`initializer_list`s Are Broken - Let's Fix Them

Jason Turner

- First used C++ in 1996, professionally since 2002
- Co-host of CppCast http://cppcast.com
- Host of C++ Weekly https://www.youtube.com/c/JasonTurner-lefticus
- Co-creator of ChaiScript http://chaiscript.com
- Curator of http://cppbestpractices.com
- Microsoft MVP for C++ 2015-present

Jason Turner

Independent and available for training or contracting

- http://articles.emptycrate.com/idocpp
- http://articles.emptycrate.com/training.html

About my Talks

- Move to the front!
- Please interrupt and ask questions
- This is approximately what my training looks like

What Are Initializer Lists?

[dcl.init.list]

> *List-initialization is initialization of an object or reference from a braced-init-list. Such an initializer is called an initializer list, and the comma-separated initializer-clauses of the initializer-list or designated-initializer-clauses of the designated-initializer-list are called the elements of the initializer list.*

What Are Initializer Lists?

[dcl.init.list]

[Note: List-initialization can be used

- as the initializer in a variable definition ([dcl.init])
- as the initializer in a new-expression
- in a return statement
- as a for-range-initializer
- as a function argument ([expr.call])
- as a subscript

What Are Initializer Lists?

[dcl.init.list]

- as an argument to a constructor invocation ([dcl.init], [expr.type.conv])
- as an initializer for a non-static data member
- in a mem-initializer
- on the right-hand side of an assignment

What Are Initializer Lists?

[dcl.init.list]

[Example:

```cpp
int a = {1};
std::complex<double> z{1,2};
new std::vector<std::string>{"once","upon","a","time"}; //4 string elements
f( {"Nicholas","Annemarie"} );  // pass list of two elements
return { "Norah" };             // return list of one element
int* e {};                      // initialization to zero / null pointer
x = double{1};                  // explicitly construct a double
std::map<std::string,int> anim = {{"bear",4},{"cassowary",2},{"tiger",7}};
```

— end example] — end note]

What Are Initializer Lists?

[dcl.init.list]

List-initialization of an object or reference of type T is defined as follows:

What Are Initializer Lists?

[dcl.init.list]

- If the braced-init-list contains a designated-initializer-list, T shall be an aggregate class. The ordered identifiers in the designators of the designated-initializer-list shall form a subsequence of the ordered identifiers in the direct non-static data members of T. Aggregate initialization is performed ([dcl.init.aggr])

What Are Initializer Lists?

[dcl.init.list]

- If T is an aggregate class and the initializer list has a single element of type cv U, where U is T or a class derived from T, the object is initialized from that element (by copy-initialization for copy-list-initialization, or by direct-initialization for direct-list-initialization).

What Are Initializer Lists?

[dcl.init.list]

- Otherwise, if T is a character array and the initializer list has a single element that is an appropriately-typed string literal ([dcl.init.string]), initialization is performed as described in that subclause.

What Are Initializer Lists?

[dcl.init.list]

- Otherwise, if T is an aggregate, aggregate initialization is performed.

What Are Initializer Lists?

[dcl.init.list]

- Otherwise, if the initializer list has no elements and T is a class type with a default constructor, the object is value-initialized.

What Are Initializer Lists?

[dcl.init.list]

- Otherwise, if T is a specialization of std::initializer_list, the object is constructed as described below.

What Are Initializer Lists?

[dcl.init.list]

- Otherwise, if T is a class type, constructors are considered. The applicable constructors are enumerated and the best one is chosen through overload resolution ([over.match], [over.match.list]). If a narrowing conversion (see below) is required to convert any of the arguments, the program is ill-formed.

What Are Initializer Lists?

[dcl.init.list]

- Otherwise, if T is an enumeration with a fixed underlying type ([dcl.enum]), the initializer-list has a single element v, and the initialization is direct-list-initialization, the object is initialized with the value T(v) ([expr.type.conv]); if a narrowing conversion is required to convert v to the underlying type of T, the program is ill-formed.

What Are Initializer Lists?

[dcl.init.list]

- Otherwise, if the initializer list has a single element of type E and either T is not a reference type or its referenced type is reference-related to E, the object or reference is initialized from that element (by copy-initialization for copy-list-initialization, or by direct-initialization for direct-list-initialization); if a narrowing conversion (see below) is required to convert the element to T, the program is ill-formed.

What Are Initializer Lists?

[dcl.init.list]

- Otherwise, if T is a reference type, a prvalue of the type referenced by T is generated. The prvalue initializes its result object by copy-list-initialization or direct-list-initialization, depending on the kind of initialization for the reference. The prvalue is then used to direct-initialize the reference.

What Are Initializer Lists?

[dcl.init.list]

- Otherwise, if the initializer list has no elements, the object is value-initialized.

What Are Initializer Lists?

[dcl.init.list]

- Otherwise, the program is ill-formed.

What Are Initializer Lists?

[dcl.init.list]

And I stripped out all the notes and examples

(I wonder if the committee could learn something from Kate's talks on taking the time to simplify)

So what is an initializer list?

It can be many different things

Bonus Base Class Aggregate Initialization

```cpp
struct Base {
  int i;
};

struct Intermediate : Base {
  double d;
};

struct Derived : Intermediate {
  char c;
};

int main() {
  Derived obj{{{5}, 4.3}, 'c'}; /// also an initializer list
}
```

Reddit: Not Convinced Of Brace Initialization

I've read it is supposed to be better for years. For me, the edge cases seem to outweigh the benefits. I've tried to use it in a few projects now, including full conversions, and I'm pretty sure I'm going to revert back and refuse to use it in future. It causes more confusion than not. There are weird effects when using auto. Weird effects when using it with std. It just adds cognitive overhead, the opposite of the intended goal. It seems to increase the amount of issues with code rather than decrease them. I'm curious if others have the same feeling?

Reddit: Not Convinced Of Brace Initialization

71 comments of agreement

But this is undeniably better. Right?

```
std::vector vec{1,2,3,4,5}; // C++17
```

vs

```
std::vector<int> vec;
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
vec.push_back(4);
vec.push_back(5);
```

What we will be focusing on

The usage of Initializer Lists that results in an `initializer_list<>` object

What does this do?

```
std::vector<int> vec(2,2);
```

Creates a vector of 2 integers of value 2.

What does this do?

```
std::vector<int> vec(3,3);
```

Creates a vector of 3 integers of value 3.

What does this do?

```
std::vector<int> vec{3,3};
```

Creates a vector of 2 integers of value 3.

By calling:

```
std::vector<int>(std::initializer_list<int>)
```

What does this do?

```
std::vector vec{std::make_shared<int>(1), std::make_shared<int>(2)};
```

Vector of 2 `shared_ptr` objects, using C++17’s class template type deduction.

What does this do?

```cpp
std::vector vec{std::make_unique<int>(1), std::make_unique<int>(2)};
```

Fails to compile!

How many `shared_ptr` objects are there?

On this line of code?

```
std::vector vec{std::make_shared<int>(1), std::make_shared<int>(2)};
```

4!

What does this code print? (assume argc=1)

```
#include <initializer_list>
#include <iostream>

auto f(int i, int j, int k ) {
  return std::initializer_list<int>{ i, j, k};
}

int main(int argc, const char *[]) {
  for (int i : f(argc+1, argc+2, argc+3)) {
    std::cout << i << ',';
  }
}
```

Thanks Patrice and Ben for this example.

What does this code print? (assume argc=1)

```
#include <initializer_list>
#include <iostream>

auto f(int i, int j, int k ) {
  return std::initializer_list<int>{ i, j, k};
}

int main(int argc, const char *[]) {
  for (int i : f(argc+1, argc+2, argc+3)) {
    std::cout << i << ',';
  }
}
```

Unknown!

The “Below”

[dcl.init.list]

> *An object of type `std::initializer_list<E>` is constructed from an initializer list as if the implementation generated and materialized a prvalue of type “array of N const E”, where N is the number of elements in the initializer list. Each element of that array is copy-initialized with the corresponding element of the initializer list, and the `std::initializer_list<E>` object is constructed to refer to that array. [Note: A constructor or conversion function selected for the copy shall be accessible in the context of the initializer list. — end note] If a narrowing conversion is required to initialize any of the elements, the program is ill-formed. [Example:*

The “Below”

[dcl.init.list]

```
struct X {
  X(std::initializer_list<double> v);
};
X x{ 1,2,3 };
```

The initialization will be implemented in a way roughly equivalent to this:

```
const double __a[3] = {double{1}, double{2}, double{3}};
X x(std::initializer_list<double>(__a, __a+3));
```

assuming that the implementation can construct an initializer_list object with a pair of pointers. — end example]

Notes

- `const west` vs `east const`
- No narrowing conversions allowed

What is going on?

```
std::vector vec{std::make_unique<int>(1), std::make_unique<int>(2)};
```

This is equiv to:

```
const std::unique_ptr<int> __a[]={std::make_unique<int>(1),
                                  std::make_unique<int>(2)};
std::vector vec(std::initializer_list<std::unique_ptr<int>>(__a, __a + 2));
```

What is going on?

```
std::vector vec{std::make_shared<int>(1), std::make_shared<int>(2)};
```

This is equiv to:

```
const std::shared_ptr<int> __a[]={std::make_shared<int>(1),
                                  std::make_shared<int>(2)};
std::vector vec(std::initializer_list<std::shared_ptr<int>>(__a, __a + 2));
```

What is going on?

```cpp
auto f(int i, int j, int k ) {
  return std::initializer_list<int>{ i, j, k};
}
```

This is equiv to:

```cpp
auto f(int i, int j, int k ) {
  const int __a[] = {i, j, k};
  return std::initializer_list<int>{ __a, __a + 3 }; /// pointer to local
}
```

Initializer Lists Are Broken. Agreed?

Let’s Fix Them!

What is going on?

```
const std::shared_ptr<int> __a[]={std::make_shared<int>(1),
                                  std::make_shared<int>(2)};
std::vector vec(std::initializer_list<std::shared_ptr<int>>(__a, __a + 2));
```

- 1 vector allocation
- 2 constructions
- 2 copy constructors
- 1 vector deallocation
- 4 destructions

What is going on?

```
std::vector<shared_ptr<int>> vec;
vec.emplace_back(std::make_shared<int>(1));
vec.emplace_back(std::make_shared<int>(2));
```

- How many `shared_ptr`s constructed?
- How many `shared_ptr`s copy constructed?
- How many `shared_ptr`s move constructed?
- How many `shared_ptr` destructors?

`emplace_back()` 1: construction

```
std::vector<shared_ptr<int>> vec;
```

- 0 allocations
- 0 shared_ptr operations

`emplace_back()` 2: first `emplace_back`

```
vec.emplace_back(std::make_shared<int>(1));
```

- 1 vector allocation
- 1 construction
- 1 move
- 1 destruction

`emplace_back()` 3: second `emplace_back`

```
vec.emplace_back(std::make_shared<int>(2));
```

- 1 vector reallocation
- 1 construction
- 2 moves
- 2 destructions (moved from objects)

And on scope exit:

- 1 vector deallocation
- 2 destructions

Example

```cpp
#include <cstdio>
#include <vector>
struct S {
  S()                          {puts("S()");}
  ~S()                         {puts("~S()");}
  S(const S &) noexcept  {puts("S(const S &)");}
  S(S &&)      noexcept  {puts("S(S&&)");}
  S&operator=(const S&)noexcept{puts("operator=(const S&)");return *this;}
  S&operator=(S &&) noexcept   {puts("operator(S &&)"); return *this;}
};

int main() {
  std::vector<S> vec;
}
```

And What if We Use `emplace` Correctly?

```cpp
#include <cstdio>
#include <vector>
struct S {
  S()                        {puts("S()");}
  ~S()                       {puts("~S()");}
  S(const S &) noexcept  {puts("S(const S &)");}
  S(S &&)      noexcept  {puts("S(S&&)");}
  S&operator=(const S&)noexcept{puts("operator=(const S&)");return *this;}
  S&operator=(S &&) noexcept   {puts("operator(S &&)"); return *this;}
};

int main() {
  std::vector<S> vec;
}
```

And What if We Use `emplace` Correctly?

But there's no way to do this with `std::make_shared`.

Why are we using `make_shared`?

Because that’s what we’re told to do!

Comparison

`initialzer_list`	`emplace_back`
1 vector allocation	1 vector allocaton
2 constructions	2 constructions
2 copies	3 moves
4 destructions	5 destructions
1 vector deallocation	1 vector deallocation
	1 vector reallocation

How about using reserve?

```
std::vector<shared_ptr<int>> vec;
vec.reserve(2);
vec.emplace_back(std::make_shared<int>(1));
vec.emplace_back(std::make_shared<int>(2));
```

- 1 vector allocation
- 2 constructions
- 2 moves
- 4 destructions
- 1 vector deallocation

How about using an array?

```
std::array<shared_ptr<int>, 2> arr{
  std::make_shared<int>(1),
  std::make_shared<int>(2)
};
```

- 2 constructions
- 2 destructions

How about using an array?

Why is the array this much better?

Aggregate Initialization

`std::array<>` looks something like this:

```
template<typename T, std::size_t Size>
struct array
{
  T data[Size];
};
```

Remember The Aggregate Initialization?

```cpp
struct Base {
  int i;
};

struct Intermediate : Base {
  double d;
};

struct Derived : Intermediate {
  char c;
};

int main() {
  Derived obj{{{5}, 4.3}, 'c'}; /// also an initializer list
}
```

Aggregate Initialization

`std::array<>` looks something like this:

```
template<typename T, std::size_t Size>
struct array
{
  T data[Size];
};
```

So construction of an `std::array` object is directly initializing the data.

This is our idealized goal.

Movable `initializer_list`

First question is, since the `initializer_list<>` can only be accessed from one location, why is it not a `move_iterator` pair?

```
std::shared_ptr<int> __a[] = { std::make_shared<int>(1),
                               std::make_shared<int>(2) };
std::vector vec(
  std::initializer_list<std::shared_ptr<int>>(
    std::make_move_iterator(std::begin(_a)),
    std::make_move_iterator(std::end(_a))
  )
);
```

1. `std::initializer_list` only has `const` accessors, but that could be changed.
2. The definition of the array created for us is `const`.

Movable `initializer_list`

We can emulate the concept of a moveable `initializer_list<>` manually:

```
std::shared_ptr<int> __a[] = { std::make_shared<int>(1),
                               std::make_shared<int>(2) };
std::vector vec(std::make_move_iterator(std::begin(_a)),
                std::make_move_iterator(std::end(_a)));
```

- 1 vector allocation
- 2 constructions
- 2 moves
- 4 destructions
- 1 vector deallocation

Or Provide a Variadic Constructor for vector

```cpp
#include <vector>
#include <cstdio>

template<typename T>
struct Better_Vector : private std::vector<T> {
    using std::vector<T>::emplace_back;
    using std::vector<T>::operator[];
    using std::vector<T>::reserve;
    using std::vector<T>::size;

    template<typename ... Param>
    explicit Better_Vector(Param && ... param) {
        reserve(sizeof...(param)); ///
        (emplace_back(std::forward<Param>(param)), ...);
    }
};

int main() {
    Better_Vector<S> vec{S{}, S{}};
}
```

Or Provide a Variadic Constructor for vector

```cpp
#include <vector>
#include <cstdio>

template<typename T>
struct Better_Vector : private std::vector<T> {
    using std::vector<T>::emplace_back;
    using std::vector<T>::operator[];
    using std::vector<T>::reserve;
    using std::vector<T>::size;

    template<typename ... Param>
    explicit Better_Vector(Param && ... param) {
        reserve(sizeof...(param));
        (emplace_back(std::forward<Param>(param)), ...); ///
    }
};

int main() {
    Better_Vector<S> vec{S{}, S{}};
}
```

Or Provide a Variadic Constructor for vector

The variadic constructor gets us to the second-best state of

- 1 vector allocation
- 2 constructions
- 2 moves
- 4 destructions
- 1 vector deallocation

We Have A Trade Off To Make

- Move the constructor parameters?
- Move the constructed object?

Moving the Parameters

```cpp
#include <vector>
#include <tuple>

template<typename T>
struct Better_Vector : private std::vector<T> {
  using std::vector<T>::push_back;
  using std::vector<T>::emplace_back;
  using std::vector<T>::reserve;

  template <class Tuple, std::size_t... I>
  void emplace_from_tuple_impl( Tuple&& t, std::index_sequence<I...> ) {
    // optional if-constexpr to push_back if types match
    emplace_back(std::get<I>(std::forward<Tuple>(t))...);
  }
  template <class Tuple> void emplace_from_tuple(Tuple&& t) {
    emplace_from_tuple_impl(std::forward<Tuple>(t),
        std::make_index_sequence<std::tuple_size_v<
           std::remove_reference_t<Tuple>>>{});
  }
  template<typename ... Param> explicit Better_Vector(Param && ...param) {
    reserve(sizeof...(param)); /// 1
    (emplace_from_tuple(std::forward<Param>(param)), ... );
  }
};
```

```cpp
#include <vector>
#include <tuple>

template<typename T>
struct Better_Vector : private std::vector<T> {
  using std::vector<T>::push_back;
  using std::vector<T>::emplace_back;
  using std::vector<T>::reserve;

  template <class Tuple, std::size_t... I>
  void emplace_from_tuple_impl( Tuple&& t, std::index_sequence<I...> ) {
    // optional if-constexpr to push_back if types match
    emplace_back(std::get<I>(std::forward<Tuple>(t))...);
  }
  template <class Tuple> void emplace_from_tuple(Tuple&& t) {
    emplace_from_tuple_impl(std::forward<Tuple>(t), /// 2
        std::make_index_sequence<std::tuple_size_v<
           std::remove_reference_t<Tuple>>>{});
  }
  template<typename ... Param> explicit Better_Vector(Param && ...param) {
    reserve(sizeof...(param)); // 1
    (emplace_from_tuple(std::forward<Param>(param)), ... );
  }
};
```

```cpp
#include <vector>
#include <tuple>

template<typename T>
struct Better_Vector : private std::vector<T> {
  using std::vector<T>::push_back;
  using std::vector<T>::emplace_back;
  using std::vector<T>::reserve;

  template <class Tuple, std::size_t... I>
  void emplace_from_tuple_impl( Tuple&& t, std::index_sequence<I...> ) {
    // optional if-constexpr to push_back if types match
    emplace_back(std::get<I>(std::forward<Tuple>(t))...); /// 3
  }
  template <class Tuple> void emplace_from_tuple(Tuple&& t) {
    emplace_from_tuple_impl(std::forward<Tuple>(t), //  2
        std::make_index_sequence<std::tuple_size_v<
           std::remove_reference_t<Tuple>>>{});
  }
  template<typename ... Param> explicit Better_Vector(Param && ...param) {
    reserve(sizeof...(param)); // 1
    (emplace_from_tuple(std::forward<Param>(param)), ... );
  }
};
```

Moving the Parameters - Usage

```
int main() {
  Better_Vector<S> vec{std::forward_as_tuple(param), // one param
                       std::forward_as_tuple()};     // zero params
}
```

Moving the Parameters

If we move the parameters, and all the parameters are literal types (or no parameters are passed), we can get:

- 1 vector allocation
- 2 constructions
- 2 destructions
- 1 vector deallocation

Performance Comparisons

`initializer_list<>`

Our baseline comparison.

```cpp
std::vector<DesiredType> vec{get_value(), get_value()};
```

`const std::array<>`

Our control comparison.

```
const std::array a{get_value(), get_value()};
std::vector<DesiredType> vec{std::begin(a),
                             std::end(a)};
```

`std::array<>`

First attempt at better: no copies.

```
std::array a{get_value(), get_value()};
std::vector<DesiredType> vec{std::make_move_iterator(std::begin(a)),
                             std::make_move_iterator(std::end(a))};
```

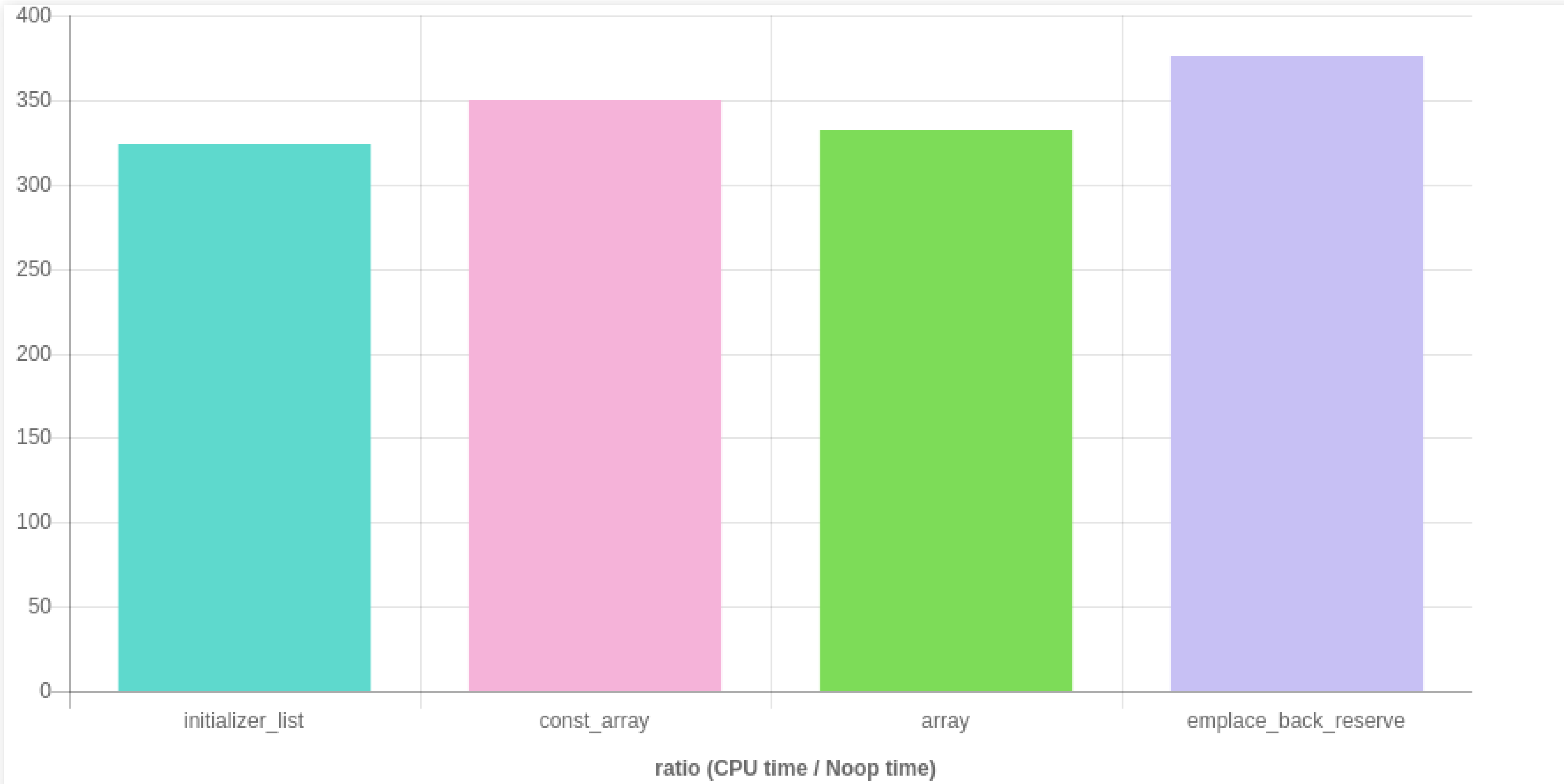
`emplace_back()` with `reserve()`

Second attempt at better: no copies, better API.

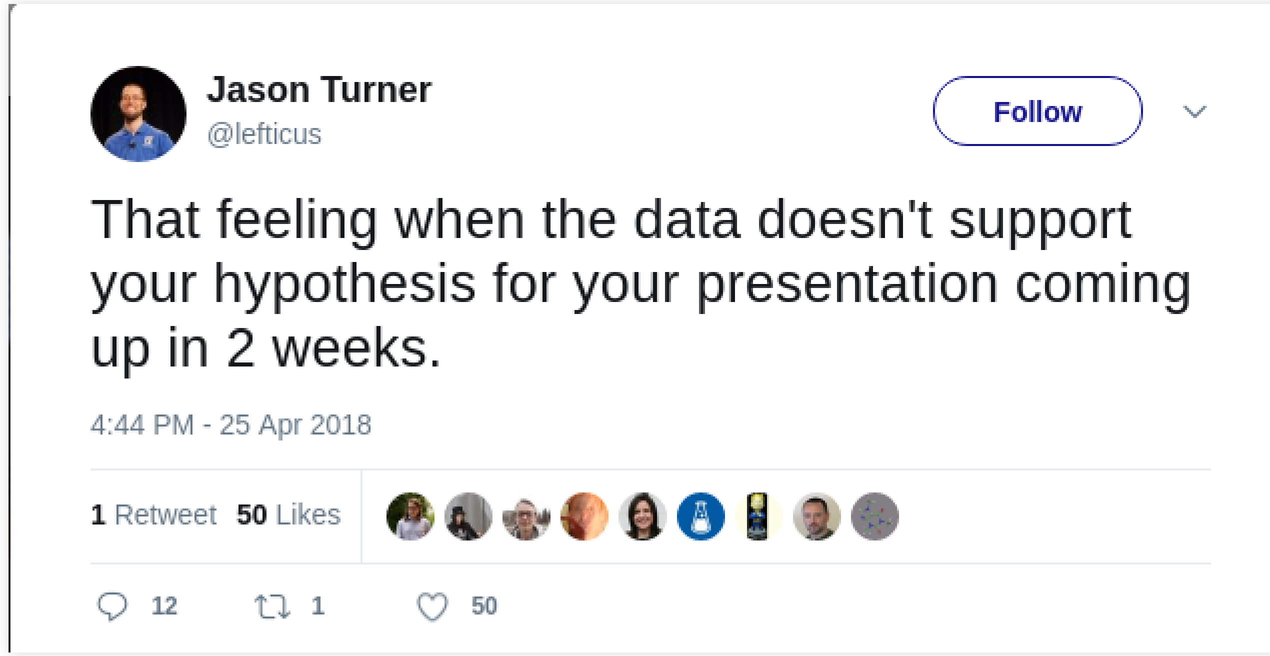
```cpp
#include <vector>
#include <cstdio>

template<typename T>
struct Better_Vector : private std::vector<T> {
    using std::vector<T>::emplace_back;
    using std::vector<T>::reserve;

    template<typename ... Param>
    explicit Better_Vector(Param && ... param) {
        reserve(sizeof...(param));
        (emplace_back(std::forward<Param>(param)), ...);
    }
};

int main() {
    Better_Vector<DesiredType> vec{get_value(), get_value()};
}
```

clang 6.0 with libc++ 10 Elements

From quick-bench.com



clang 6.0 with libc++ 10 Elements



Note that the small differences can be due to differences in measurement

clang 6.0 with libc++

Any guesses as to what caused the worse performance with clang?

```
#include <vector>
#include <string>
#include <array>

constexpr auto get_value = []{ return "Hello World"; };

#ifdef OPT1
void move_from_array() {
    std::array<decltype(get_value()), 1> a {get_value()};
    std::vector<std::string> v{std::make_move_iterator(begin(a)),
                               std::make_move_iterator(end(a))};
}
#else
void init_list() {
    std::vector<std::string> v{get_value()};
}
#endif
```

clang 6.0 with libc++

What’s the return type?

```
#include <vector>
#include <string>
#include <array>

constexpr auto get_value = []{ return "Hello World"; }; ///

#ifdef OPT1
void move_from_array() {
    std::array<decltype(get_value()), 1> a {get_value()};
    std::vector<std::string> v{std::make_move_iterator(begin(a)),
                               std::make_move_iterator(end(a))};
}
#else
void init_list() {
    std::vector<std::string> v{get_value()};
}
#endif
```

clang 6.0 with libc++

What’s the type of the array?

```
#include <vector>
#include <string>
#include <array>

constexpr auto get_value = []{ return "Hello World"; };

#ifdef OPT1
void move_from_array() {
    std::array<decltype(get_value()), 1> a {get_value()}; ///
    std::vector<std::string> v{std::make_move_iterator(begin(a)),
                               std::make_move_iterator(end(a))};
}
#else
void init_list() {
    std::vector<std::string> v{get_value()};
}
#endif
```

clang 6.0 with libc++

What's the type of the `initializer_list`?

```cpp
#include <vector>
#include <string>
#include <array>

constexpr auto get_value = []{ return "Hello World"; };

#ifdef OPT1
void move_from_array() {
    std::array<decltype(get_value()), 1> a {get_value()};
    std::vector<std::string> v{std::make_move_iterator(begin(a)),
                               std::make_move_iterator(end(a))};
}
#else
void init_list() {
    std::vector<std::string> v{get_value()}; ///
}
#endif
```

clang 6.0 with libc++

Compare to:

```
#include <vector>
#include <string>
#include <array>

constexpr auto get_value = []{ return "Hello World"; };

#ifdef OPT1
void move_from_array() {
    std::array<std::string, 1> a {get_value()}; ///
    std::vector<std::string> v{std::make_move_iterator(begin(a)),
                               std::make_move_iterator(end(a))};
}
#else
void init_list() {
    std::vector<std::string> v{get_value()};
}
#endif
```

Second Round of Comparisons

std::array<DesiredType>

`const` array of desired type

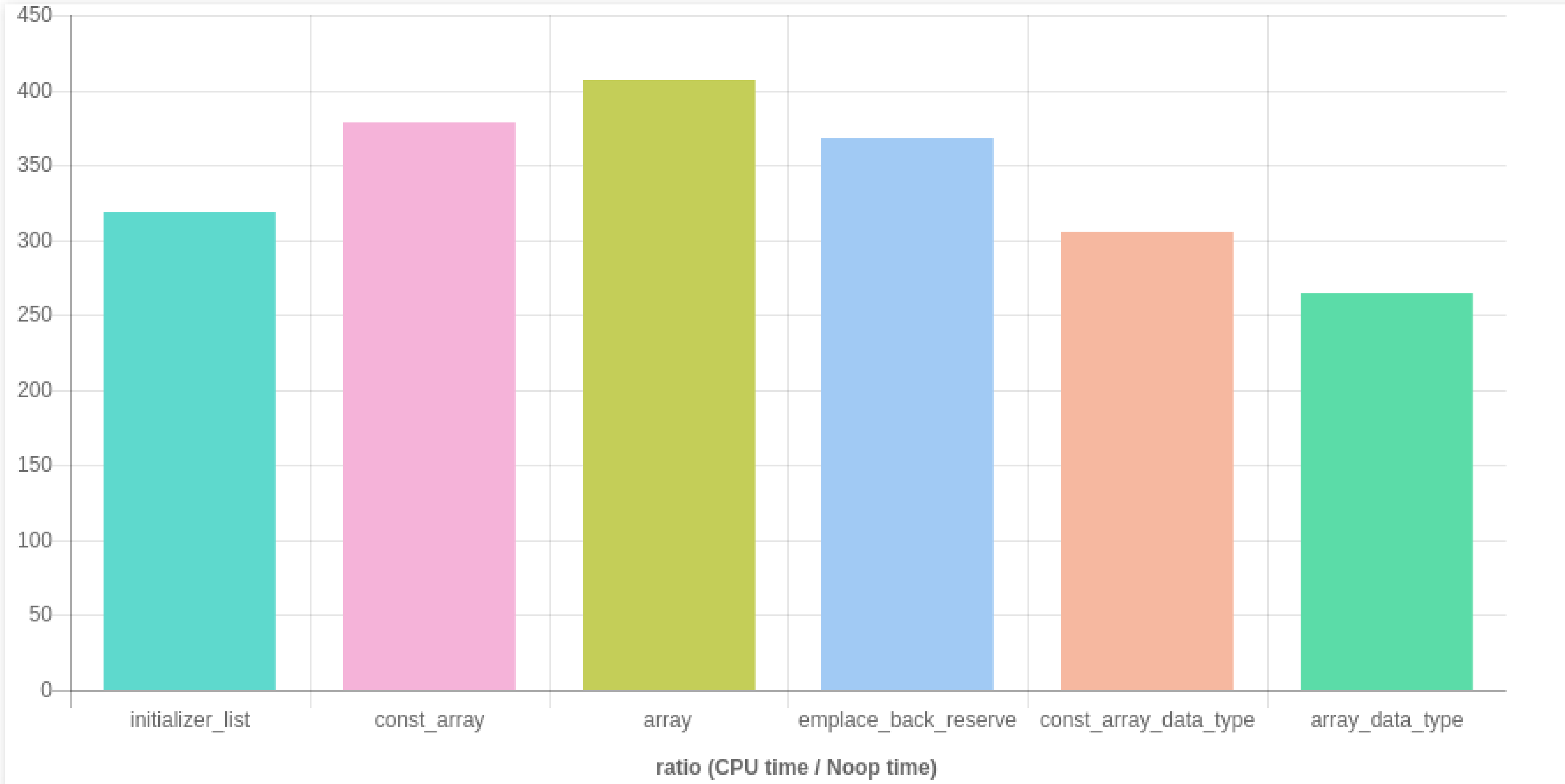
```
const std::array<DesiredType, 2> a{get_value(), get_value()};
std::vector<DesiredType> vec{std::begin(a), std::end(a)};
```

`std::array<DesiredType>`

Non-`const` array of desired type

```
std::array<DesiredType, 2> a{get_value(), get_value()};
std::vector<DesiredType> vec{std::make_move_iterator(std::begin(a)),
                             std::make_move_iterator(std::end(a))};
```

clang 6.0 with libc++ 10 Elements



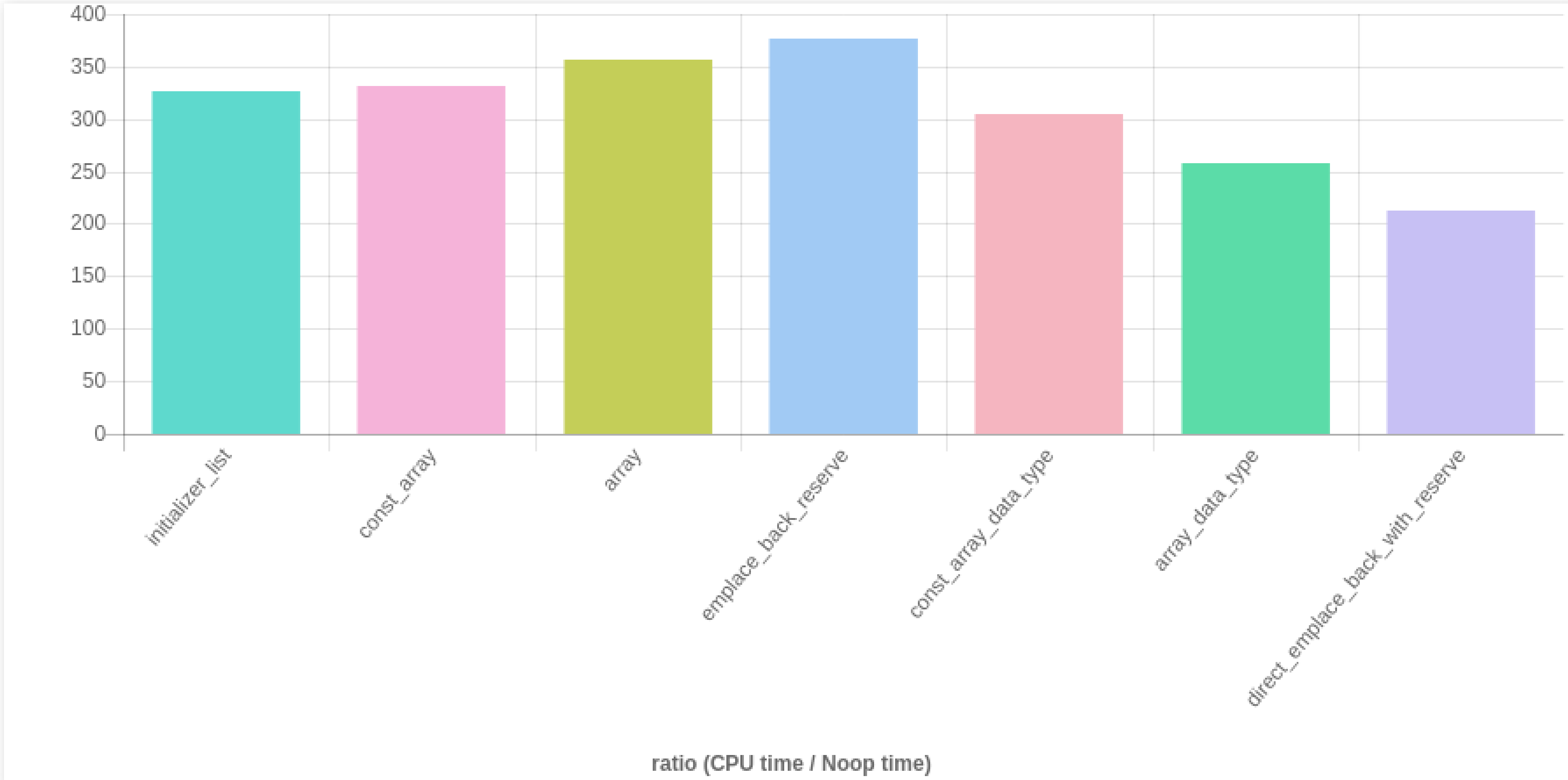
Avoiding The Layer of Indirection

The `emplace_back_reserve` example required going through a constructor with “perfect forwarding.” What happens if we avoid that perfect forwarding?

```
std::vector<DataType> data;
data.reserve(2);
data.emplace_back(get_value());
data.emplace_back(get_value());
```

```
constexpr auto get_value = []{ return "Hello World"; };
```

clang 6.0 with libc++ 10 Elements



Performance Wrap Up

Performance Wrap Up

I ran 16 tests:

- strings that fit in SSO vs strings that do not
- returning of `const char *` vs `std::string`
- clang trunk with libc++ vs gcc trunk
- 5 parameters vs 10 parameters

The moveable temporary array beat `initializer_list` in every case (but was not always the overall winner).

```
std::array<std::string, 1> a {get_value()}; ///
std::vector<std::string> v{std::make_move_iterator(begin(a)),
                           std::make_move_iterator(end(a))};
```

Small Strings Were Messing With My Performance Tests

Small Strings Are Toying With Me

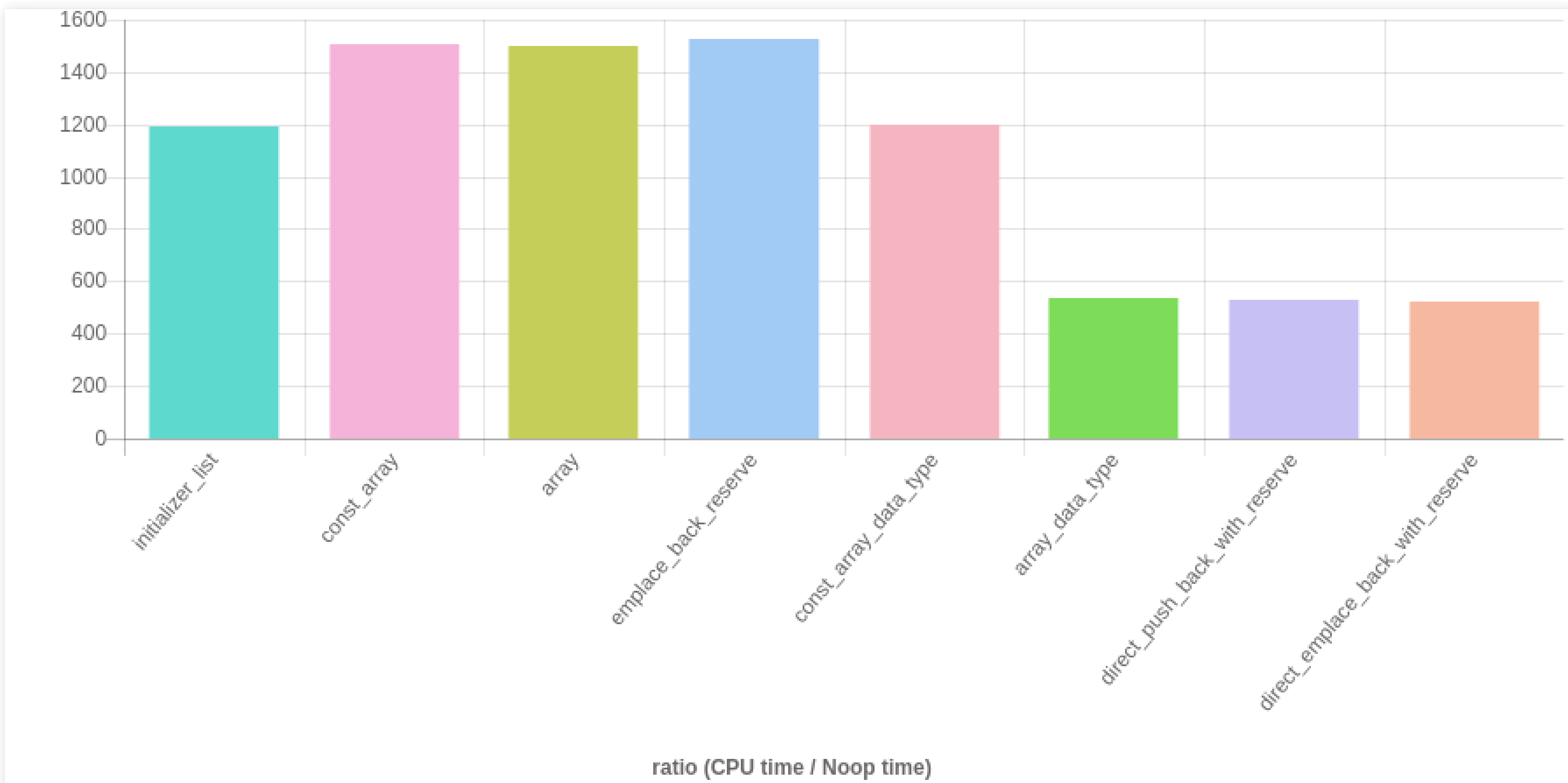
If it's a small string

- Can the compiler tell that at compile time?
- Has the string literal decayed to a `const char *` and been passed around too many times for the optimizer to know the length when the string is constructed?

Small Strings: non-trivially copyable, non-trivially moveable - but very fast to copy or move.

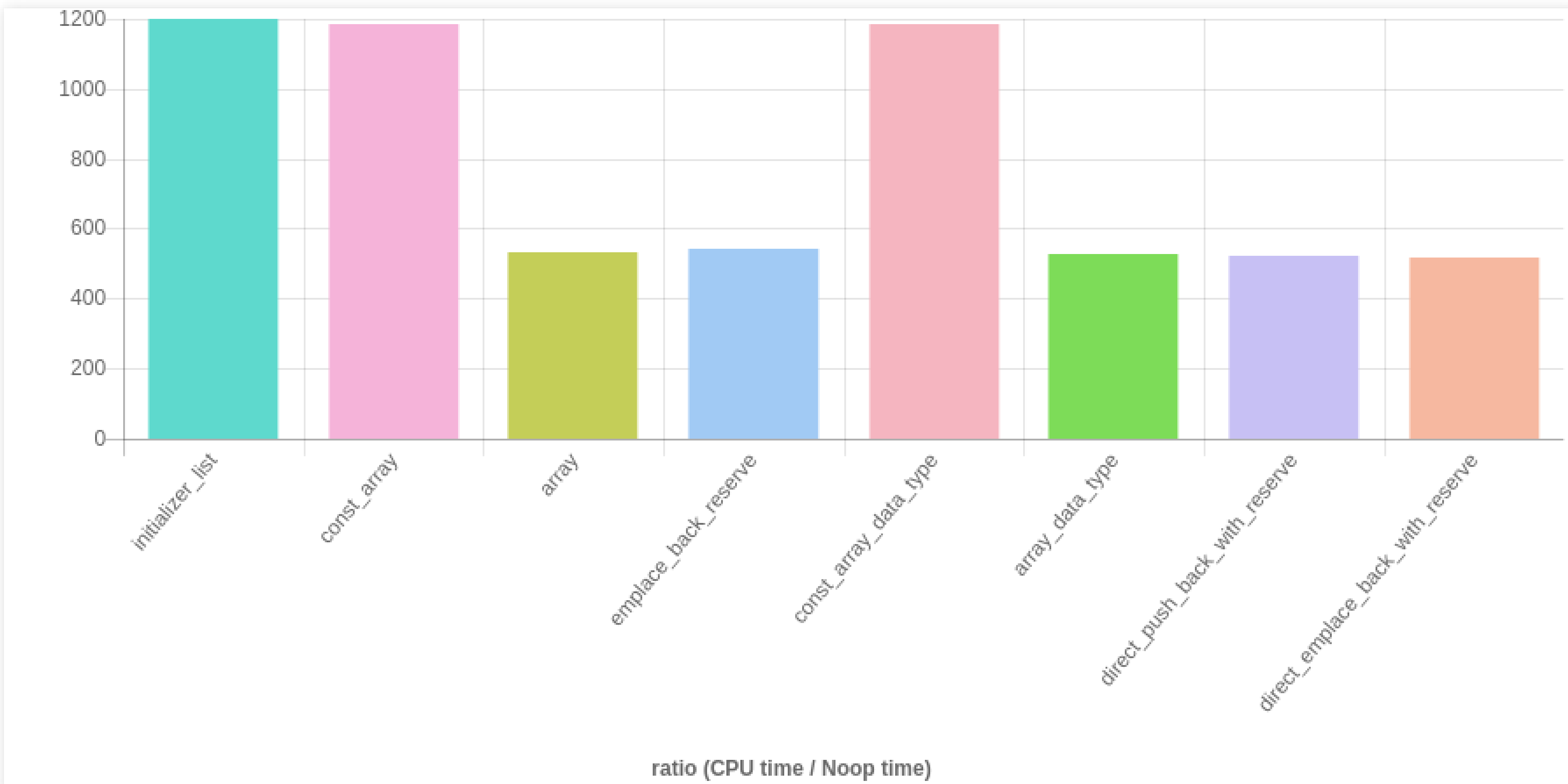
Non-SSO Strings

Things become much more obvious here, clang 6.0 / libc++ using this creator function: `[](const std::size_t) {return "Hello World Long String";};`



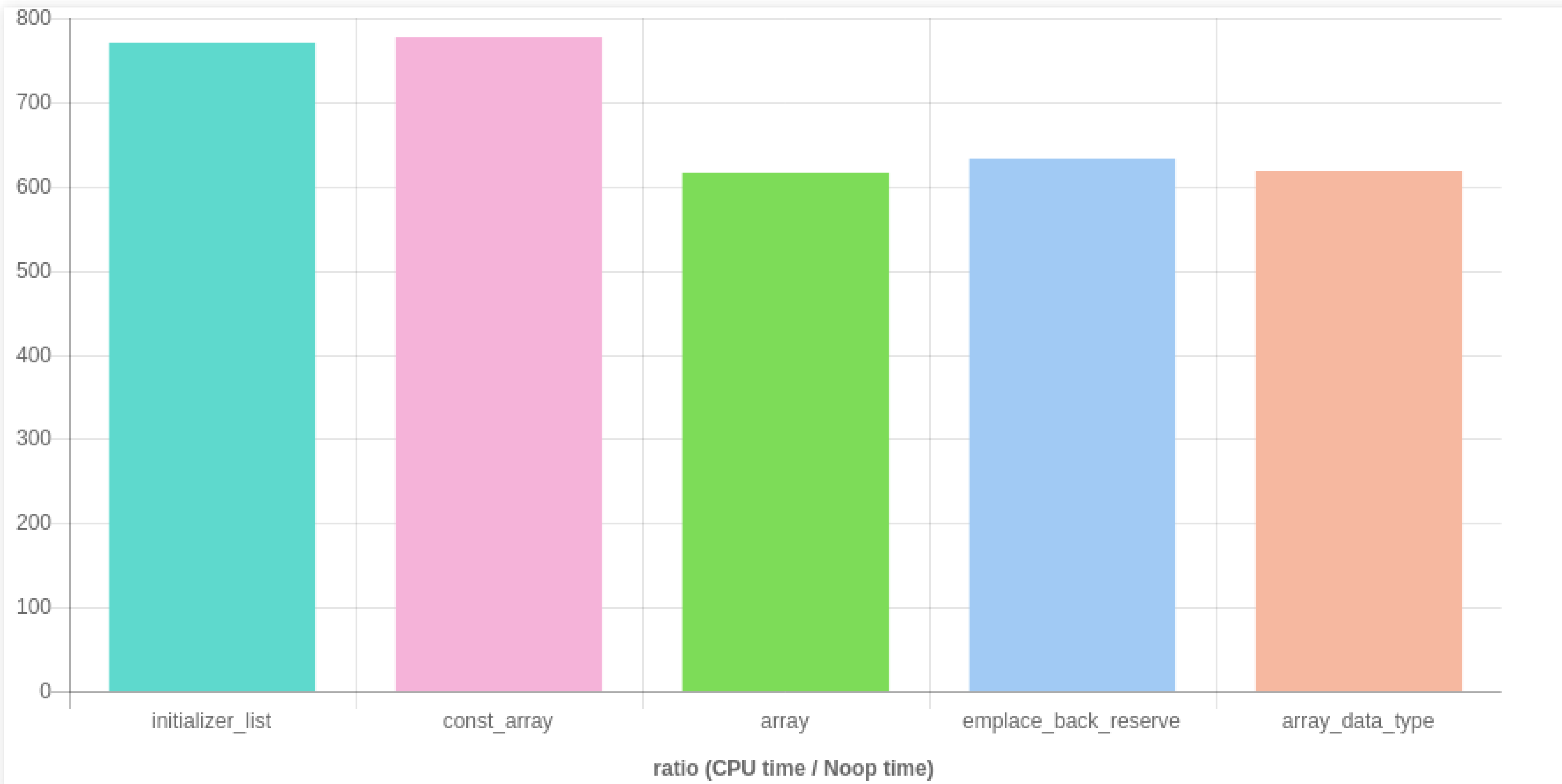
Non-SSO Strings

And for creating strings from `std::string` return value, clang 6.0 / libc++ using this creator function: `[](const std::size_t) -> std::string {return "Hello World Long String";};`



`shared_ptr`

`shared_ptr` is always expensive to copy, clang 6.0 / libc++ using this creator function: `[](const std::size_t val) {return std::make_shared<std::size_t>(val);};`



Avoiding Constructor Ambiguity

Simon Brand’s Proposal

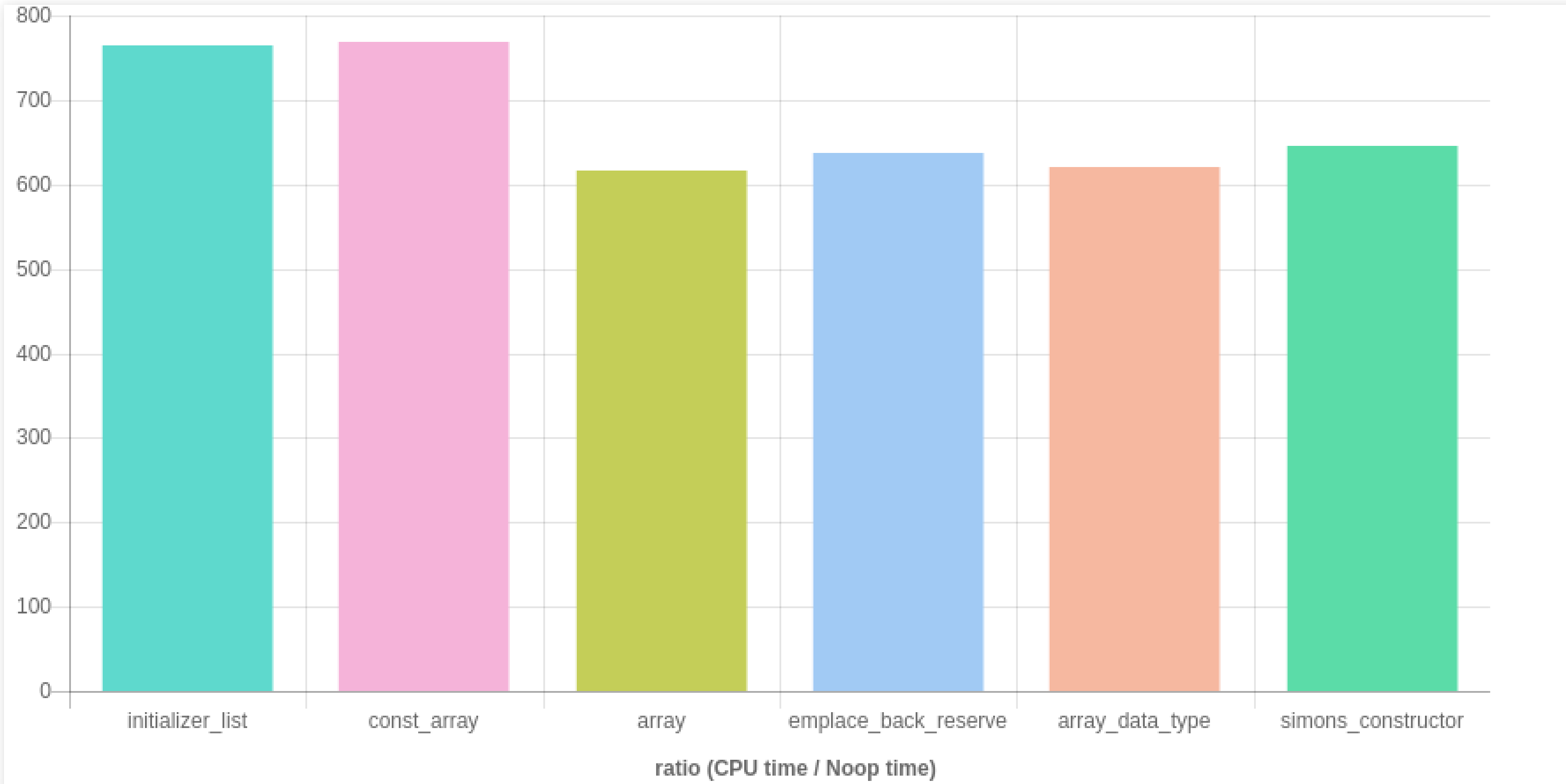
https://wg21.tartanllama.xyz/initializer_list/

Simon is proposing that containers get a new `std::in_place_t` constructor, which is consistent with `std::optional<>`, `std::any`, and `std::variant<>`

```cpp
template<typename T>
struct Simons_Vector : private std::vector<T> {
  using std::vector<T>::emplace_back;
  using std::vector<T>::reserve;

  template<typename ... Param>
  explicit Simons_Vector(std::in_place_t, Param && ... param) {
    reserve(sizeof...(param));
    (emplace_back(std::forward<Param>(param)), ...);
  }
};
```

Simon Brand's Proposal

This compares favorably with other options.



Alternative Directly Taking An `std::array`

```
#include <vector>
#include <array>
#include <iterator>

template<typename T>
struct Better_Vector : private std::vector<T> {
  template<std::size_t Size>
  Better_Vector(std::array<T, Size> &&vals)
    : std::vector<T>{std::make_move_iterator(std::begin(vals)),
                     std::make_move_iterator(std::end(vals))}
  {
  }
};

int main() {
  Better_Vector bv{std::array{1,2,3}}; /// C++17 yay!
}
```

BTW, `std::array` is *the best use* of class template type deduction from C++17!

Conclusion

So what do we do?

1. Prefer literal types that are trivially destructible and trivially moveable
2. Always try to avoid reallocations with your container
3. If we want to keep `initializer_list` around, we should add a new type: `movable_initializer_list` that containers can support and is non-`const`
4. Consider applying Simon Brand's proposal on your own containers: https://wg21.tartanllama.xyz/initializer_list/ - make `std::in_place_t` part of your vocabulary?
5. Check out http://quick-bench.com/8PyCGCEi6jGnYE1qC7FcbmgptBY
6. Results can be a bit flaky on quick-bench.com. Do not compare too many tests at once, run locally if you need harder numbers.
7. Make `std::in_place_t` part of you vocabulary

Additional Thoughts

Can we expose the internals of our containers?

Hypothetical Exposed Internals

```
std::unique_ptr<DataType []> ptr{
  new DataType[sizeof...(I)]{ creator(I) ... }
};
```

- We have erased the size from the type
- Maintained the type info
- Gained aggregate initializaton of the data

I only had moderate success going down this road. (But I think there’s more here.)

Perfect Forwarding That Isn’t

Surprise Lesson Learned (clang)

```cpp
#include <string>
#include <array>

std::string get_string() {
  return "Hello World";
}
auto get_char_string() {
  return "Hello World";
}
template<typename ... T>
std::array<std::string, sizeof...(T)> forward_strings(T && ... t) {
  return {std::forward<T>(t)...};
}

int main() {
  const auto value=forward_strings(get_char_string(),get_char_string());
//  const auto value=forward_strings(get_string(), get_string());
}
```

Surprise Lesson Learned

I’ve maintained this practice for a while:

> *Always delay dynamic allocations and type erasure as late as possible.*

But string literals and C’s legacy mess with our mental model.

Types of string literals

As seen in the last example, the compiler may or may not be able to trace the length of the string to know if it can fit the string literal into small string.

So for small strings, your best bet *might* be to immediately put it into an `std::string`. Or some sort of strongly typed `std::array<>` like wrapper around your string literal.

Types of string literals

What is the type of x?

```
const auto x = "Hello World";
```

`const char *`

Types of string literals

What is the type of y?

```
const char y[] = "Hello World";
```

`const char []`

Types of string literals

What is the return type?

```
auto get_string() {
  const char y[] = "Hello World";
  return y;
}
```

```
const char *
```

Strongly Typed String Literals

```cpp
#include <string>
#include <string_view>
#include <array>

template<typename Char, std::size_t Len>
struct string_literal {
  constexpr string_literal(const Char (&array)[Len]) noexcept
    : data{array}  { }

  explicit operator std::basic_string<Char>() const {
    return std::basic_string<Char>{data, data + Len - 1};
  }
  const Char *data;
};

const char * get_char_string() { return "Hello World"; }
std::string get_string() { return "Hello World"; }
std::string_view get_string_view() { return "Hello World"; }
auto get_string_literal(){return string_literal{"Hello World"};}

template<typename ... T>
std::array<std::string, sizeof...(T)> forward_strings(T && ... t) {
    return { std::string{std::forward<T>(t)}...};
}

int main() {
//    const auto value =
forward_strings(get_char_string(),get_char_string());
```

```
//    const auto value = forward_strings(get_string(), get_string());
//    const auto value =
forward_strings(get_string_view(),get_string_view());
//    const auto value =
forward_strings(get_string_literal(),get_string_literal());
}
```

Conclusion 2

So what do we do?

- We need strongly typed string literals in C++
- We should probably just deprecate `std::initializer_list<>` or limit it to only literal types
- We can solve the performance issues to some extent, but cannot solve the lifetime issues due to guaranteed copy elision in C++17

Final Thoughts

But didn't you say something about `constexpr` recently?

```
#include <string>
#include <string_view>
#include <array>

template<typename Char, std::size_t Len>
struct string_literal {
  constexpr string_literal(const Char (&array)[Len]) noexcept
    : data{array}  { }

  explicit operator std::basic_string<Char>() const {
    return std::basic_string<Char>{data, data + Len - 1};
  }
  const Char *data;
};

constexpr const char * get_char_string() { return "Hello World"; }
std::string get_string() { return "Hello World"; }
constexpr std::string_view get_string_view(){ return "Hello World"; }
constexpr auto get_string_literal(){return string_literal{"Hello World"};}

template<typename ... T>
std::array<std::string, sizeof...(T)> forward_strings(T && ... t) {
    return { std::string{std::forward<T>(t)}...};
}

int main() {
//    const auto value =
forward_strings(get_char_string(),get_char_string());
```

```
//    const auto value = forward_strings(get_string(), get_string());
//    const auto value =
forward_strings(get_string_view(),get_string_view());
//    const auto value =
forward_strings(get_string_literal(),get_string_literal());
}
```

This Stuff Can Be Super Finicky (clang)

```
#include <string>
#include <array>
namespace { ///
  std::string get_string() {
    return "Hello World";
  }
  auto get_char_string() {
    return "Hello World";
  }
  template<typename ... T>
  std::array<std::string, sizeof...(T)> forward_strings(T && ... t) {
    return {std::forward<T>(t)...};
  }
}

int main() {
  const auto value=forward_strings(get_char_string(),get_char_string());
//  const auto value=forward_strings(get_string(), get_string());
}
```

Conclusion 3

`constexpr`

- Is not a magic bullet
- Stronger typing is better than weaker typing
- Besides my strongly typed `string_literal`, what about a fixed length `std::span` so we only drop as much information as necessary?

Jason Turner

- First used C++ in 1996, professionally since 2002
- Co-host of CppCast http://cppcast.com
- Host of C++ Weekly https://www.youtube.com/c/JasonTurner-lefticus
- Co-creator of ChaiScript http://chaiscript.com
- Curator of http://cppbestpractices.com
- Microsoft MVP for C++ 2015-present

Jason Turner

Independent and available for training or contracting

- http://articles.emptycrate.com/idocpp
- http://articles.emptycrate.com/training.html