Associative containers

The art of inserting gracefully

Jean Guegant

Conditional insertion: if not already in there

Overlookuping : overlooking the lookups

```
std::unordered_map<std::string, aclass> cache;

auto it = cache.find(key);

if (it == cache.end()) {
     cache[key] = aclass(desc);
}

return cache[key];
```

Overlookuping : overlooking the lookups

```
std::unordered_map<std::string, aclass> cache;

auto it = cache.find(key);

if (it == cache.end()) {
    cache[key] = aclass(desc);
}

return cache[key];
```

Overlookuping : overlooking the lookups

```
std::unordered_map<std::string, aclass> cache;

auto it = cache.find(key);

if (it == cache.end()) {
    cache[key] = aclass(desc);
}

return cache[key];
```

SLOW

C++98

```
auto result = cache.insert(
    std::pair<std::string, aclass>(
        key,
        desc
    )
);

return result.first->second;
```

C++98

```
auto result = cache.insert(
    std::pair<std::string, aclass>(
        key,
        desc
    )
);

return result.first->second;
```

SLOW

C++11

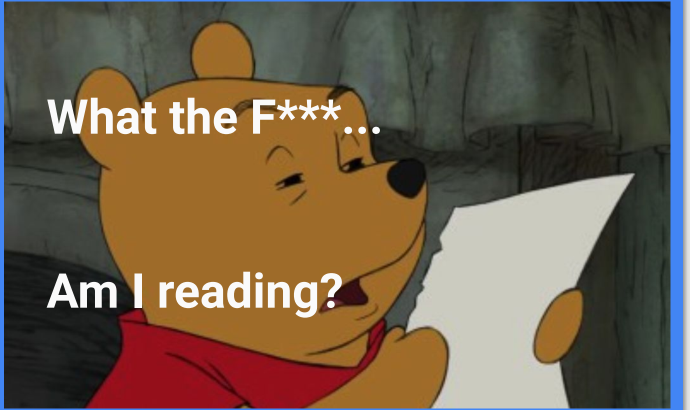
```cpp
auto result = cache.emplace(key, desc);

return result.first->second;
```

C++11

```
auto result = cache.emplace(key, desc);

return result.first->second;
```

SLOW

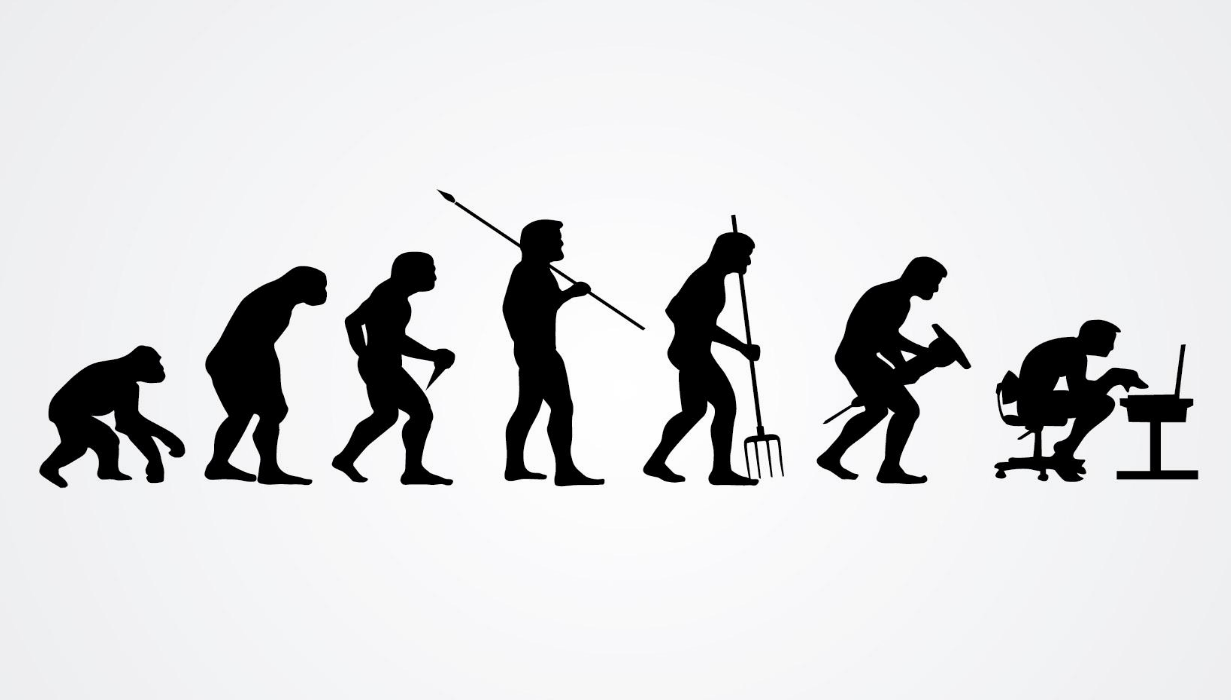
The amazing standard quoting interlude



Effects: *Inserts a value_type object t constructed with std::forward<Args>(args)... if and only if there is no element in the container with key equivalent to the key of t. The bool component of the returned pair is true if and only if the insertion takes place, and the iterator component of the pair points to the element with key equivalent to the key of t.*

What about the failure case?

Cppreference: *The element may be* ***constructed even*** *if there* ***already*** *is an element with the* ***key*** *in the container, in which case the newly constructed element will be destroyed immediately.*



HOMO CPLUSPLUS COMMITUS

C++17

```cpp
auto [it, success] = cache.try_emplace(key, desc);

return it->second;
```

C++17

```
auto [it, success] = cache.try_emplace(key, desc);

return it->second;
```

FINE

> Smart pointers joins the game!

Back to square one

```
std::unordered_map<std::string, std::unique_ptr<aclass>> cache;


auto [it, success] = cache.try_emplace(key, std::make_unique<aclass>(desc));
```

Allocate & construct

Back to square one

```
std::unordered_map<std::string, std::unique_ptr<aclass>> cache;

auto [it, success] = cache.try_emplace(key, std::make_unique<aclass>(desc));
```

SLOW

Allocate & construct

Exception safety

```
auto [it, success] = cache.try_emplace(key, nullptr);

if (success) {
     it->second = std::make_unique<aclass>(desc);
}
```

Exception safety

```
auto [it, success] = cache.try_emplace(key, nullptr);

if (success) {
     it->second = std::make_unique<aclass>(desc);
}
```

What if there is an exception?

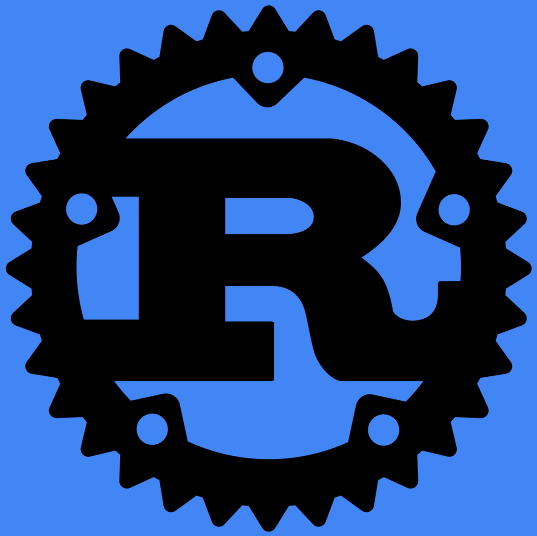
Exception safety

DANGEROUS

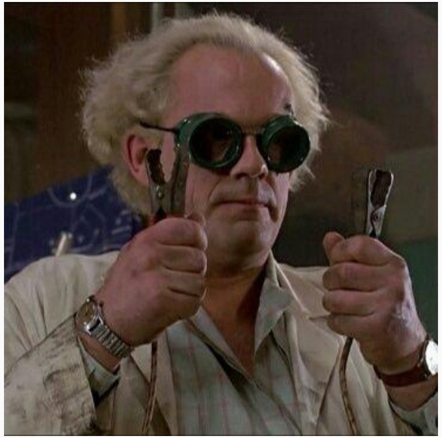
```
auto [it, success] = cache.try_emplace(key, nullptr);

if (success) {
     it->second = std::make_unique<aclass>(desc);
}
```

What if there is an exception?

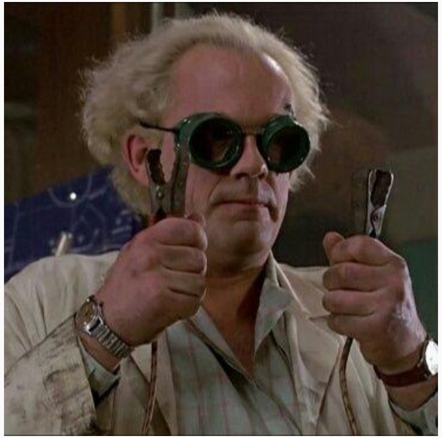
So... I had an affair



Lazy arguments à la D

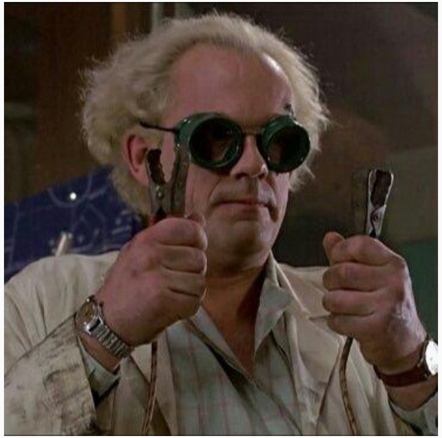


Lazy arguments à la D



```
template<class Factory>
struct lazy_arg
{
	using result_type = std::invoke_result_t<const Factory&>;

	constexpr lazy_arg(Factory&& factory) : factory(std::move(factory)) { }




	constexpr operator result_type() const noexcept(std::is_nothrow_invocable_v<const Factory&>) {
		return factory();
	}

	Factory factory;
};
```

Lazy arguments à la D



```
template<class Factory>
struct lazy_arg
{
      using result_type = std::invoke_result_t<const Factory&>;

      constexpr lazy_arg(Factory&& factory) : factory(std::move(factory)) { }




      constexpr operator result_type() const noexcept(std::is_nothrow_invocable_v<const Factory&>) {
            return factory();
      }

      Factory factory;
};
```

Call a callable and return its result

Lazy arguments à la D

```
auto arg = lazy_arg([&desc](){ return std::make_unique<aclass>(desc); });

cache.try_emplace(key, std::move(arg));
```

Award: works with C++17

Factory method à la Rust

```
auto factory = [&desc](){ return std::make_unique<desc>(desc) };

cache.try_emplace_with(key, std::move(factory));
```

Award: neat but unavailable

in_place constructors for smart pointers

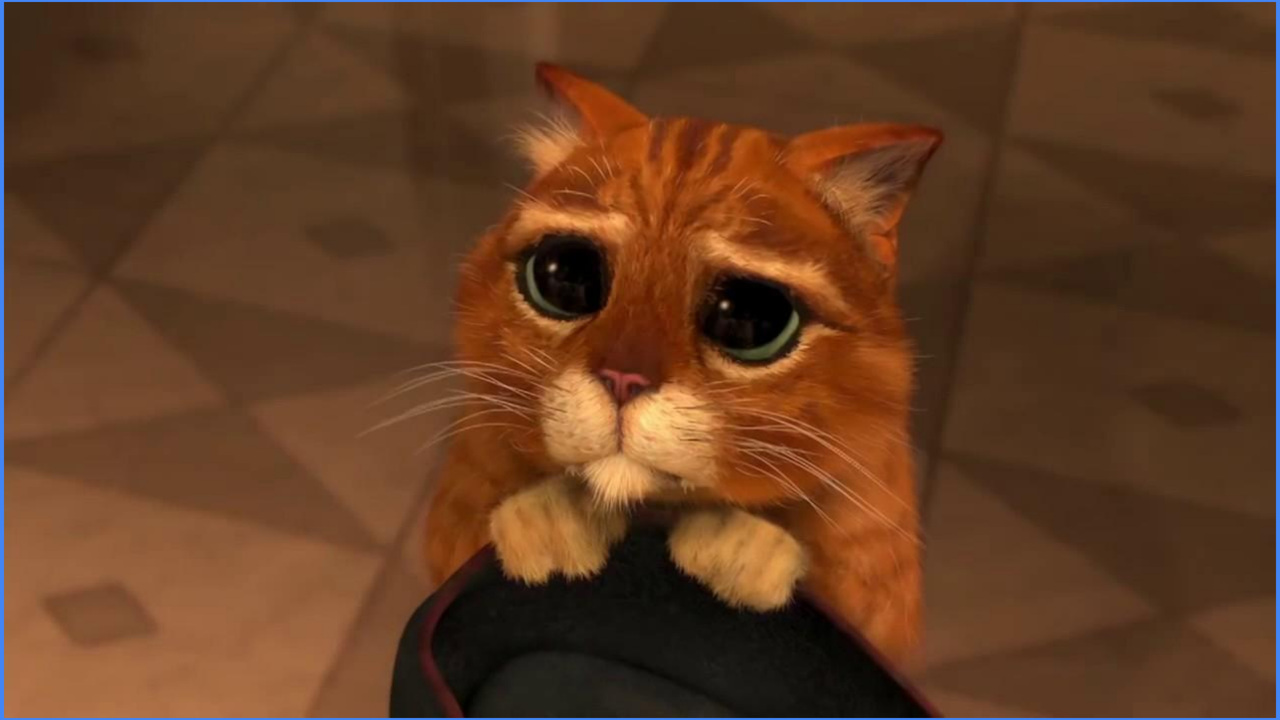
```
cache.try_emplace(key, proposal::allocate_in_place<aclass>{}, desc);
```

Award: can be used with CTAD
(Class Template Argument Deduction)

```
auto ptr = std::unique_ptr(proposal::allocate_in_place<aclass>{}, desc);

==

auto ptr = std::make_unique<aclass>(desc);
```

Thanks

Charming the committee



A recipe for bugs

```
std::string key = "fiction";

auto result = cache.emplace(std::move(key), desc);

if (!result.second) {
    std::cout << "There was an issue with " << key;
}

return result.first->second;
```

A recipe for bugs

```
std::string key = "fiction";

auto result = cache.emplace(std::move(key), desc);

if (!result.second) {
    std::cout << "There was an issue with " << key;
}

return result.first->second;
```

SLOW &
DANGEROUS

Conditional insertion

Associative containers (such as std::map, std::unordered_map...) have seen their interface (or concept) evolve quite a bit along the C++ standards: a lot more lookup and modifiers member functions are now available in C++20 than in C++98. While some of these operations were added mostly for convenience, quite a few of them brought more expressiveness and improved performance alongside. For example: try_emplace (C++17) has more guarantees than emplace (C++11) on what happen to a r-value key , emplace_hint (C++11) can be more efficient with the help of the user, et cetera.