EVERYDAY EFFICIENCY: IN-PLACE CONSTRUCTION

(BACK TO BASICS?)

> *"Make no collection of it: let him show*
> *His skill is in the construction."*

-- William Shakespeare, Cymbeline

BEN DEANE / @ben_deane

SEPTEMBER 19 2019

GET READY

1. What happens when we `move` something?
2. Copy elision (RVO)
3. Putting stuff into a `vector`
4. `initializer_list`
5. Putting stuff into a `map`
6. Putting stuff into other things
7. Final thoughts

PRELIMINARIES

```
#include <cstdio>

struct Arg {};

struct S
{
  S() { puts("Default construct"); }
  S(Arg) { puts("Value construct"); }
  explicit S(int) { puts("Explicit value construct (1)"); }
  explicit S(int, int) { puts("Explicit value construct (2)");}
  ~S() { puts("Destruct"); }
  S(const S&) { puts("Copy construct"); }
  S(S&&) { puts("Move construct"); }
  S& operator=(const S&) { puts("Copy assign"); return *this; }
  S& operator=(S&&) { puts("Move assign"); return *this; }
};

int main()
{
    S s;
}
```

1. WHAT HAPPENS WHEN WE `move` SOMETHING?

> *"Mov'd! In good time! Let him that mov'd you hither remove you hence."*

-- William Shakespeare, The Taming of the Shrew

Are moves cheap or not?

OBSERVATION

Moving from a `string` *usually isn't any faster* than copying from it.

(If you doubt this, ask yourself why the small string optimization exists in the first place.)

Moves *only* matter for objects on the heap.

http://quick-bench.com/eb54Wv8Bmvr08frpgtqFOIxQqa4

WHY IS RVO SO IMPORTANT?

```
using PhoneBook = std::map<std::string, int>;

PhoneBook build_phonebook()
{
  PhoneBook pb;
  pb.insert(std::make_pair("Jenny", 8675309));
  return pb;
}
```

Because *moves aren't necessarily cheap.*

WHY IS RVO SO IMPORTANT?

Billy O'Neal
@MalwareMinigun

Replying to @MalwareMinigun @lefticus and 3 others

Move ctor needs to allocate new sentinel node for the *source* container, because the moved from container must still be a container.
Move assignment can destroy all the elements in the destination, then swap.
No change in # of alive containers == no alloc needed == noexcept

11:10 AM · Aug 24, 2019 · Twitter Web App

2. COPY ELISION

> *"If you will, lead these graces to the grave*
> *And leave the world no copy."*

-- William Shakespeare, Twelfth Night, or What You Will

COPY ELISION, AKA RVO

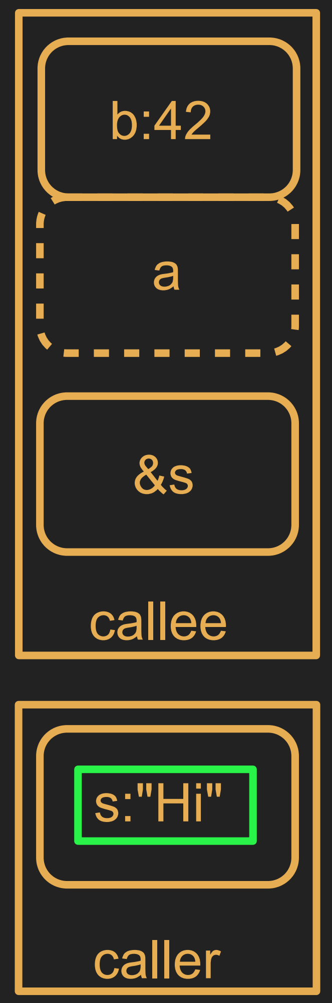
Q. What is the Return Value Optimization?

A. Perhaps the most important optimization the compiler does.

[class.copy.elision]

RVO IN PICTURES

```
string callee()
{
  string a{"Hi"};
  int b = 42;
  // ...
  return a;
}
```




```
void caller()
{
  string s{callee()};
}
```

WHEN CAN RVO NOT APPLY?

RVO Rules: what is returned has to be either:

- a temporary (prvalue) - guaranteed in C++17
- the name of a stack variable

WHEN CAN RVO NOT APPLY?

RVO Ability: sometimes, the callee *can't* construct the object in-place.

- if there is no opportunity to
- if it's not of the right type
- if the callee doesn't know enough

NO RVO BECAUSE NO OPPORTUNITY

You can't RVO a variable if you didn't get the chance to construct it in the first place.

```
std::string sad_function(std::string s)
{
  s += "No RVO for you!";
  return s;
}
```

But the compiler will still move it. (Since C++11)

NO RVO BECAUSE WRONG TYPE

An rvalue-ref is not the same type.

```
std::string sad_function()
{
  std::string s = "No RVO for you!";
  return std::move(s);
}
```

Don't `return std::move(x)` in most cases - you will get a move when you didn't need anything!

NO RVO BECAUSE NOT ENOUGH INFO

It has to be decidable at construction time.

```
std::string undecided_function()
{
  std::string happy = "Hooray";
  std::string sad = "Boo hoo";

  if (getHappiness() > 0.5)
    return happy;
  else
    return sad;
}
```

Again, return value will still be moved.

QUIZ TIME

Wake up!

And tell me if the upcoming code snippets will activate RVO.

WILL IT RVO?

```
const S will_it_rvo()
{
  return S{1};
}
```

WILL IT RVO?

```
const S will_it_rvo()
{
  return S{1};
}
```

Yes.

WILL IT RVO?

```
S will_it_rvo(bool b)
{
  if (b)
    return S{1};
  else
    return S{0};
}
```

WILL IT RVO?

```
S will_it_rvo(bool b)
{
  if (b)
    return S{1};
  else
    return S{0};
}
```

Yes. Even in debug builds.

WILL IT RVO?

```
S will_it_rvo(bool b, S s)
{
  if (b)
    s = S{1};
  return s;
}
```

WILL IT RVO?

```
S will_it_rvo(bool b, S s)
{
  if (b)
    s = S{1};
  return s;
}
```

No (no opportunity).

WILL IT RVO?

```
S get_S() { return S{1}; }

S will_it_rvo(bool b)
{
  if (b)
    return get_S();
  return S{0};
}
```

WILL IT RVO?

```
S get_S() { return S{1}; }

S will_it_rvo(bool b)
{
  if (b)
    return get_S();
  return S{0};
}
```

Yes (can elide multiple copies).

WILL IT RVO?

```
S will_it_rvo(bool b)
{
  if (b)
  {
    S s{1};
    return s;
  }
  return S{0};
}
```

WILL IT RVO?

```
S will_it_rvo(bool b)
{
  if (b)
  {
    S s{1};
    return s;
  }
  return S{0};
}
```

Yes (Clang), no (MSVC/GCC).

WILL IT RVO?

```
S will_it_rvo(bool b)
{
  S s{1};
  if (b)
    return s;
  return S{0};
}
```

WILL IT RVO?

```
S will_it_rvo(bool b)
{
  S s{1};
  if (b)
    return s;
  return S{0};
}
```

No. Possibly in future?

WILL IT RVO?

```
S will_it_rvo(bool b)
{
  S s{1};
  return b ? s : S{0};
}
```

WILL IT RVO?

```
S will_it_rvo(bool b)
{
  S s{1};
  return b ? s : S{0};
}
```

No. (Against the rules - not "the name of a stack variable".)

WILL IT RVO?

```
S get_S() { return S{1}; }

S will_it_rvo(bool b)
{
  return b ? get_S() : S{0};
}
```

WILL IT RVO?

```
S get_S() { return S{1}; }

S will_it_rvo(bool b)
{
  return b ? get_S() : S{0};
}
```

Yes. (Returning temporary.)

WILL IT RVO?

```
S will_it_rvo()
{
  S s{1};
  s = S{2};
  return s;
}
```

WILL IT RVO?

```
S will_it_rvo()
{
  S s{1};
  s = S{2};
  return s;
}
```

Yes.

WILL IT RVO?

```
S will_it_rvo()
{
  S s{1};
  return (s);
}
```

WILL IT RVO?

```
S will_it_rvo()
{
  S s{1};
  return (s);
}
```

Yes (Clang/MSVC), no (GCC).

WILL IT RVO?

```
S will_it_rvo()
{
  S s{1};
  return (s);
}
```

Yes (Clang/MSVC), no (GCC).

WILL IT RVO?

```
S will_it_rvo()
{
  S s{1};
  return (s);
}
```

Yes (Clang/MSVC), no (GCC).

class.copy.elision § 3.1

FINALLY, WHAT'S THE RETURN VALUE?

```
struct P {
  constexpr P() : x{0} {}
  constexpr P(P&&) : x{1} {}
  int x;
};

constexpr auto will_this_rvo() {
  P p;
  return p;
}

int main() {
  const auto p = will_this_rvo();
  return p.x;
}
```

FINALLY, WHAT'S THE RETURN VALUE?

```
struct P {
  constexpr P() : x{0} {}
  constexpr P(P&&) : x{1} {}
  int x;
};

constexpr auto will_this_rvo() {
  P p;
  return p;
}

int main() {
  const auto p = will_this_rvo();
  return p.x;
}
```

It depends...

EXHIBIT A

```
unsigned long long Time() const override
{
  auto ts = std::get<1>(std::move(Base::metrics_->GetDataPointAndTime()));
  return ts;
}
```

Superfluous (potentially dangerous?) call to `std::move`.

NRVO is not guaranteed in debug mode. Better:

```
unsigned long long Time() const override
{
  return std::get<1>(Base::metrics_->GetDataPointAndTime());
}
```

INTERLUDE

Before we continue…

3. PUTTING STUFF INTO A `vector`

Should you use `push_back` or should you use `emplace_back`?

How should you use them?

> *"Didst thou not say, when I did push thee back --*
> *Which was when I perceived thee -- that thou camest*
> *From good descending?"*

-- William Shakespeare, Pericles

push_back **AND** emplace_back

```
void push_back(const T& x);
void push_back(T&& x);

template <class... Args>
reference emplace_back(Args&&... args);
```

EXAMPLE 1

What's the difference here?

```
std::vector<std::string> v;
std::string s;
// ...

v.push_back(std::move(s));
v.emplace_back(std::move(s));
```

EXAMPLE 1.1

What's the difference here?

```cpp
std::vector<std::string> v;
std::string s;
// ...

v.push_back(std::move(s));
std::string& last_s = v.emplace_back(std::move(s));
```

EXAMPLE 2

What's the difference here?

```
std::vector<std::string> v;
const char* s = "Hello";
// ...

v.push_back(s);
v.emplace_back(s);
```

EXAMPLE 2.1

Default in-place construct.

```
std::vector<S> v;

// first default-construct in the vector
S& s = v.emplace_back();

// now mutate s
// ...
```

`emplace_back` **takes a parameter pack. Parameter packs can be empty.**

EXAMPLE 3

In-place construct with explicit constructor.

```
// recall: S has an explicit constructor from int
std::vector<S> v;

// push_back can't do explicit construction
v.push_back(1);  // compiler error!

// explicit construction is no problem for emplace_back
S& s = v.emplace_back(1);
```

emplace_back does perfect forwarding. It can call explicit constructors.

EXAMPLE: COPY

Recall: our `S` class has a constructor from `Arg`, and an `explicit` constructor from `int`.

```
std::array<Arg, 3> a = { Arg{}, Arg{}, Arg{} };

std::vector<S> v;
v.reserve(a.size());
std::copy(a.cbegin(), a.cend(), std::back_inserter(v));
```

What does `back_insert_iterator` do here?

EXAMPLE: COPY

What if we have an `array` of `int`?

```
std::array a = { 1,2,3,4,5 };

std::vector<S> v;
v.reserve(a.size());
std::copy(a.cbegin(), a.cend(), std::back_inserter(v));
```

EXAMPLE: COPY

What if we have an array of int?

```
std::array a = { 1,2,3,4,5 };

std::vector<S> v;
v.reserve(a.size());
std::copy(a.cbegin(), a.cend(), std::back_inserter(v));
```

Oops. The compiler is angry at us.

EXAMPLE: COPY?

OK, no problem, right?

```
std::vector<S> v;
std::array a = { 1,2,3,4,5 };
v.reserve(a.size());

std::transform(a.cbegin(), a.cend(), std::back_inserter(v),
               [] (int i) { return S{i}; });
```

back_emplacer

```
template <typename Container>
struct back_emplace_iterator
{
  explicit back_emplace_iterator(Container& c) : c(&c) {}

  back_emplace_iterator& operator++() { return *this; }
  back_emplace_iterator& operator*() { return *this; }

  template <typename Arg>
  back_emplace_iterator& operator=(Arg&& arg) {
    c->emplace_back(std::forward<Arg>(arg));
    return *this;
  }

private:
  Container* c;
};
```

(Slideware - some details omitted)

back_emplacer

```
// pre-CTAD maker function

template <typename Container>
auto back_emplacer(Container& c)
{
  return back_emplace_iterator<Container>(c);
}
```

(Or write a deduction guide for C++17)

back_emplacer

What if we have an array of int?

```
std::vector<S> v;
std::array a = { 1,2,3,4,5 };
v.reserve(a.size());

std::copy(a.cbegin(), a.cend(), back_emplacer(v));
```

The compiler is happy now! And we get in-place construction.

EXHIBIT B

```
std::vector<std::string_view> tokens;
// ...
std::string_view token = /* stuff */;
tokens.emplace_back(std::move(token));
```

EXHIBIT C

```
m_headers.emplace_back(std::string(headerData, numBytes));
```

A superfluous move! Better:

```
m_headers.emplace_back(headerData, numBytes);
```

Don't explicitly call a constructor with `emplace_back`.

vector OF pair = map

Sometimes, we use a sorted vector of pair as a replacement for map.

What do you do if part of your pair has a multi-argument constructor?

```
struct Value { Value(int, std::string, double); };

std::vector<std::pair<int, Value>> v;

// this is very common!
v.push_back(std::make_pair(1, Value{42, "hello", 3.14}));

// this is no better
v.emplace_back(std::make_pair(1, Value{42, "hello", 3.14}));

// how can we do better?
v.emplace_back( /* what here? */ );
```

piecewise_construct TO THE RESCUE!

pair has a constructor that will handle your multi-argument constructor.

```
template <class... Args1, class... Args2>
pair(piecewise_construct_t,
     tuple<Args1...> first_args,
     tuple<Args2...> second_args);

template <class... Types>
constexpr tuple<Types&&...> forward_as_tuple(Types&&... args) noexcept;
```

piecewise_construct_t is a tag type.

USING piecewise_construct

```cpp
struct Value { Value(int, std::string, double); };

std::vector<std::pair<int, Value>> v;

// instead of this...
v.push_back(std::make_pair(1, Value{42, "hello", 3.14}));

// ...we can do this
v.emplace_back(
  std::piecewise_construct,
  std::forward_as_tuple(1),                    // args to int "constructor"
  std::forward_as_tuple(42, "hello", 3.14)); // args to Value constructor
```

Perfect forwarding and in-place construction.

RECOMMENDATIONS

- `push_back` is perfectly fine for rvalues
- use `emplace_back` only when you need its powers
  - in-place construction (including nullary construction)
  - a reference to what's added (C++17)
- never pass an explicit temporary to `emplace_back`
- use `piecewise_construct` / `forward_as_tuple` to forward args through `pair`

4. initializer_list

> *"I fear these stubborn lines lack power to move."*

-- William Shakespeare, Love's Labours Lost

WHAT IS `initializer_list`?

When you write:

```
std::vector<int> v{ 1,2,3 };
```

It's as if you wrote:

```
const int a[] = { 1,2,3 };
std::vector<int> v = std::initializer_list<int>(a, a+3);
```

initializer_list HAS const STORAGE, 1

```
template <int... Is>
auto f() ()
{
  return std::initializer_list<int>{ Is... };
}

void fine() {
  for (int i: {1,2,3})
    cout << i << '\n';
}

void works_fine_until_it_explodes() {
  for (int i: f<1,2,3>())
    cout << i << '\n';
}
```

initializer_list HAS const STORAGE, 2

```
unique_ptr<int> v = { make_unique<int>(1), make_unique<int>(2) };
```

That also means move can't work.

```
const std::unique_ptr<int> a[] = { std::make_unique<int>(1),
                                   std::make_unique<int>(2) };
std::vector<std::unique_ptr<int>> v =
  std::initializer_list<std::unique_ptr<int>>(a, a+3);
```

BUT THEY'RE SO CONVENIENT!

I'd much rather write:

```
std::vector<S> v = { S{1}, S{2}, S{3} };
```

(3 constructs, 3 copies, 3 destructs)

Than:

```
std::vector<S> v;
v.reserve(3);
v.emplace_back(1);
v.emplace_back(2);
v.emplace_back(3);
```

(3 constructs)

WE CAN MAKE IT A LITTLE(?) BETTER...

```
std::vector<S> v = { S{1}, S{2}, S{3} };
```

(3 constructs, 3 copies, 3 destructs)

```
S a[3] = { S{1}, S{2}, S{3} };
std::vector<S> v(std::make_move_iterator(std::begin(a)),
                 std::make_move_iterator(std::end(a)));
```

(3 constructs, 3 moves, 3 destructs)

WHAT WE REALLY NEED...

Is an in-place constructor for `vector`. (For everything?)

```
template <class... Args>
explicit vector(in_place_t, Args&&... args);
```

There is some work going on here, e.g. future (?) proposal by Sy Brand & Chris Di Bella... https://wg21.tartanllama.xyz/initializer_list

EXHIBIT D

```cpp
std::unordered_set<std::string> kKeywords = {
    "alignas", "alignof", "and", "and_eq", "asm", "auto", "bitand", "bitor",
    "bool", "break", "case", "catch", "char", "class", "compl", "const",
    "constexpr", "const_cast", "continue", "decltype", "default", "delete",
    "do", "double", "dynamic_cast", "else", "enum", "explicit", "extern",
    "false", "float", "for", "friend", "goto", "if", "inline", "int", "long",
    "mutable", "namespace", "new", "noexcept", "not", "not_eq", "NULL",
    "operator", "or", "or_eq", "private", "protected", "public", "register",
    "reinterpret_cast", "return", "short", "signed", "sizeof", "static",
    "static_assert", "static_cast", "struct", "switch", "template", "this",
    "thread_local", "throw", "true", "try", "typedef", "typeid", "typename",
    "union", "unsigned", "using", "virtual", "void", "volatile", "wchar_t",
    "while", "xor", "xor_eq"
};
```

CAVEAT CONSTRUCTOR

`std::string` is an interesting case here. We intuit/are taught:

> *Delay construction, allocation, etc. as late as possible.*

But that might hurt us with `std::string`.

"Initializer Lists are Broken, Let's Fix Them" – Jason Turner, C++Now 2018

SURPRISING: `string` VS `const char*`

SBO-strings http://quick-bench.com/5dPSX8rx-R8_BIUYbYOp6DcqhAc

Non SBO-strings 1: http://quick-bench.com/mr6ZIQ8Jy0ghe1scBcTznYF2s5w

Non SBO-strings 2: http://quick-bench.com/vzlG11LwZN-uMAKdK8X1XgRuaWs

RECOMMENDATIONS

- use `initializer_list` only for literal types
- consider using `array` and manually moving?
- probably don't use `initializer_list` for anything that'll get run more than once
- wait for an `in_place_t` constructor on `vector`?
- wait for more work on `std::initializer_list`?
- watch Jason's talk

5. PUTTING STUFF INTO A `map`

(or other associative container)

It's ~~a bit~~ complicated.

> *"A plague upon it! I have forgot the map."*

-- William Shakespeare, Henry IV, Part I

`initializer_list` WITH `map`

It's perfectly possible to initialize a map with an `initializer_list`.

```
// recall S has an implicit constructor from Arg

using M = std::map<int, S>;
M m { {0, Arg{}} }; // how many constructs/copies/moves?
```

Use aggregate initialization with `pair`.

Is this good?

ALTERNATIVE: TEMPLATERY

(Originally? by Vittorio Romeo)

```
// call an N-ary function on each lot of N args passed in
template <size_t N, typename F, typename... Ts>
void for_each_n_args(F&& f, Ts&&... ts);
```

```
using M = std::map<int, S>;
M m;
for_each_n_args<2>(
  [&] (auto&& k, auto&& v) {
      m.emplace(forward<decltype(k)>(k),
                forward<decltype(v)>(v)); },
  0, 1); // we can call explicit constructor
```

If you know the types, you can probably write the lambda in a less ugly way.

ALTERNATIVE: MULTI-ARG TEMPLATERY

```
using M = std::map<int, S>;
M m;
for_each_n_args<3>(
  [&] (auto&& k, auto&&... v) {
      m.emplace(
        std::piecewise_construct,
        std::forward_as_tuple(std::forward<decltype(k)>(k)),
        std::forward_as_tuple(std::forward<decltype(v)>(v)...)); },
  0, 1, 2); // explicit multi-arg value constructor
```

Everything constructed in place.

ENOUGH ABOUT INITIALIZING

How about putting things into an existing `map`?

THE EASY WAY: `operator[]`

```
// recall S has an implicit constructor from Arg
// but an explicit constructor from int

using M = std::map<int, S>;
M m;
m[0] = S{1};
m[1] = Arg{};
```

How many constructs/moves/copies?

THE OTHER EASY(?) WAY: insert

```
// recall S has an implicit constructor from Arg
// but an explicit constructor from int

using M = std::map<int, S>;
M m;

// pair<iterator,bool> insert(value_type&& value);

// template <class T1, class T2>
// pair<V1,V2> make_pair(T1&& t, T2&& u);

// alternatives:
m.insert(std::make_pair(0, S{1}));
m.insert(std::pair<int, S&&>(0, S{1}));
m.insert(std::make_pair(0, 1));
```

How many constructs/moves/copies?

emplace

Enter the *wonderful C++11 panacea* **that is move semantics.**

```
// recall S has an implicit constructor from Arg
// but an explicit constructor from int

using M = std::map<int, S>;
M m;

// template <class... Args>
// pair<iterator,bool> emplace(Args&&... args);

// this was 2 moves
// m.insert(make_pair(0, S{1}));

// much better, right?
m.emplace(std::make_pair(0, S{1}));
```

You guessed it…

emplace, BETTER USAGE

```
// recall S has an implicit constructor from Arg
// but an explicit constructor from int

using M = std::map<int, S>;
M m;

// template <class... Args>
// pair<iterator,bool> emplace(Args&&... args);

m.emplace(0, 1); // no moves, just a construct
```

`emplace` PROBLEM

What do we do when we want to default-construct the value?

```
using M = std::map<int, S>;
M m;
m.emplace(0); // default construct S please!
```

`emplace` PROBLEM

What do we do when we want to default-construct the value?

```
using M = std::map<int, S>;
M m;
m.emplace(0); // default construct S please!
```

error 2665: `std::pair<const _Kty,_Ty>::pair`: none of the 2 overloads could convert all the argument types

`emplace` PROBLEM

What do we do when we want to default-construct the value?

```
using M = std::map<int, S>;
M m;
m[0]; // default construct S please!
```

emplace PROBLEM

What do we do when we want to default-construct the value?

```
using M = std::map<int, S>;
M m;
m[0]; // default construct S please!
```

ಠ_ಠ

`emplace` WITH ZERO-ARG CONSTRUCTOR

Our old friend `piecewise_construct` can help.

```
using M = std::map<int, S>;
M m;
m.emplace(std::piecewise_construct,
          std::forward_as_tuple(0),
          std::forward_as_tuple()); // default construct S please!
```

Tuples are allowed to be empty!

Yes, we can also use this for more-than-one-arg constructors.

EXHIBIT E

```
// explicit ClientRecord(
//     const string& clientId,
//     const ProcessId& clientProcess,
//     const MachineId& clientMachine);

using Storage = std::unordered_set<ClientRecord>;
Storage m_storage;
m_storage.emplace(clientId, processId, machineId);
```

Perfectly fine as far as `emplace` usage.

Then we want to change the `unordered_set` to an `unordered_map`.

EXHIBIT E

```cpp
// explicit ClientRecord(
//     const string& clientId,
//     const ProcessId& clientProcess,
//     const MachineId& clientMachine);

using Storage = std::unordered_map<std::string, ClientRecord>;
Storage m_storage;
m_storage.emplace(
  std::make_pair(clientId,
                 ClientRecord(clientId, processId, machineId)));
```

Is this optimal?

EXHIBIT E

```
using Storage = std::unordered_map<std::string, ClientRecord>;
Storage m_storage;
m_storage.emplace(std::piecewise_construct,
                  std::forward_as_tuple(clientId),
                  std::forward_as_tuple(clientId, processId, machineId));
```

Use `piecewise_construct` again.

`emplace` PROBLEM 2

What do you do if you want to `emplace` the result of a function call?

```
S get_S() { return S{1}; }
```

```
using M = std::map<int, S>;
M m;
m.emplace(0, get_S());
```

How can we avoid the move?

Is it possible to in-place construct here?

IN-PLACE CONSTRUCT A FUNCTION CALL RESULT

We can't avoid evaluating the function call before calling `emplace`.

But we can control when the result of the function call becomes an `S`.

IN-PLACE CONSTRUCT A FUNCTION CALL RESULT

```
template <typename F>
struct with_result_of_t
{
  using T = std::invoke_result_t<F>;
  explicit with_result_of_t(F&& f) : f(std::forward<F>(f)) {}
  /* explicit(false) */ operator T() { return f(); }

private:
  F f;
};

// prior to CTAD
template <typename F>
inline auto with_result_of(F&& f) {
  return with_result_of_t<F>(std::forward<F>(f));
}
```

Superconstructing super elider, Arthur O'Dwyer

Rvalues redefined, Andrzej Krzemieński

emplace PROBLEM 2

```
S get_S() { return S{1}; }

using M = std::map<int, S>;
M m;
m.emplace(0, with_result_of([] { return get_S(); }));

// m.emplace(0, with_result_of(get_S));
```

Compilers are really good at optimizing single-use lambdas.

C++17: `insert_or_assign`

Of course, `insert` / `emplace` and `operator[]` actually do different things.

What do you do if you want to insert, or assign if the element is already there?

```
template <class M>
pair<iterator, bool> insert_or_assign(const key_type& k, M&& obj);

template <class M>
pair<iterator, bool> insert_or_assign(key_type&& k, M&& obj);
```

Introduced with C++17.

C++17: insert_or_assign

```
using M = std::map<int, S>;
M m;
m.insert_or_assign(0, Arg{}); // implicit construction - fine

// m.insert_or_assign(1, 1);  // explicit construction - error!
m.insert_or_assign(1, with_result_of([]{ return S{1}; })); // RVO
```

IN CASE YOU'RE NOT KEEPING COUNT...

We now have at least ~~3~~ 4 ~~5~~ N (>5) different *interface styles* for putting things in a `map`...

- `insert` takes a `value_type` (aka `pair`)
  - or an iterator pair
  - or an `initializer_list`
  - or a node
- `emplace` takes a parameter pack
- `try_emplace` takes a key and a parameter pack
- `insert_or_assign` takes a key and [something convertible to] a value
  - so does `operator[]` (without forwarding)
- `merge` takes another map...

See also: "A Clean and Minimal Map API" – Chandler Carruth, C++Now 2019

`emplace` & `emplace_back` EPILOGUE

What to do if `mapped_type` is an aggregate? You want the rule of zero.

C++20 **P0960**: Aggregate initialization with parentheses.

```
using M = std::map<int, Aggregate>;

M m;
m.emplace(std::piecewise_construct,
          std::forward_as_tuple(1),
          std::forward_as_tuple(/* members of Aggregate */));
```

RECOMMENDATIONS

Yes, C++ is complicated.

- Initialization: consider `for_each_n_args`
- You can use `insert` with `make_pair` and implicit construction
  - But don't use call-site explicit construction
- Use `emplace` but beware of explicit construction
- Use `piecewise_construct` for other than single-arg construction
- Use `operator[]` only when you know the key exists
- Adopt `insert_or_assign` when you can
- Consider `with_result_of`
- Aggregates will suck until C++20

Or, use a non-standard map with a better API

6. PUTTING STUFF INTO OTHER THINGS

Like `optional`, `variant`, `any`.

> *"There's more depends on this than on the value."*

-- William Shakespeare, The Merchant of Venice

`optional` AND FRIENDS

```
template <class... Args>
constexpr explicit optional(in_place_t, Args&&... args);

template <class T, class... Args>
constexpr explicit variant(in_place_type_t<T>, Args&&... args);
template <size_t I, class... Args>
constexpr explicit variant(in_place_index_t<I>, Args&&... args);

template <class ValueType, class... Args>
explicit any(in_place_type_t<ValueType>, Args&&... args);
```

optional CONSTRUCTION

implicit constructor

```
std::optional<S> opt = Arg{};
```

explicit constructor (naive method)

```
std::optional<S> opt = S{1};
```

explicit constructor (in-place method)

```
std::optional<S> opt(std::in_place, 1);
```

optional ASSIGNMENT

implicit constructor

```
std::optional<S> opt;
opt = Arg{};
```

explicit constructor (naive method)

```
std::optional<S> opt;
opt = S{1};
```

explicit constructor (in-place method)

```
std::optional<S> opt;
opt.emplace(1);
```

optional RECOMMENDATIONS

- use the in_place_t constructor
- avoid explicit construction
- use emplace for assignment

```
std::optional<S> opt(std::in_place, 1);
opt.emplace(2);
```

variant CONSTRUCTION

implicit constructor

```
std::variant<int, S> v = Arg{};
```

explicit constructor (naive method)

```
std::variant<int, S> v = S{1};
```

explicit constructor (oops method)

```
std::variant<int, S> v = 1;
```

`variant` CONSTRUCTION

Recommendation: use either of these two constructions.

```
std::variant<int, S> v(std::in_place_type<S>, 1);
```

```
std::variant<int, S> v(std::in_place_index<1>, 1);
```

variant ASSIGNMENT

Similar story to construction.

```
std::variant<int, S> v;

v = Arg{}; // fine
v = S{1};  // constructs a temporary
v = 1;     // oops
```

`variant` DANGER!

Implicitly-typed `variant` construction/assignment can be dangerous.

```
int main() {
  std::variant<bool, std::string> v = "Hello";
  std::cout << "index is " << v.index() << '\n';
}
```

What does this output?

`variant` DANGER!

Implicitly-typed `variant` construction/assignment can be dangerous.

```
int main() {
  std::variant<bool, std::string> v = "Hello";
  std::cout << "index is " << v.index() << '\n';
}
```

What does this output?

C++20 P0608 A sane variant converting constructor

SAFE, EFFICIENT `variant` ASSIGNMENT

```
std::variant<int, S> v;
// template <class T, class... Args>
// T& emplace(Args&&... args);
v.emplace<S>(1);  // S{1}
```

```
std::variant<int, S> v;
// template <size_t I, class... Args>
// variant_alternative_t<I, variant>& emplace(Args&&... args);
v.emplace<1>(1);  // S{1}
```

`variant` RECOMMENDATIONS

- always be explicit about types
- use `in_place_type` or `in_place_index` constructors
- use `emplace<T>` or `emplace<I>`
- avoid `operator=` (except actual `variant`-to-`variant`)

7. FINAL GUIDELINES AND RECOMMENDATIONS

> *"Share the advice betwixt you; if both gain all,*
> *The gift doth stretch itself as 'tis receiv'd,*
> *And is enough for both."*

-- William Shakespeare, All's Well That Ends Well

RECOMMENDATIONS

Think about copies and moves.

Moves aren't free, and may not be cheap.

Usually, in-place construction is preferable. And it is (nearly?) always possible.

Know how RVO works, and check that the compiler is doing it when you think it is.

Study the interfaces of the containers you're using.

Don't be afraid to use `push_back`.

Beware `initializer_list`.