map<common_type_t<F…>,
             common_type_t<S…>,
             sizeof…(p)> {
    /*…*/
}
```

Sorting Elements

```
template <typename T, size_t N>
constexpr array<T, N> msort(array<T, N> a) {
  return merge(
      msort(first half of a),
      msort(second half of a));
}

constexpr auto a
   = msort(array<int, 7>{{ 1, 7, 2, 6, 3, 5, 4 }});
```

Sorting Elements

```
template <typename T, size_t N>
constexpr array<T, N> msort(array<T, N> a) {
    if constexpr (1u < N) // C++17
        return merge(
            msort(first half of a),
            msort(second half of a));
    else
        return a;
}
```

Sorting Elements

```
template <typename T, size_t N,
                    typename = enable_if_t<!(1u < N>)>

constexpr array<T, N> msort(array<T, N> a) {
    return a;
}

template <typename T, size_t N,
                    typename = enable_if_t<1u < N>>
constexpr array<T, N> msort(array<T, N> a) { … }
```

Sorting Elements

```
template <typename T, size_t N,
                typename = enable_if_t<!(1u < N>),
                typename = void>
constexpr array<T, N> msort(array<T, N> a) {
    return a;
}

template <typename T, size_t N,
                typename = enable_if_t<1u < N>>
constexpr array<T, N> msort(array<T, N> a) { … }
```

Sorting Elements

```
template <typename T, size_t N>
constexpr array<T, N> msort(array<T, N> a) {
  return merge(
      msort(first half of a),
      msort(second half of a));
}

constexpr auto a
   = msort(array<int, 7>{{ 1, 7, 2, 6, 3, 5, 4 }});
```

Sorting Elements

```
template <typename T, size_t N>
constexpr array<T, N> msort(array<T, N> a) {
  return merge(
      msort(select(a, indices for first half)),
      msort(select(a, indices for second half));
}

constexpr auto a
   = msort(array<int, 7>{{ 1, 7, 2, 6, 3, 5, 4 }});
```

Sorting Elements

- selecting elements base on indices

```
template <typename T, size_t N, size_t… I>
constexpr array<T, sizeof…(I)>
select(array<T, N> a, ???) {
    return array<T, sizeof…(I)>{ a[I]… };
}
```

Sorting Elements

- selecting elements base on indices

```
template <typename T, size_t N, size_t… I>
constexpr array<T, sizeof…(I)>
select(array<T, N> a, index_sequence<I…>) {
    return array<T, sizeof…(I)>{ a[I]… };
}
```

Sorting Element

- create a sequence of indices

```
template <size_t B, size_t... I>
constexpr auto mkseq(index_sequence<I...>) {
    return index_sequence<(B + I)...>();
}
```

Sorting Element

- create a sequence of indices

```
template <size_t B, size_t... I>
constexpr auto mkseq(index_sequence<I...>) {
    return index_sequence<(B + I)...>();
}
template <size_t B, size_t S>
constexpr auto mkidxs() {
    return mkseq<B>(make_index_sequence<S>());
}
```

Sorting Elements

```
template <typename T, size_t N>
constexpr array<T, N> msort(array<T, N> a) {
  return merge(
      msort(select(a, mkidxs<0, (N+1)/2>())),
      msort(select(a, mkidxs<(N+1)/2, N/2>()));
}

constexpr auto a
   = msort(array<int, 7>{{ 1, 7, 2, 6, 3, 5, 4 }});
```

Sorting Elements

```
template <typename T, size_t N>
constexpr array<T, N> msort(array<T, N> a) {
  return merge(
      msort(select(a, mkidxs<0, (N+1)/2>())),
      msort(select(a, mkidxs<(N+1)/2, N/2>()));
}

constexpr auto a
   = msort(array<int, 7>{{ 1, 7, 2, 6, 3, 5, 4 }});
```

Sorting Elements

```
template <typename T, size_t N1, size_t N2>
constexpr array<T, N1 + N2>
merge(array<T, N1> a1, array<T, N2> a2) {
    return a1[0] < a2[0]
        ? cons(a1[0], merge(tail(a1), a2))
        : cons(a2[0], merge(a1, tail(a2)));
}
```

Sorting Elements

```
template <typename T, size_t N>
constexpr array<T, N>
merge(array<T, 0>, array<T, N> a) { return a; }
template <typename T, size_t N>
constexpr array<T, N>
merge(array<T, N> a, array<T, 0>) { return a; }

template <typename T, size_t N1, size_t N2>
constexpr array<T, N1 + N2>
merge(array<T, N1> a1, array<T, N2> a2) { … }
```

Sorting Elements

```
template <typename T, size_t N1, size_t N2>
constexpr array<T, N1 + N2>
merge(array<T, N1> a1, array<T, N2> a2) {
    return a1[0] < a2[0]
        ? cons(a1[0], merge(tail(a1), a2))
        : cons(a2[0], merge(a1, tail(a2)));
}
```

Sorting Elements

```
template <typename T, std::size_t N>
constexpr cf::array<T, N - 1u>
tail(cf::array<T, N> a) {
    return select(a, mkidxs<1u, N - 1u>());
}
```

Sorting Elements

```
template <typename T, size_t N1, size_t N2>
constexpr array<T, N1 + N2>
merge(array<T, N1> a1, array<T, N2> a2) {
    return a1[0] < a2[0]
        ? cons(a1[0], merge(tail(a1), a2))
        : cons(a2[0], merge(a1, tail(a2)));
}
```

Sorting Elements

```
template <typename T, size_t N, size_t...I>
constexpr array<T, N + 1u>
cons_(T v, array<T, N> a, index_sequence<I...>) {
    return array<T, N + 1>{{ v, a[I]... }};
}
```

Sorting Elements

```
template <typename T, size_t N, size_t...I>
constexpr array<T, N + 1u>
cons_(T v, array<T, N> a, index_sequence<I...>) {
    return array<T, N + 1>{{ v, a[I]... }};
}
template <typename T, size_t N>
constexpr array<T, N+1> cons(T v, array<T, N> a) {
    return cons_(v, a, mkidxs<0, N>());
}
```

Sorting Elements

- sorting elements does work using constexpr
- the code [mostly] works with C++11
    - as shown integer_sequences needs C++14
    - it *can* be done using C++11
- essentially a functional approach

Sorting Elements

```
template <typename... T>
auto constexpr sort_(T&&... value) {
    using type = common_type_t<T...>;
    array<type, sizeof...(T)> array{{ value... }};
    std::sort(array.begin(), array.end());
    return array;
}
```

Sorting Elements

```
enum none {};
auto constexpr sort_() { return array<none, 0>(); }

template <typename... T>
auto constexpr sort_(T&&... value) {
    using type = common_type_t<T...>;
    array<type, sizeof...(T)> array{{ value... }};
    std::sort(array.begin(), array.end());
    return array;
}
```

Sorting Elements

```
enum none {};
auto constexpr sort_() { return array<none, 0>(); }

template <typename... T>
auto constexpr sort_(T&&... value) {
    using type = common_type_t<T...>;
    array<type, sizeof...(T)> array{{ value... }};
    sort(array.begin(), array.end());
    return array;
}
```

Sorting Elements

```
template <typename RndIt>
constexpr void sort(RndIt begin, RndIt end) {
    if (distance(begin, end) <= 1) return;
    RndIt pivot = end - 1;
    RndIt mid = partition(begin, pivot, *pivot);
    swap(*mid, *pivot);
    sort(begin, mid);
    sort(mid + 1, end);
}
```

Sorting Elements

```
template <typename RndIt, typename Value>
constexpr RndIt
partition(RndIt begin, RndIt end, Value const& pivot) {
  while (true) {
     while (begin!= end && *begin < pivot) ++begin;
     while (begin!= end && !(*--end < pivot));
     if (begin==end) break;
     swap(*begin, *end); ++begin;
  }
  return begin;
}
```

Sorting Elements

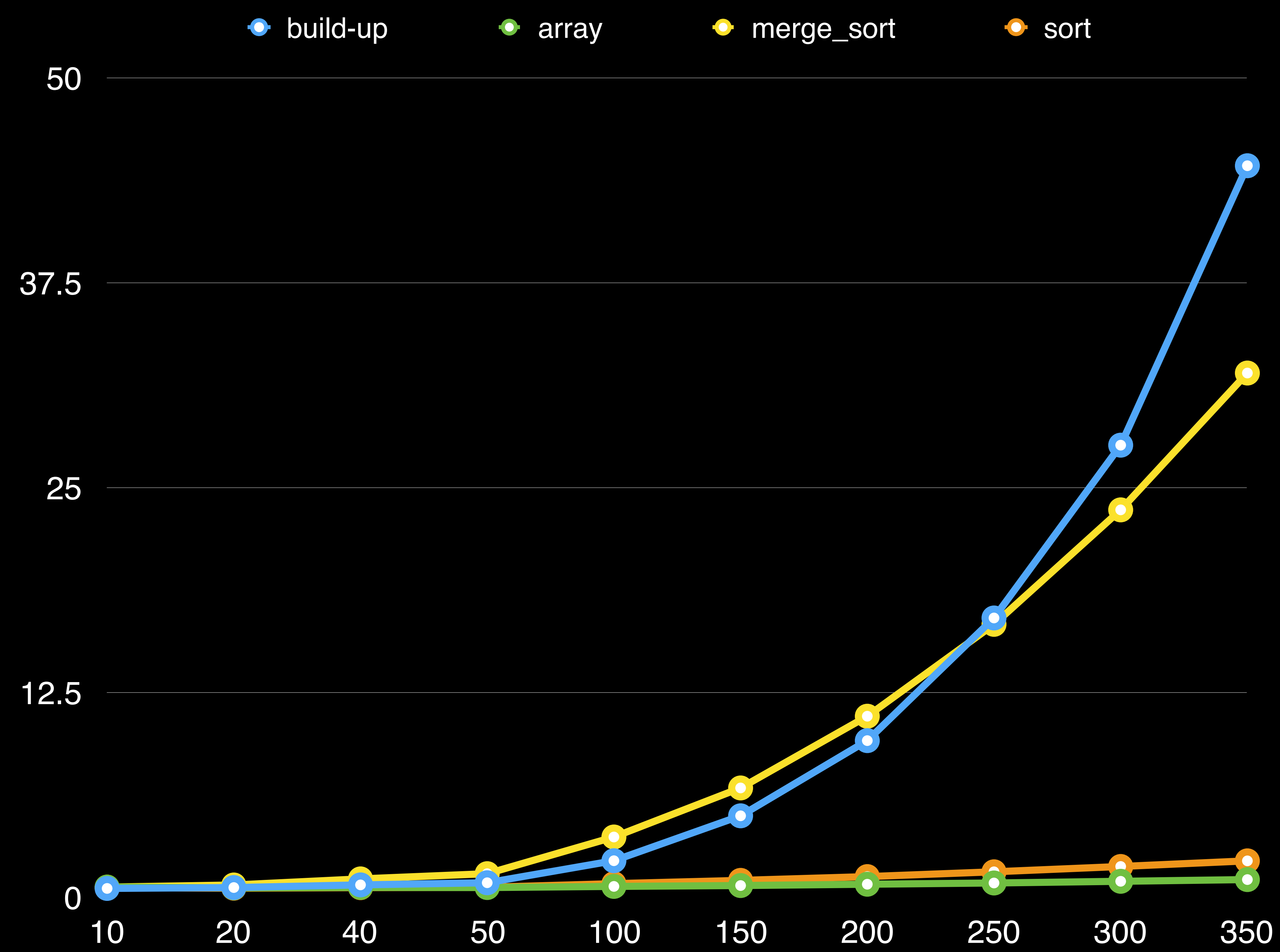
```
template <typename T>
constexpr void swap(T& a, T& b)
     noexcept(noexcept(T(declval<T&&>()))
     && noexcept(declval<T&>()=declval<T&&>()))
{
   T tmp(move(a));
   a = move(b);
   b = move(tmp);
}
```

Constexpr Factory Map

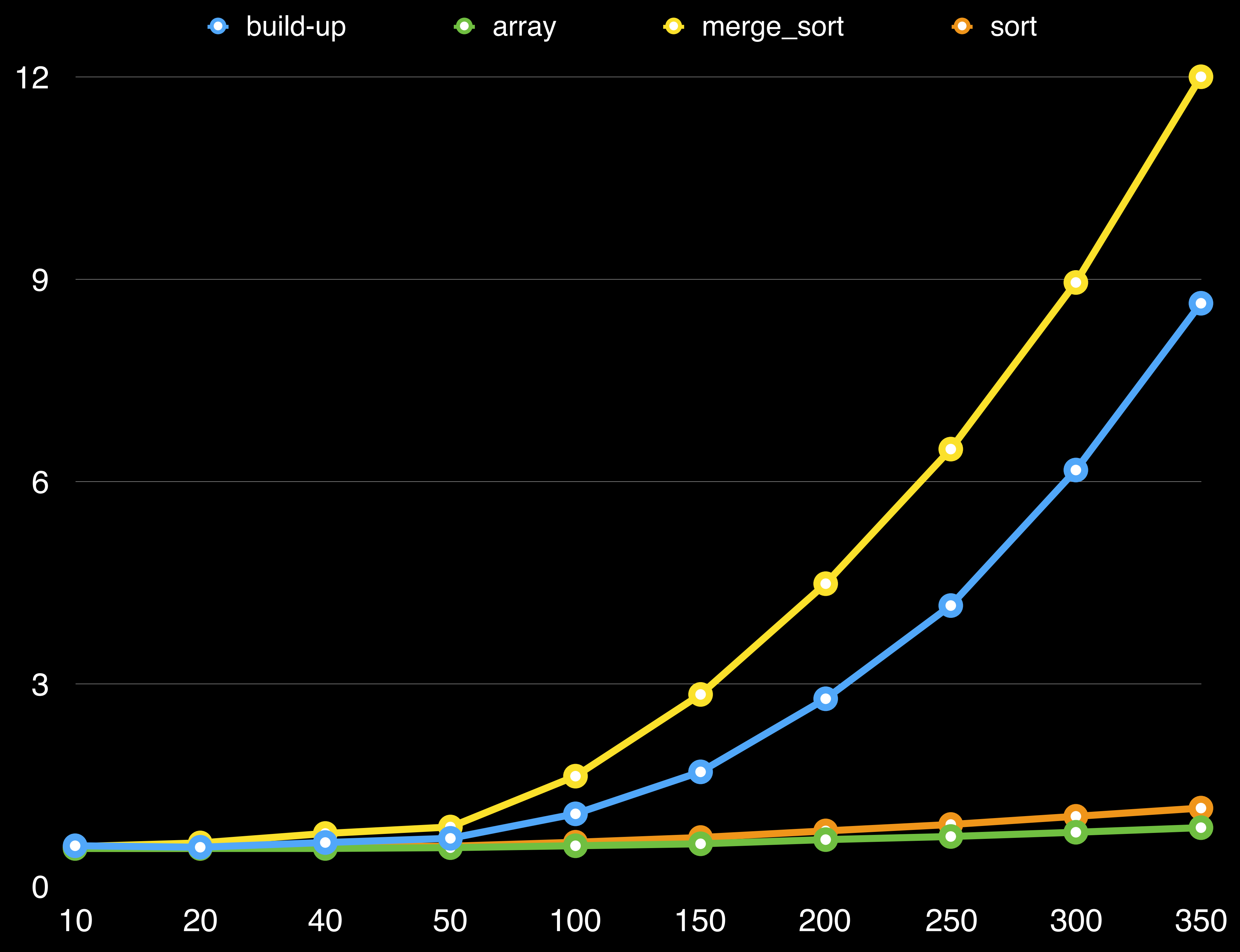
```
struct base { virtual ~base() {} };
struct foo: base {};
struct bar: base {};
template <typename T> base* make();

constexpr auto fac = make_map(
    make_pair( “foo"_sv, make<foo> ),
    make_pair( “bar"_sv, make<bar> ));
// …
unique_ptr<base> p = fac[name]();
```

Times gcc


build-up
array
merge_sort
sort
50
37.5
25
12.5
0
10
20
40
50
100
150
200
250
300
350

Times clang


build-up
array
merge_sort
sort
12
9
6
3
0
10
20
40
50
100
150
200
250
300
350

C++17 constexpr

- lambda objects can be constexpr
- the lambda function call can be constexpr
    - both implicit and explicit
- number of components are declared constexpr
- std::string_view does support constexpr strings

Conclusions

- C++11 constexpr functions are very powerful
- C++14 constexpr functions are more powerful
- a lot of standard library components are not [yet] declared to be constexpr - not even in C++17
  - some members of existing literal types
  - algorithms, function objects, etc.

Questions

?