

+ 22

Pragmatic Simplicity

Actionable Guidelines To Tame Complexity

VITTORIO ROMEO



20
22



PRAGMATIC SIMPLICITY

Actionable Guidelines To Tame Complexity

CppCon 2022
September 15

Vittorio Romeo

✉ mail@vittorioromeo.com

🐦 [@supahvee1234](https://twitter.com/supahvee1234)

Bloomberg
Engineering





TechAtBloomberg.com
Careers

A bit about me

- I've been working with C++ for over 10 years
 - Started thanks to game development 🎮
- 6+ YoE at Bloomberg, currently teaching Modern C++
- Co-authored “*Embracing Modern C++ Safely*”
 - J. Lakos, R. Khlebnikov, A. Meredith, and many other contributors
- Participating in ISO C++ standardization
 - Part of the Italian national body
- Many open-source side projects, including:
 - Modernizing [SFML](#) from C++03 to C++17
 - Game development: [Open Hexagon](#), [Quake VR](#)
 - Tools & libraries: [majsdwn](#), [ecst](#), [scelta](#)
 - Video tutorials [on YouTube](#)
 - Articles on [vittorioromeo.com](#)



About this talk

- I care about taming *complexity* – why?
 - **Open-source work:** migrating SFML to C++17
 - **Employment:** teaching Modern C++ at scale
 - **Book authorship:** “Embracing Modern C++ Safely”
 - **Personal interest:** coding can be an art
- Complexity can appear at a *high-level* or *low-level*
 - **High-level:** system design, software architecture, design patterns, etc.
 - **Low-level:** abstraction design, coding style, use of language features, etc.
-  **This talk will focus on *low-level* complexity** 
- Goals:
 - **Derive pragmatic and actionable guidelines from various examples**
 - **When facing a doubt/conflict, solve it using the knowledge gained in this talk**
 - **Spark some interesting discussion!**
-  **Let's keep it *interactive*!** 

Which snippet do you find *simpler*? (#1)



```
int hardcoded_ids[] = {997, 998, 999};
```

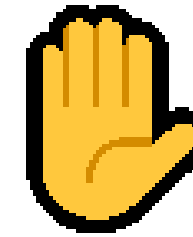


```
std::array hardcoded_ids{997, 998, 999};
```

Which snippet do you find *simpler*? (#2)



```
auto age = int{27};  
  
auto main() -> int  
{  
    std::cout << age << '\n';  
}
```

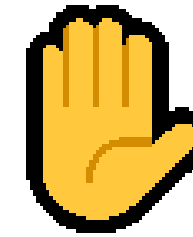


```
int age = 27;  
  
int main()  
{  
    std::cout << age << '\n';  
}
```

Which snippet do you find *simpler*? (#3)



```
for (const int i : irange(0, 100))  
{  
    std::cout << i << ", ";  
}
```



```
for (int i = 0 ; i < 100; ++i)  
{  
    std::cout << i << ", ";  
}
```

Defining “*simplicity*” and “*complexity*”

- We have an *intuition* for simplicity and complexity
 - But we also have biases (e.g. familiarity)
- Generally, we deem some code *simple* if:
 - It is easy to understand, maintain, change, debug, test
 - It protects us from mistakes at compile-time
 - It has a limited amount of “moving parts”
- Unfortunately, there is inherent subjectiveness with all the above criteria
 - Optimizing for a particular goal always involves trade-offs
- Let’s compromise:
 - 1. Somewhat agree on what simplicity means by comparing examples
 - 2. Derive more general precepts from those examples
 - 3. Discuss where such general precepts fall short in the real world

Deriving the first precept

Which snippet do you find *simpler*? (#4)

```
void fill_texture_rect(Color color, int x, int y, int width, int height);
```



```
fill_texture_rect(  
    (int) play_area.x,  
    (int) play_area.y,  
    (int) play_area.width,  
    (int) play_area.height  
);
```



```
fill_texture_rect(  
    static_cast<int>(play_area.x),  
    static_cast<int>(play_area.y),  
    static_cast<int>(play_area.width),  
    static_cast<int>(play_area.height)  
);
```


Casting (#1)

- C-style (and functional) casts are syntactically very concise
 - `static_cast` (etc.) are objectively more verbose
- Yet, a `static_cast` is mechanically simpler than a C-style cast

1) When the *C-style cast expression* is encountered, the compiler attempts to interpret it as the following cast expressions, in this order:

- a) `const_cast<new-type>(expression)` ;
- b) `static_cast<new-type>(expression)` , with extensions: pointer or reference to a **derived class** is additionally allowed to be cast to pointer or reference to unambiguous base class (and vice versa) even if the base class is **inaccessible** (that is, this cast ignores the private inheritance specifier). Same applies to casting **pointer to member** to pointer to member of unambiguous non-virtual base;
- c) `static_cast` (with extensions) followed by `const_cast` ;
- d) `reinterpret_cast<new-type>(expression)` ;
- e) `reinterpret_cast` followed by `const_cast` .

The first choice that satisfies the requirements of the respective cast operator is selected, even if it cannot be compiled (see example). If the cast can be interpreted in more than one way as `static_cast` followed by a `const_cast` , it cannot be compiled.

In addition, C-style cast notation is allowed to cast from, to, and between pointers to incomplete class type. If both *expression* and *new-type* are pointers to incomplete class types, it's unspecified whether `static_cast` or `reinterpret_cast` gets selected.

Casting (#2)

- Why is a `static_cast` simpler than a C-style cast?

- A C-style cast is defined in terms of `static_cast`
- A C-style cast is more powerful than a `static_cast`
- A C-style cast has more “implicit” behavior
- A C-style cast is more error-prone

Let a cast `as-expression` be of the form:

```
x as T                                // x is an expression
```

where `as` has the same precedence as `is`, `P` is a type predicate or specific type, the type of the expression is `refto(T,x)` unless otherwise specified, and:

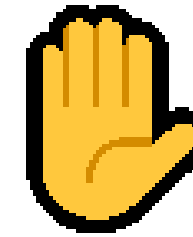
- If `std::is_same_v<T,typeof(x)>`, then `x as T` means a reference to `x`.
- Otherwise, if `x` can be bound to a `refto(T,x)`, then `x as T` means a `refto(T,x)` bound to `x`.
- Otherwise, if `operator as<T>(x)` or `x.operator as<T>()` is available, then use that. (See §2.3.3.)
- Otherwise, if `typeof(x)` is implicitly convertible to `T`, then `x as T` means to convert `x` to an rvalue of type `T` (e.g., including the case where both are Pointer types and this is a static upcast).
- Otherwise, if `P(x)` is valid and dereferenceable using unary `*`, then `x as T` means `*C(x)`.
- Otherwise, if `dynamic_cast<refto(T,x)>(x)` is well-formed, then `x as T` means `dynamic_cast<refto(T,x)>(x)`.
- Otherwise, if `dynamic_cast<T>(x)` is well-formed, then `x as T` means `dynamic_cast<T>(x)`.
- Otherwise, if `!(x is Pointer)` and `typeof(x)` is explicitly convertible to `T`, then `x as T` means to convert `x` to an rvalue of type `T`.
- Otherwise, if `x as T` is ill-formed.

Note There are two grammar productions for `as-expressions`: (1) expression `as` type-id; (2) expression `as` expression.

Which snippet do you find *simpler*? (#5)



```
std::vector<unsigned short> ports;  
  
ports.emplace_back(27015);  
ports.emplace_back(27016);  
  
listen_on(ports);
```

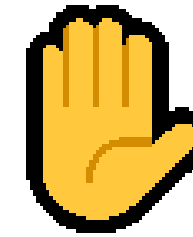


```
std::vector<unsigned short> ports;  
  
ports.push_back(27015);  
ports.push_back(27016);  
  
listen_on(ports);
```

Which snippet do you find *simpler*? (#6)



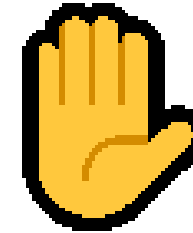
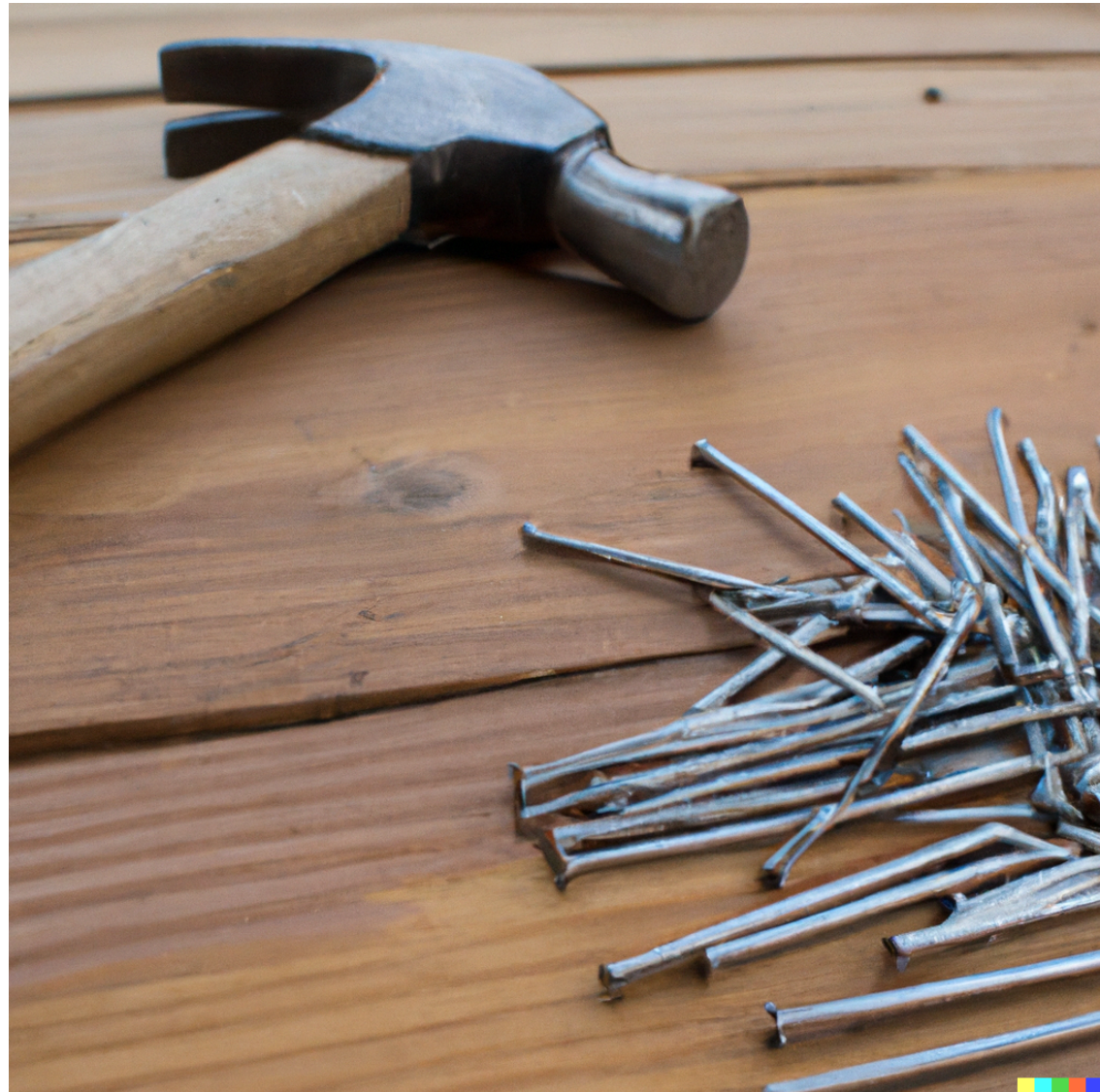
```
{  
    std::lock_guard guard{mutex};  
    something_racy();  
}
```



```
{  
    std::scoped_lock guard{mutex};  
    something_racy();  
}
```


Casting, emplacing, locking – what's in common? (#1)

- **Your task:** hang a painting to the wall, using a nail
 - What tool would you use?



Casting, emplacing, locking – what's in common? (#2)

- A C-style cast is **more powerful** than a `static_cast`
- `emplace_back` is **more powerful** than `push_back`
- `std::scoped_lock` is **more powerful** than `std::lock_guard`
- *“With great power comes great responsibility”*
 - ancient adage, popularized by Spider-Man



Casting, emplacing, locking – what's in common? (#3)

```
{  
    std::scoped_lock guard;  
    something_racy();  
}
```

- Is this code fine? (👉)
- It does compile...
- But nothing is protected!

`std::scoped_lock<MutexTypes...>::scoped_lock`

<code>explicit scoped_lock(MutexTypes&... m);</code>	(1)	(since C++17)
--	-----	---------------

<code>scoped_lock(std::adopt_lock_t, MutexTypes&... m);</code>	(2)	(since C++17)
--	-----	---------------

<code>scoped_lock(const scoped_lock&) = delete;</code>	(3)	(since C++17)
--	-----	---------------

Acquires ownership of the given mutexes m.

1) If `sizeof...(MutexTypes) == 0`, does nothing. Otherwise, if `sizeof...(MutexTypes) == 1`, effectively calls `m.lock()`. Otherwise, effectively calls `std::lock(m...)`.

Casting, emplacing, locking – what's in common? (#4)

```
const std::string my_name = "Vittorio Romeo";

std::vector<std::string> names;

names.emplace_back("John Doe");
    // `std::string::string(const char*)`

names.push_back("John Doe");
    // `std::string::string(const char*)`
    // `std::string::string(std::string&&)`

names.emplace_back(my_name);
    // `std::string::string(const std::string&)`

names.push_back(my_name);
    // `std::string::string(const std::string&)`
```

- `emplace_back` can do more than what `push_back` can
 - However, in some situations, they are equivalent
- 🙋 What's the harm in using `emplace_back` all the time?

Casting, emplacing, locking – what’s in common? (#4)

- It’s about communicating *intent*

```
const std::string my_name = "Vittorio Romeo";  
std::vector<std::string> names;
```

```
// 💡 I am making a conscious decision to use `emplace_back` here,  
//    as it provides a benefit over the use of `push_back`.  
names.emplace_back("John Doe");
```

```
// 💡 I am making a conscious decision to use `push_back` here,  
//    as there is no benefit over the use of `push_back`.  
names.push_back(my_name);
```

- Developers reading your code will know that you’ve *thought* about it
 - Mindlessly using `emplace_back` might not be “dangerous”
 - However, it will reduce the clarity and explicitness of your code
- “Local reasoning is the idea that the reader can make sense of the code directly in front of them, without going on a journey discovering how the code works.” – Nathan Gitter

More (good) examples

- `std::array` vs `T[]`
 - \oplus C-style arrays decay to pointers
- `std::variant` vs regular virtual polymorphism
 - \oplus Virtual polymorphism supports an open set of types
- `std::byte` vs `char`
 - \oplus Characters support all operations of 8-bit integers
- `enum class` vs `enum`
 - \oplus C-style enumerations allow implicit conversions
- `auto&&` vs `const auto&`
 - \oplus Forwarding references are more flexible

Deriving the first precept (#1)



Deriving the first precept (#2)

Use the right tool for the job.

Deriving the first precept (#3)

Use the ~~right~~ tool for the job.

Deriving the first precept (#4)

Use the *most limited* tool for the job.

More (bad) examples (#1)


- `absl::hash_map` vs `std::unordered_map` vs `std::map`
 - ✚ Standard associative containers offer pointer stability
 - 😞 Do we need the extra performance?
 - 😞 Is it reasonable to include an external dependency?
- Aggregate types vs non-aggregates
 - ✚ Aggregates allow (possibly erroneous) partial initialization
 - 😞 Does every simple type require a constructor and/or accessors?

```
struct person_data
```

```
{  
    std::string name;  
    std::string surname;  
    int age;  
    int height;  
};
```

```
person_data me{"Vittorio", "Romeo", 27};  
assert(me.height == 0); // I am not that short...
```

More (bad) examples (#2)

- Uniform initialization
 - \oplus List-initialization can invoke `std::initializer_list` constructors
 - \oplus List-initialization can result in aggregate initialization^(*).
 - \equiv List-initialization prevents implicit narrowing conversions
 -  Which default should be used?

^(*) Since C++20, even using round parentheses can result in aggregate initialization.

The 1st precept

Use the *most limited* tool for the job.^(*)

^(*) Within reason.

A note on abstraction

- The 1st precept holds even at higher level of abstraction, e.g.:
 - Don't use a *class* when a *function* suffices
 - $T \rightarrow \text{std::unique_ptr}\langle T \rangle \rightarrow \text{std::shared_ptr}\langle T \rangle$
 - Only use multithreading when *provably* beneficial
- Implementation complexity can lead to usage simplicity
 - (When properly abstracted)

```
for (const int i : irange(0, 100)) { /* ... */ }  
// vs  
for (int i = 0; i < 100; ++i) { /* ... */ }
```

- Using `irange(0, 100)` is *less powerful* than the traditional loop
 - `i` can be marked as `const` (fewer moving parts)
 - Consistent type between lower and upper boundaries
- However, there are tradeoffs:
 - Compilation speed impact?
 - Teachability?

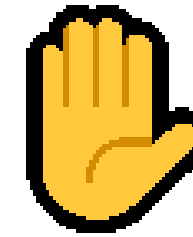
Deriving the second precept

Which snippet do you find *simpler*? (#7)



```
class vec2
{
private:
    float _x, _y;

public:
    float x() const;
    float y() const;
};
```



```
class vec2
{
private:
    float _x, _y;

public:
    [[nodiscard]] float x() const;
    [[nodiscard]] float y() const;
};
```

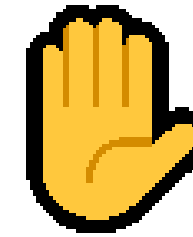

Which snippet do you find *simpler*? (#8)



```
class vec2
{
private:
    float _x, _y;

public:
    float x() const;
    float y() const;

    vec2 normalize() const;
};
```



```
class vec2
{
private:
    float _x, _y;

public:
    [[nodiscard]] float x() const;
    [[nodiscard]] float y() const;

    [[nodiscard]] vec2 normalize() const;
};
```

Which snippet do you find *simpler*? (#9)



```
class vec2
{
private:
    float _x, _y;

public:
    float x() const;
    float y() const;

    [[nodiscard]] vec2 normalize() const;
};
```



```
class vec2
{
private:
    float _x, _y;

public:
    [[nodiscard]] float x() const;
    [[nodiscard]] float y() const;

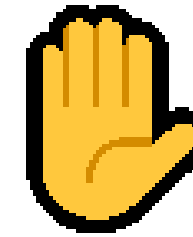
    [[nodiscard]] vec2 normalize() const;
};
```

Which snippet do you find *simpler*? (#10)



```
const long target_id = 10;

for (std::forward_list<int>::iterator it =
    ids.begin(); it != ids.end(); ++it)
{
    if (*it == target_id)
    {
        do_something(it);
    }
}
```



```
const auto target_id = 10;

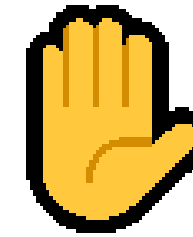
for (auto it = ids.begin(); it != ids.end();
    ++it)
{
    if (*it == target_id)
    {
        do_something(it);
    }
}
```

Which snippet do you find *simpler*? (#11)



```
const auto target_id = 10;

for (auto it = ids.begin(); it != ids.end();
    ++it)
{
    if (*it == target_id)
    {
        do_something(it);
    }
}
```



```
const long target_id = 10;

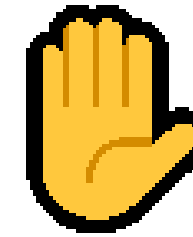
for (auto it = ids.begin(); it != ids.end();
    ++it)
{
    if (*it == target_id)
    {
        do_something(it);
    }
}
```

Which snippet do you find *simpler*? (#12)



```
double cost(float a, float b)
{
    return a * a + b * b - a * b;
}

double growth(float a, float b)
{
    return std::sqrt(a) * b;
}
```

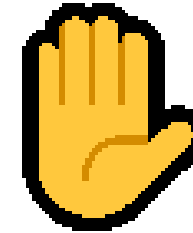


```
double cost(float a, float b) noexcept
{
    return a * a + b * b - a * b;
}

double growth(float a, float b) noexcept
{
    return std::sqrt(a) * b;
}
```


Attributes, auto , noexcept – what's in common? (#1)

- **Your task:** improve the safety of the roads in your city
 - How would you do it?



Attributes, auto , noexcept – what's in common? (#2)

- Zealous use of some features is technically correct, for example:
 - `[[nodiscard]]` and other attributes
 - `auto`, `noexcept`, `final`, `constexpr`
- However, *sparingly* using a feature increases its value and reduces noise:

```
class vec2
{
private:
    float _x, _y;

public:
    float x() const;
    float y() const;

    // 💡 I am consciously using `nodiscard`
    // here but not above, as the name of
    // this function might lead to some
    // misunderstandings and/or bugs.
    [[nodiscard]] vec2 normalize() const;
};
```

```
class node
{
    // ...
public:
    node();

    // 💡 I am consciously using `noexcept`
    // here but not above, as this class
    // might be stored in vectors.
    // Default construction won't throw,
    // but it's unproven that `noexcept`
    // there would result in any benefit.
    node(node&&) noexcept;
};
```


A note on `final` (on classes)

- It is reasonable to use `final` on classes not intended for inheritance
 - But inheritance in C++ is not always about polymorphism
- Are sure that there isn't a valid use case for your type as a base class?
 - E.g. ad-hoc "strong typedef"
- We deem `final` an "unsafe" feature in **Embracing Modern C++ Safely**
 - "Unsafe" doesn't mean inherently bad
 - It means generally not worth using/teaching at a large scale
 - `final` can harm reuse in a large company, which can cost real money
- However, used *very sparingly*, `final` can be a lifesaver:

```
#ifdef COOL_PLATFORM
using int512 = __int512; // fast built-in type
#else
class int512 final
{
    // ...slow software implementation...
};
#endif
```

More examples (good and bad)

- `constexpr` functions
 - ⚠ Overuse can result in API lock-in and more physical dependencies
 - 😞 Even if your function can technically be used at compile-time, will people do so?
- Trailing return types
 - ⚠ Overzealous use might obscure their benefits in specific cases
 - 😞 Isn't it useful to draw more attention to places where they make a difference?
 - Consistent with 1st precept
- `const` variables
 - ⚠ Copious use might be correct but useless (e.g. `const int` parameter in short function)
 - 😞 Wouldn't it make more sense to use `const` where it semantically matters or in large scopes?
 - Inconsistent with 1st precept...? (*non-const is more powerful than const*)
- `override` contextual keyword
 - ➕ Use liberally!
 - Consistent with 1st precept

Deriving the second precept (#1)



Deriving the second precept (#2)

Value is a function of *rarity*.

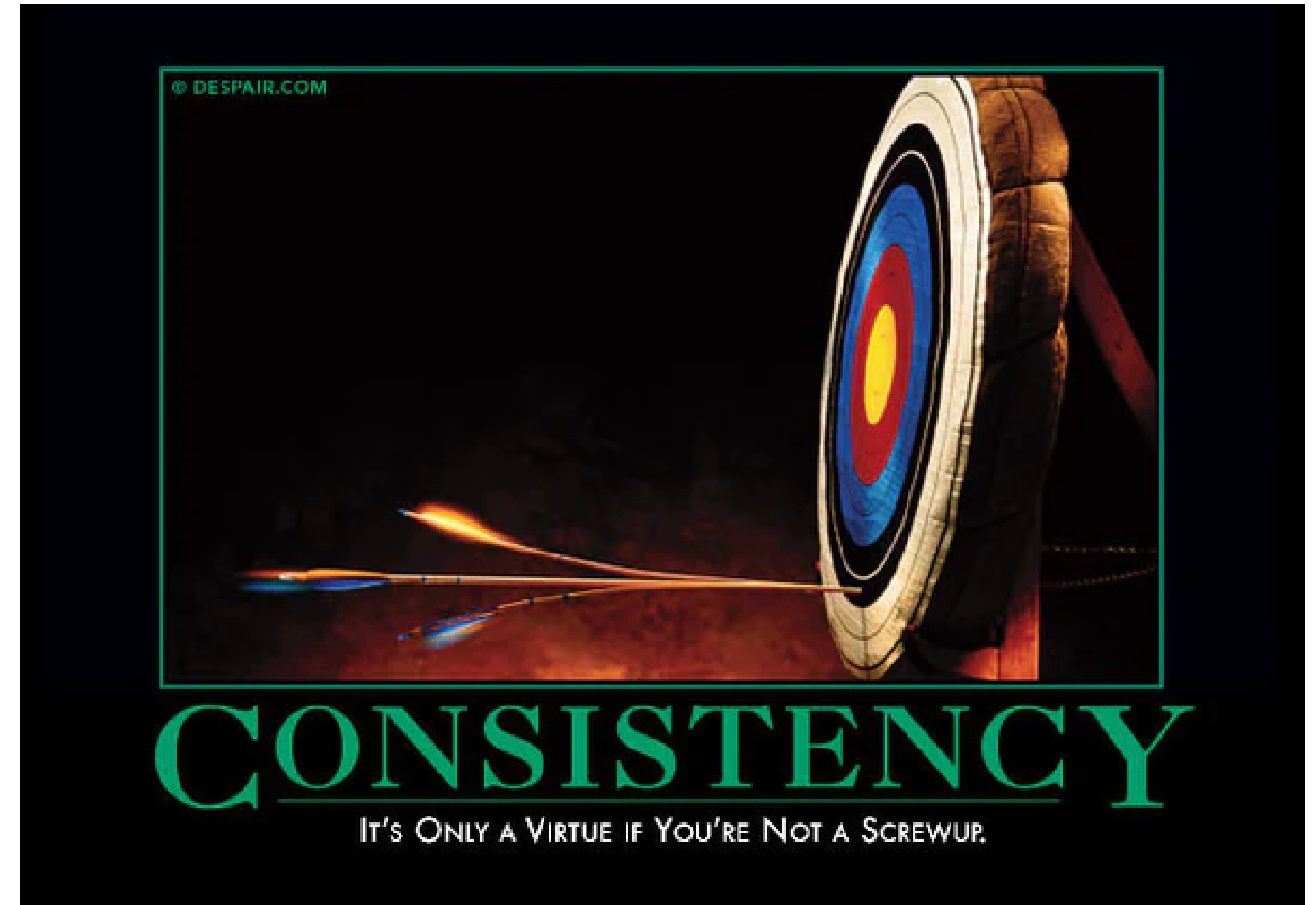
The 2nd precept

Value is a function of rarity.^(*)

^(*) Most of the times.

A note on consistency

- Consistency is valuable and important...
 - ...but it's only a *factor* among many
- Consistency for the sake of it is harmful
- Consistency can be easier than simplicity
 - e.g. `clang-tidy` and `[[nodiscard]]`
 - e.g. strict style guides
- My stance:
 - Value correctness and simplicity over consistency
 - Go for consistency afterwards
 - Don't be dogmatic (if you can afford it)



Conclusion

How to use these precepts

- (1) “**Use the *most limited* tool for the job.**”
- (2) “***Value* is a function of *rarity*.**”
- Scenarios:
 - + Excitement when using a new C++ (or library) feature
 - + Resolving conflict during code reviews or debates
 - + Migrating a legacy project to more modern standards
 - + Preventive damage control for new C++ developers
 - + Teaching and mentorship, reducing decision-making surface area
- Shortcomings:
 - = Additional verbosity
 - = Loss of style consistency
 - = More mental focus required
 - = Sometimes subjective

The truth is in the middle

- Think of these precepts as *guidelines*
 - (1) “**Use the *most limited* tool for the job.**” (*Within reason.*)
 - (2) “**Value is a function of *rarity*.**” (*Most of the time.*)
- Follow the precepts, but *not* blindly
- Precepts and guidelines are *tools*
 - Use them to your own benefit
 - ⚠ Don't let them use *you*!
- I derived the precepts from my own research/experience
 - They have helped me a lot in my latest projects and in teaching
- Want *objective* data to derive your own? Read **EMC++S!**
 - 📖 <https://emcpps.com>
 - No opinions: just *facts, use cases, pitfalls, and annoyances*
- Thanks!
 - 🙋 Questions? 🗨 Comments? 🗨 Criticism? 🗨 Stories?
 - <https://vittorioromeo.com> | [@supahvee1234](https://twitter.com/supahvee1234) | <https://github.com/vittorioromeo> | mail@vittorioromeo.com

