NOKIA

The Less, the Equal and the Greater.

Author: Piotr Nycz (Nokia)

Additional materials: https://github.com/PiotrNycz/LEG

© 2022 Nokia

Agenda: operators ==, <, …

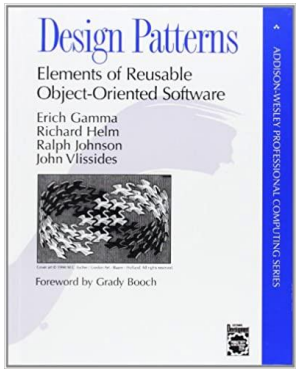
- C++17
  - How to implement them correctly
    - Problems with straightforward ways
  - Be smart – several smarter ways of implementing comparison correctly
- C++20
  - Big change: defaults and spaceship
  - New problems

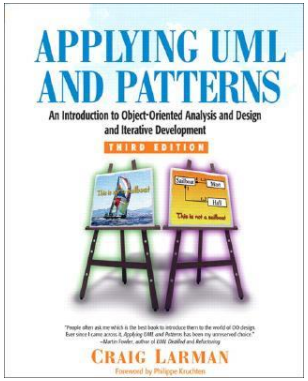
© 2022 Nokia
NOKIA

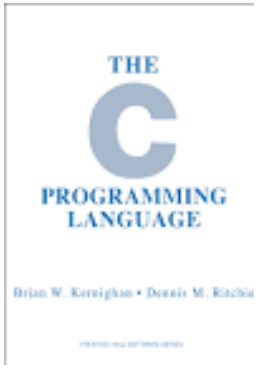
About myself:

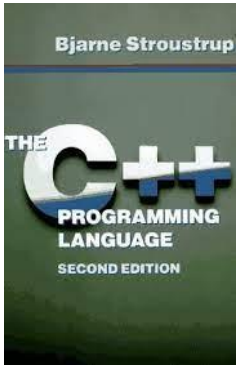
Software Architect and Code Guard at Nokia Krakow

Books:



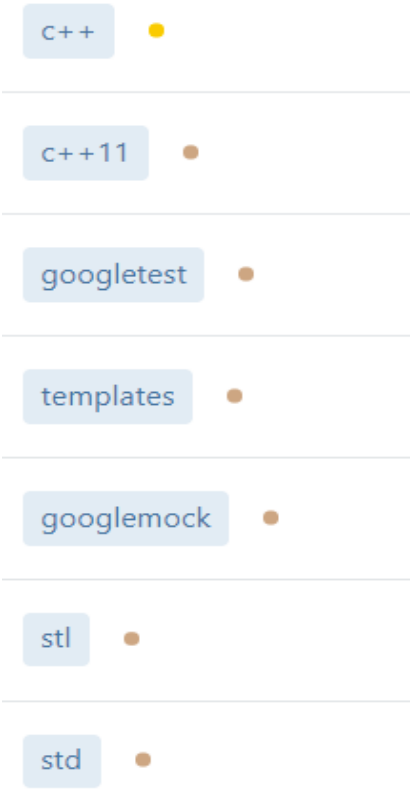






- 1991 – “C”
- 1992 – “C++”
- 1996 – “Design Patterns”

StackOverflow



© 2022 Nokia

Is operator== easy to implement?

```cpp
#include <cstdint>

struct Color {
    uint16_t r,g,b;
};

struct Object {
    uint32_t x;
    uint32_t y;
    bool hasZ;
    uint32_t z;
    uint16_t weight;
    uint16_t velocity;
    Color color;
};
```

```cpp
inline bool operator==(const Color& lhs, const Color& rhs)
{
    return lhs.r == rhs.r
        && lhs.g == rhs.g
        && lhs.b == rhs.b;
}
inline bool operator==(const Object& lhs, const Object& rhs)
{
    return lhs.x == rhs.x
        && lhs.y == rhs.y
        && lhs.hasZ == rhs.hasZ
        && (!lhs.hasZ || (lhs.z == rhs.z))
        && lhs.weight == rhs.weight
        && lhs.velocity == rhs.weight
        && lhs.color == rhs.color;
}
```

Is this implementation correct?

© 2022 Nokia

Is operator== easy to implement?

```cpp
#include <cstdint>

struct Color {
    uint16_t r,g,b;
};

struct Object {
    uint32_t x;
    uint32_t y;
    bool hasZ;
    uint32_t z;
    uint16_t weight;
    uint16_t velocity;
    Color color;
};
```

```cpp
inline bool operator==(const Color& lhs, const Color& rhs)
{
    return lhs.r == rhs.r
        && lhs.g == rhs.g
        && lhs.b == rhs.b;
}
inline bool operator==(const Object& lhs, const Object& rhs)
{
    return lhs.x == rhs.x
        && lhs.y == rhs.y
        && lhs.hasZ == rhs.hasZ
        && (!lhs.hasZ || (lhs.z == rhs.z))
        && lhs.weight == rhs.weight
        && lhs.velocity == rhs.weight
        && lhs.color == rhs.color;
}
```


© 2022 Nokia
NOKIA

Is operator== easy to implement?

Main problem: duplications; duplicated column (not rows)

```
inline bool operator==(const Color& lhs, const Color& rhs)
{
    return lhs.r == rhs.r
        && lhs.g == rhs.g
        && lhs.b == rhs.b;
}
inline bool operator==(const Object& lhs, const Object& rhs)
{
    return lhs.x == rhs.x
        && lhs.y == rhs.y
        && lhs.hasZ == rhs.hasZ
        && (!lhs.hasZ || (lhs.z == rhs.z))
        && lhs.weight == rhs.weight
        && lhs.velocity == rhs.velocity
        && lhs.color == rhs.color;
}
```


© 2022 Nokia

Is operator< easy to implement?

Have to implement lexicographical order

```
#include <cstdint>

struct Color {
    uint16_t r,g,b;
};

struct Object {
    uint32_t x;
    uint32_t y;
    bool hasZ;
    uint32_t z;
    uint16_t weight;
    uint16_t velocity;
    Color color;
};
```

An example: std::pair<T1,T2> operator<()

```
  template<typename _T1, typename _T2>
    inline _GLIBCXX_CONSTEXPR bool
    operator<(const pair<_T1, _T2>& __x, const pair<_T1, _T2>& __y)
    { return __x.first < __y.first
         || (!(__y.first < __x.first) && __x.second < __y.second); }
```

```
inline bool operator<(const Color& lhs, const Color& rhs)
{
    return lhs.r < rhs.r ||
        (!(rhs.r < lhs.r) && (
            lhs.g < rhs.g || (
         (!(rhs.g < lhs.g) && lhs.b < rhs.b))));
}
```

But, how to write operator< for 7 fields Object struct?

© 2022 Nokia

Is operator< easy to implement?

```
#include <cstdint>

struct Color {
    uint16_t r,g,b;
};

struct Object {
    uint32_t x;
    uint32_t y;
    bool hasZ;
    uint32_t z;
    uint16_t weight;
    uint16_t velocity;
    Color color;
};
```

```
inline bool operator<(const Color& lhs, const Color& rhs)
{
    if (lhs.r < rhs.r) return true;
    if (rhs.r < lhs.r) return false;

    if (lhs.g < rhs.g) return true;
    if (rhs.g < lhs.g) return false;

    return lhs.b < rhs.b;
}
```

Better idea: process each field, one by one

© 2022 Nokia

Is operator< easy to implement?

```cpp
#include <cstdint>

struct Color {
    uint16_t r,g,b;
};

struct Object {
    uint32_t x;
    uint32_t y;
    bool hasZ;
    uint32_t z;
    uint16_t weight;
    uint16_t velocity;
    Color color;
};
```

```cpp
inline bool operator<(const Object& lhs, const Object& rhs)
{
    if (lhs.x < rhs.x) return true;
    if (rhs.x < lhs.x) return false;

    if (lhs.y < rhs.y) return true;
    if (rhs.y < lhs.y) return false;

    if (lhs.hasZ != rhs.hasZ) return rhs.hasZ;

    if (lhs.hasZ) {
        if (lhs.z < rhs.z) return true;
        if (rhs.z < lhs.z) return false;
    }

    if (lhs.weight < rhs.weight) return true;
    if (rhs.weight < lhs.weight) return false;

    if (lhs.velocity < rhs.velocity) return true;
    if (rhs.velocity < lhs.velocity) return false;

    return lhs.color < rhs.color;
}
```


© 2022 Nokia

Operators: !=, <=, >, >= are easy!

When == and < are defined!

```
inline bool operator!=(const Object& lhs, const Object& rhs)
{
    return !(lhs == rhs);
}
inline bool operator>(const Object& lhs, const Object& rhs)
{
    return rhs < lhs;
}
inline bool operator>=(const Object& lhs, const Object& rhs)
{
    return !(lhs < rhs);
}
inline bool operator<=(const Object& lhs, const Object& rhs)
{
    return !(rhs < lhs);
}
```


© 2022 Nokia
NOKIA

Straightforward implementation

Summary

PROS

- Simplicity: no extra utilities (helper functions, classes) needed

CONS

- Fields are used twice on left and right side (error prone)
- Big effort to provide test coverage (many tests to write)
- No simple mechanism to ensure that all fields were used (maintainability)
  - We can use static analysis!
    Like write own clang-tidy check to ensure all fields were used in operator==, operator<

© 2022 Nokia

<tuple> std::tie

Duplicated fields problem solved!

```
#include <cstdint>

struct Color {
    uint16_t r,g,b;
};

struct Object {
    uint32_t x;
    uint32_t y;
    bool hasZ;
    uint32_t z;
    uint16_t weight;
    uint16_t velocity;
    Color color;
};
```

```
#include <tuple>

inline auto tieMembers(const Color& arg)
{
   return std::tie(arg.r, arg.g, arg.b);
}
inline bool operator==(const Color& lhs, const Color& rhs)
{
    return tieMembers(lhs) == tieMembers(rhs);
}
inline bool operator<(const Color& lhs, const Color& rhs)
{
    return tieMembers(lhs) < tieMembers(rhs);
}
```


© 2022 Nokia

<tuple> with in-house

If possible – change definition of class to compare.
Remove “primitive obsession” code smell

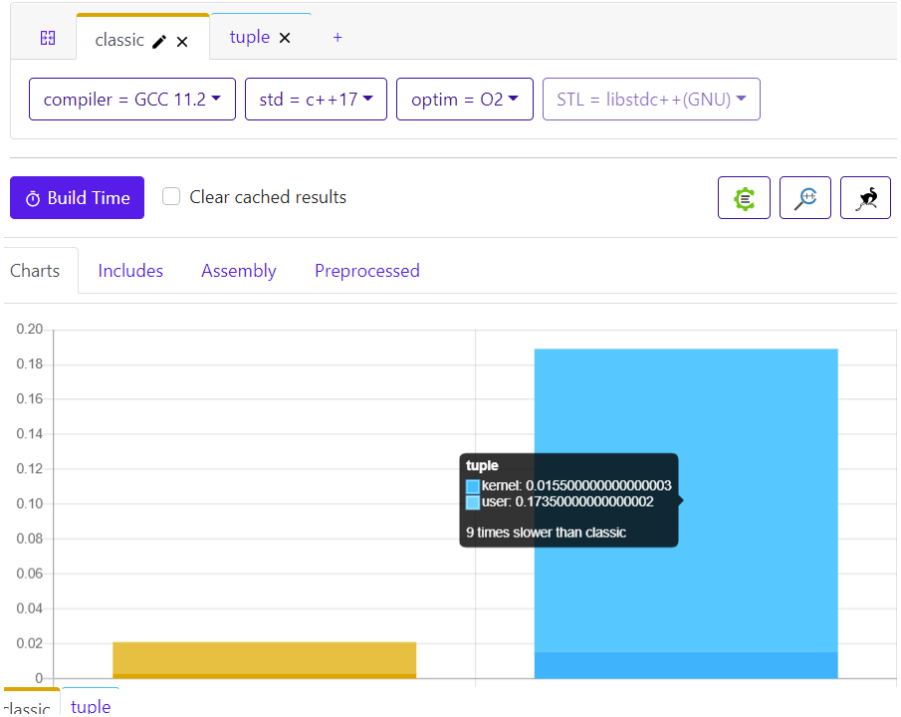
```
template <typename T>
struct TrivialOptional
{
    bool hasValue;
    T value;
    bool operator==(const TrivialOptional& rhs) const
    {
        return hasValue == rhs.hasValue &&
            (!hasValue || (value == rhs.value));
    }
    bool operator<(const TrivialOptional& rhs) const
    {
        return rhs.hasValue && (!hasValue || (value < rhs.value));
    }
};
```

```
#include <cstdint>
#include <tuple>
#include "TrivialOptional.hpp"

struct Color {
    uint16_t r,g,b;
};

struct Object {
    uint32_t x;
    uint32_t y;
    TrivialOptional<uint32> z;
    uint16_t weight;
    uint16_t velocity;
    Color color;
};

inline auto tieMembers(const Object& arg)
{
   return std::tie(arg.x, arg.y, arg.z,
                   arg.weight, arg.velocity,
                   arg.color);
}
```


© 2022 Nokia
NOKIA

<tuple> vs. “classic”

Problems: compilation time https://build-bench.com/



CONS:

- <tuple> **9x slower** than “classic”
- High level of optimization needed to achieve the same performance

PROS:

- Less error-prone (no duplication)
- Single lines to cover (tests)

© 2022 Nokia
NOKIA

Double apply pattern + lambda + fold expressions (C++17)

Yet another solution to duplicated fields problem

```
#include <cstdint>
#include “IsEqual.hpp"

struct Color {
    uint16_t r,g,b;
};
```

```
struct IsEqual {
    template <typename ...V>
    constexpr
    auto operator()(const V& ...lhs) const noexcept
    {
        return [&lhs...](const auto& ...rhs) {
            return ((lhs == rhs) && ...);
        };
    }
};
constexpr IsEqual isEqual{};
```

```
template <typename F> inline auto apply(const F& f, const Color& v)
{
    return f(v.r, v.g, v.b);
}

inline bool operator==(const Color& lhs, const Color& rhs)
{
    const auto lhs_equal_to = apply(isEqual, lhs);
    return apply(lhs_equal_to, rhs);
}
```


© 2022 Nokia

Double apply pattern: spaceship implementation

```
struct Spaceship {
    template <typename V>
    static constexpr int spaceship(const V& lhs, const V& rhs) noexcept
    {
        if (lhs < rhs) return -1;
        if (rhs < lhs) return 1;
        return 0;
    }
    /// lhs<rhs: -1, lhs==rhs: 0, lhs>rhs: 1
    template <typename ...V>
    constexpr auto operator()(const V& ...lhs) const noexcept
    {
        return [&lhs...](const auto& ...rhs) {
            int result = 0;
            static_cast<void>(
                (((result=spaceship(lhs, rhs))==0) && ...)
            );
            return result;
        };
    }
};
constexpr Spaceship spaceship{};
```

- `lhs<rhs  is  -1`
- `lhs==rhs  is  0`
- `lhs>rhs  is  1`

© 2022 Nokia
NOKIA

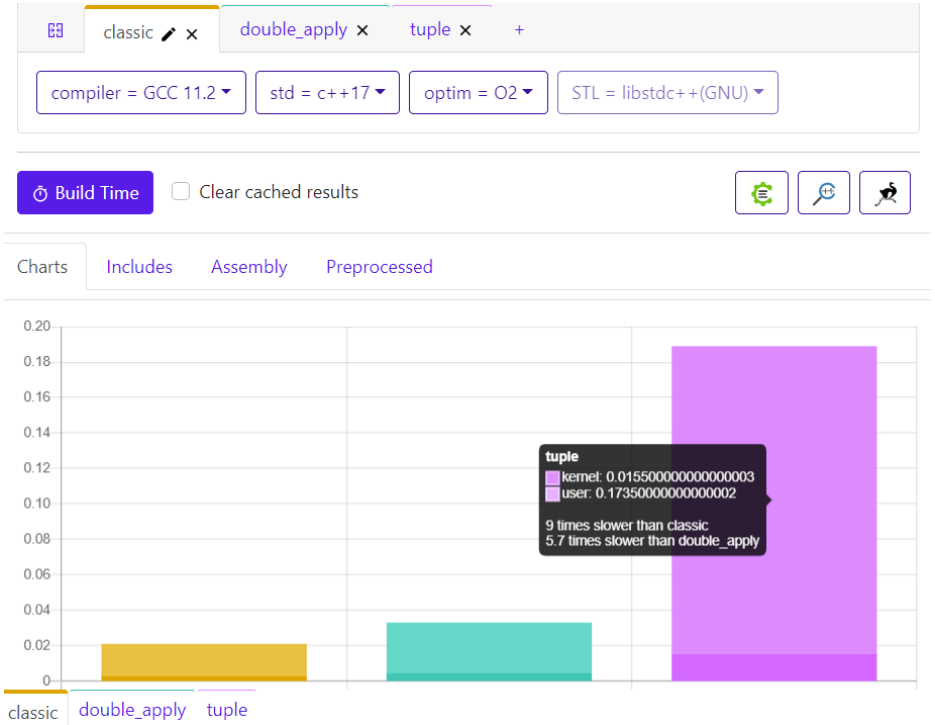
Double apply pattern

```
template <typename F>
inline auto apply(const F& f, const Object& arg)
{
   return f(arg.x, arg.y, arg.z,
            arg.weight, arg.velocity,
            arg.color);
}

inline bool operator==(const Object& lhs, const Object& rhs)
{
    const auto lhs_equal_to = apply(isEqual, lhs);
    return apply(lhs_equal_to, rhs);
}

inline bool operator<(const Object& lhs, const Object& rhs)
{
    const auto lhs_spaceship = apply(spaceship, lhs);
    return apply(lhs_spaceship, rhs) < 0;
}
```


© 2022 Nokia

“classic” vs double-apply vs <tuple>

Problems: compilation time https://build-bench.com/



Classic approach compilation vs.

- <tuple> 900% slower
- Double-apply – 60% slower

<tuple> vs. double-apply(lambdas)

- Lower optimization level (-O2) for double-apply to achieve optimal code

© 2022 Nokia

C++17: summary

Straightforward

PROS:

- Simple, no need for extra utils

CONS:

- Error-prone (duplicated fields)
- Costly to test
- Need to use static-analysis to check completeness (all fields used)

*) https://www.boost.org/doc/libs/1_78_0/libs/preprocessor/doc/index.html

Smarter ways:

PROS:

- Less error-prone (no duplications)
- Easy to test

CONS:

- Heavy for compilation
  - Double-apply pattern less heavy than <tuple>
  - Preprocessor magic can be even easier to compile
    - But BOOST.PP headers are heavy (*)
- High level of optimization needed to achieve the same performance
  - Especially for <tuple> way
- Need to use static-analysis to check completeness (all fields used)

© 2022 Nokia

C++20

Why doesn't compiler generate comparison operators for us?

https://stackoverflow.com/questions/217911/why-dont-c-compilers-define-operator-and-operator

Why don't C++ compilers define operator== and operator!=?

Asked 13 years, 7 months ago Modified 1 year, 11 months ago Viewed 103k times

338

64

I am a big fan of letting the compiler do as much work for you as possible. When writing a simple class the compiler can give you the following for 'free':

- A default (empty) constructor
- A copy constructor
- A destructor
- An assignment operator (`operator=`)

But it cannot seem to give you any comparison operators - such as `operator==` or `operator!=`. For example:

asked Oct 20, 2008 at 9:42

Rob

73.3k • 53 • 155 • 192

© 2022 Nokia

NOKIA

C++20

Why doesn’t compiler generate comparison operators for us?

https://stackoverflow.com/questions/217911/why-dont-c-compilers-define-operator-and-operator

337

The argument that if the compiler can provide a default copy constructor, it should be able to provide a similar default `operator==()` makes a certain amount of sense. I think that the reason for the decision not to provide a compiler-generated default for this operator can be guessed by what Stroustrup said about the default copy constructor in "The Design and Evolution of C++" (Section 11.4.1 - Control of Copying):

> I personally consider it unfortunate that copy operations are defined by default and I prohibit copying of objects of many of my classes. However, C++ inherited its default assignment and copy constructors from C, and they are frequently used.

answered Oct 20, 2008 at 14:53

Michael Burr

321k • 49 • 514 • 738

So instead of "why doesn't C++ have a default `operator==()`?", the question should have been "why does C++ have a default assignment and copy constructor?", with the answer being those

© 2022 Nokia

C++20: defaults not provided

But you can ask – please generate defaults for comparison!

- `= default;` can be specified for any comparison operator (including new <=> spaceship operator)
- Class definition needs to be altered – because default is allowed only to member operator or friend operator
- From <=>, all other operators are derived (automatically), but frequently operator== is defined additionally for pure performance reasons

```
#include <cstdint>

#include <compare>

struct Color {
    uint16_t r,g,b;

    constexpr bool operator==  (const Color&) const noexcept = default;
    constexpr auto operator<=> (const Color&) const noexcept = default;
};
```


© 2022 Nokia
NOKIA

C++20: Spaceship operator by hand

```
template <typename T>
struct TrivialOptional {
    bool hasValue;
    T value;
    constexpr bool operator==(const TrivialOptional& rhs) const noexcept
    {
        return hasValue == rhs.hasValue &&
            (!hasValue || (value == rhs.value));
    }
    constexpr auto operator<=>(const TrivialOptional& rhs) const noexcept
        -> decltype(value <=> rhs.value)
    {
        if (auto res = hasValue <=> rhs.hasValue; res != 0) return res;
        if (!hasValue) return std::strong_ordering::equal;
        return value <=> rhs.value;
    }
};
```

In most cases – return type has to be specified (or like here, with decltype):

std::strong_ordering, (equal means identical)

std::weak_ordering, (equal does not mean identical)

std::partial_ordering (some values are not comparable)

© 2022 Nokia

C++20 comparison

Read more:

- https://en.cppreference.com/w/cpp/language/operator_comparison#Three-way_comparison
- https://en.cppreference.com/w/cpp/header/compare

- **Note**: auto-symmetrical comparison added

```
struct Zero {
    bool operator==(int a) const { return a==0; }
};
int main() {
    (void)(Zero{} == 1);
    (void)(1 == Zero{});
    // C++17 ^^ error: no match for 'operator==' (operand types are 'int' and 'Zero')
    // C++20 OK
}
```


© 2022 Nokia

C++20 Problem: ignoring fields?

```cpp
#include <cassert>
#include <map>
#include <compare>

class Quadratic {
public:
    Quadratic(double a, double b, double c) : a(a), b(b), c(c) {}

    bool operator==(const Quadratic&) const = default;
    auto operator<=>(const Quadratic&) const = default;

    double operator()(double x) const {
        if (auto it = cachedResults.find(x); it != cachedResults.end())
            return it->second;
        return cachedResults.emplace(x, a*x*x + b*x + c).first->second;
    }

private:
    // a*x*x + b*x + c
    double a, b, c;
    // cache for f(x)
    mutable std::map<double, double> cachedResults;
};
```

```cpp
int main() {
    Quadratic f1(1, 2, 3);
    Quadratic f2(1, 2, 3);
    assert(f1 == f2);
    assert(f1(1) == 6);
    assert(f1 == f2);
   // Assertion `f1 == f2' failed.
}
```


© 2022 Nokia

C++20 Problem: ignoring fields – explicit ignoring

```cpp
template <typename T>
struct IgnoreWhenComparing {
    T value;
    constexpr bool operator==(const IgnoreWhenComparing&) const noexcept {
        return true;
    }
    constexpr auto operator<=>(const IgnoreWhenComparing&) const noexcept
    {
        return std::strong_ordering::equal;
    }
};
```

```cpp
class Quadratic {
public:
    //...
    bool operator==(const Quadratic&) const = default;
    auto operator<=>(const Quadratic&) const = default;
    //...
private:
    // a*x*x + b*x + c
    double a, b, c;
    // cache for f(x)
    using CachedResults = std::unordered_map<double, double>;
    mutable IgnoreWhenComparing<CachedResults> cachedResults;
};
```


© 2022 Nokia

C++20 Problem: ignoring fields – grouping key fields

```
#include <unordered_map>
#include <compare>

struct Quadratic {
    Quadratic(double a, double b, double c) : m{a, b, c} {}
    bool operator==(const Quadratic& rhs) const {
        return m == rhs.m;
    }
    auto operator<=>(const Quadratic& rhs) const {
        return m <=> rhs.m;
    }
    //...
private:
    struct Members {
        bool operator==(const Members&) const = default;
        auto operator<=>(const Members&) const = default;

        double a, b, c;
    } m;
    using CachedResults = std::unordered_map<double, double>;
    mutable CachedResults cachedResults;
};
```


© 2022 Nokia
NOKIA

Summary: C++17 vs C++20

	straightforward	<tuple> std::tie	double apply	macros	C++20 = default;
standard	C++98	C++11	C++17	C++98	C++20
Duplications (error-prone)	**Yes**	**No**	**No**	**No**	**No**
Maintainability (missing fields)	**No**	**No**	**No**	**No**	**Yes**
Flexibility (ignoring fields)	**Yes**	**Yes**	**Yes**	**Yes**	**Not easy**
Test coverage	**Hard**	**Easy**	**Easy**	**Hard**	**Easy**
Compilation time	**Fastest**	**Slow**	**Fast**	**Fastest**	**Fast**
Optimization level for best performance	**-O1**	**-O3**	**-O2**	**-O1**	**-O1**

© 2022 Nokia

tieMembers, double-apply patterns

Have these patterns any value in C++20?

- You can use these patterns for:
  - Hashing (std::hash<t>::operator()) (in this case single-apply is enough)
  - Printing (with extra tuple for member names, extra applyNames method)
  - Adding (operator+) and other arythmethic
  - Probably many others…

© 2022 Nokia


NOKIA