C++11/14 at Scale: What Have We Learned?

Vittorio Romeo

ACCU 2021 VIRTUAL EVENT

C++11/14 at Scale

What Have We Learned?

Vittorio Romeo

vittorioromeo.info
vittorio.romeo@outlook.com
vromeo5@bloomberg.net
@supahvee1234

ACCU 2021

2021/03/12
Virtual Event

Bloomberg

Backstory


2016
Joined Bloomberg as a Software Engineer

2017

Developed in-house C++11/14 course

2018

Plan to coauthor book with John Lakos

C++ now

2018-2021

Rediscovering C++11/14

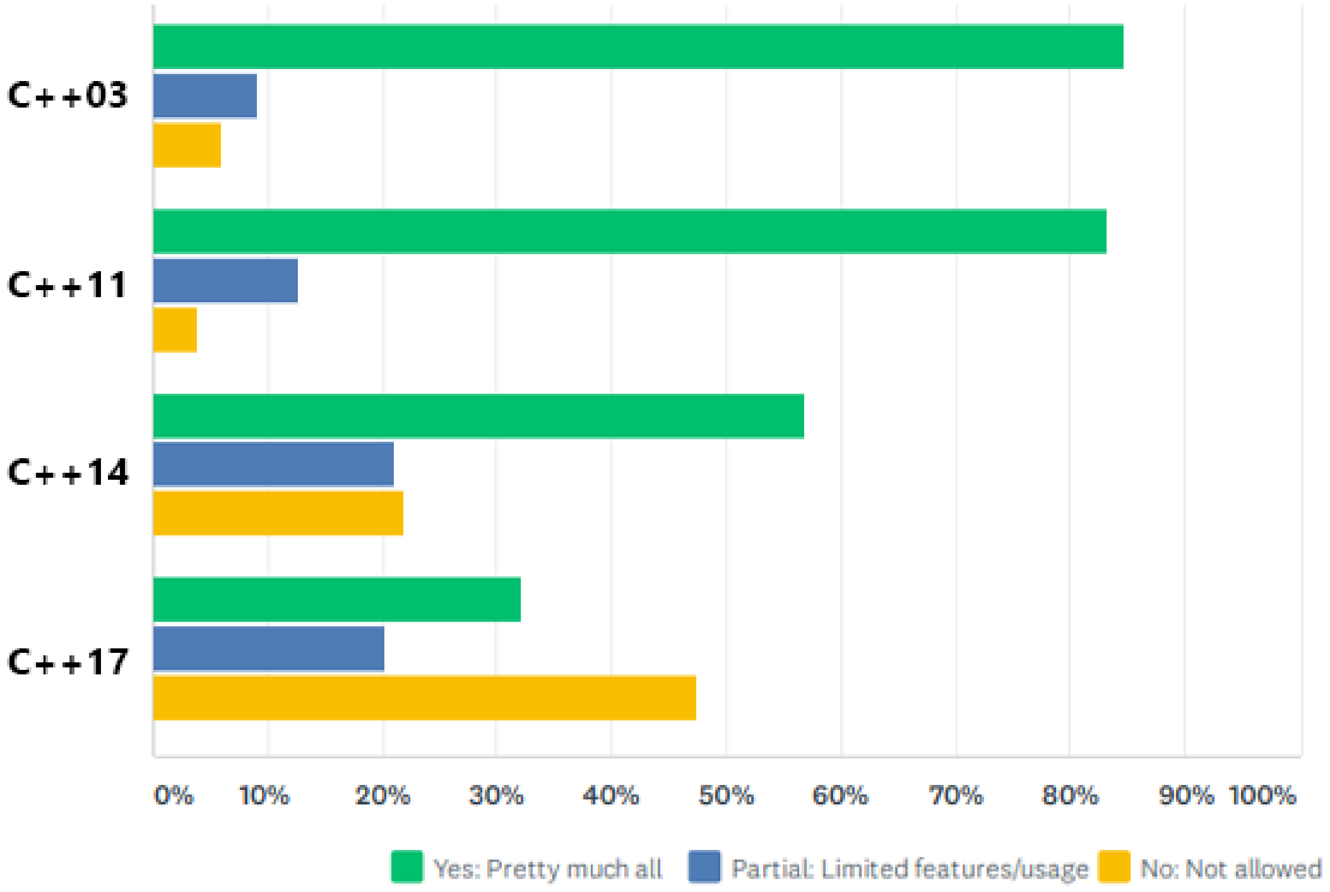
2021

“Embracing Modern C++ Safely” Release

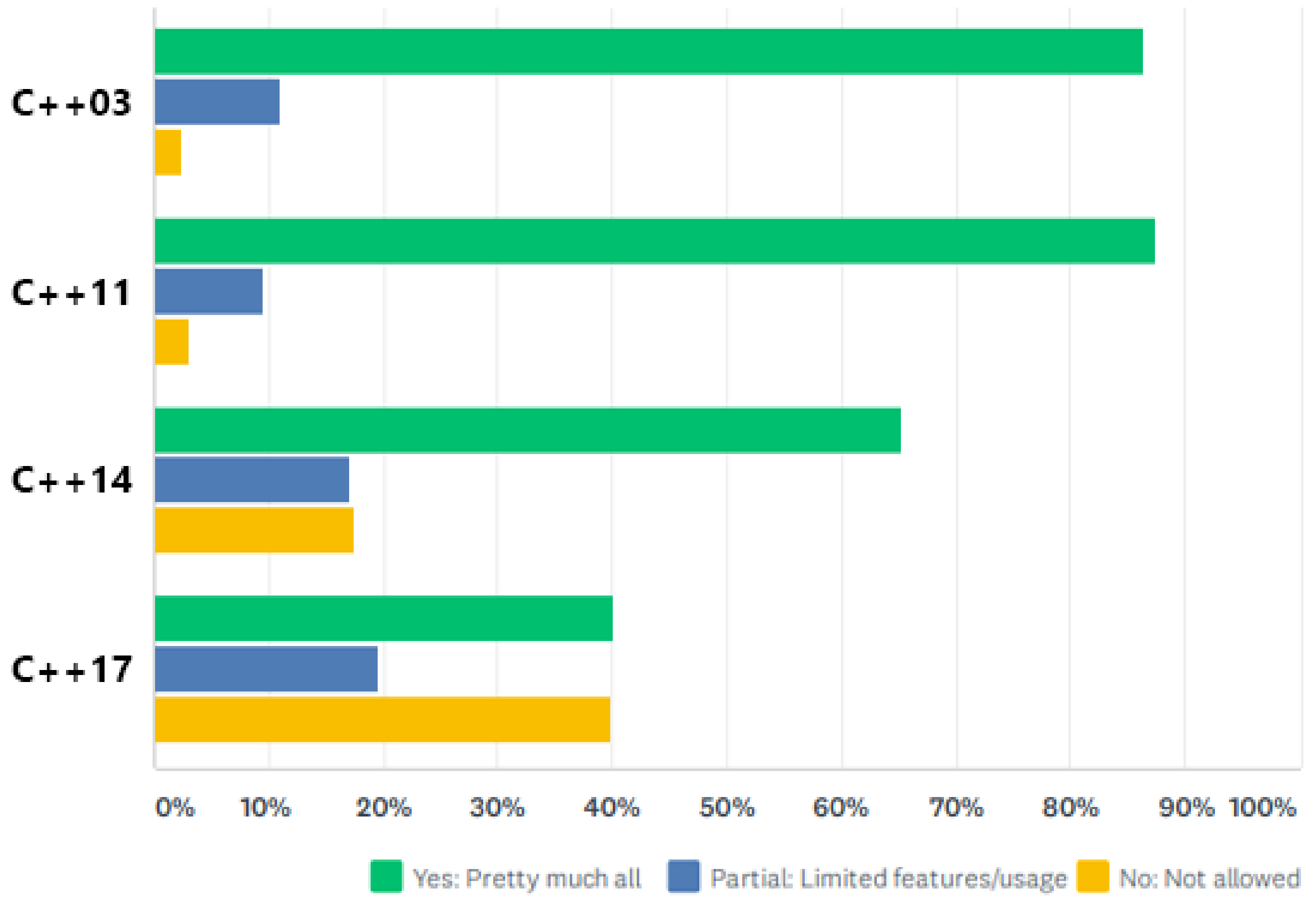
Introduction

- Why are we talking about C++11/14 in 2020?
- How C++11/14 can surprise you today
- C++ at scale
- "Safety" of a feature
- *Case study:* extended `friend` declarations

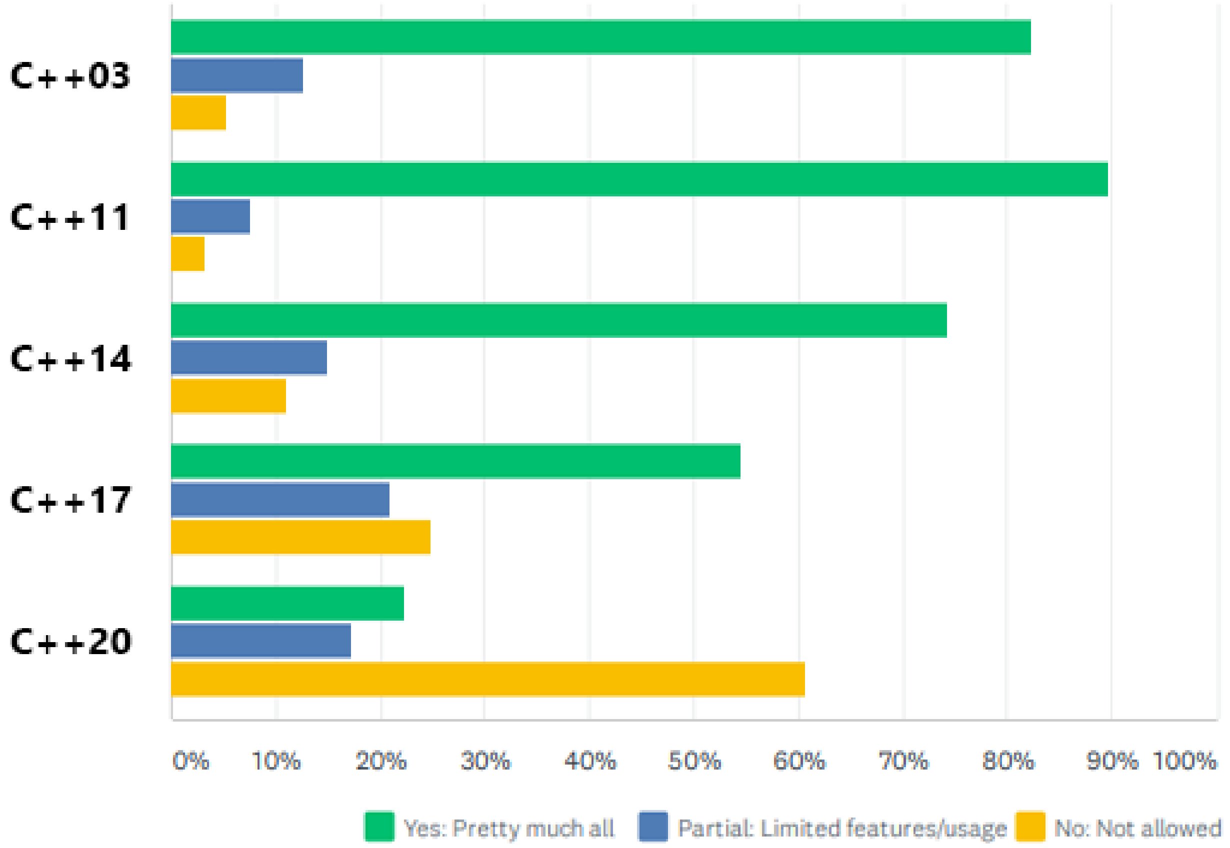
Why are we talking about C++11/14 in 2020?



- Full C++11 adoption: ~83%
- Full C++14 adoption: ~58%



- Full C++11 adoption: **~88%**
- Full C++14 adoption: **~65%**



- Full C++11 adoption: ~**90%**
- Full C++14 adoption: ~**74%**

- The results might seem good...
  - However, **~20%** of people were not fully using C++11 **in 2018**
  - And **~25%** of people were not fully using C++14 **in 2020**
  - Sample size was ~3000 in 2018, ~2000 in 2019, ~1000 in 2020

- Personal experience tells me C++11 is still a luxury in some places
  - *Example:* legacy architectures
  - People still complain online -- vocal minority?

- *"Experience is the best teacher"*
- I've been using Modern C++ since 2012
- C++11/14 more widely used in production, especially over the past ~6 years
- I've been teaching C++11/14 professionally since ~4 years

- There are great learning resources
    - But most teach "the features" rather than "the experience"
    - What looks good on paper might not work in the "real world"

How C++11/14 can surprise you today

- C++11/14 features can be unpredictable, even today
- **Q:** *What's the smallest change to the core language you can think of in C++11?*

- Hint…

> >

- Did you know that `>>` closing angle brackets can...
  - ...make a valid C++03 program ill-formed?
  - ...silently change a program's behavior?


```
template <int POWER_OF_TWO>
struct PaddedBuffer { /* ... */ };

PaddedBuffer<256 >> 4> smallBuffer;
```

- Valid prior to C++03, ill-formed since C++11
- Easy fix: wrap the right shift expression in parentheses

```
enum Outer { a = 1, b = 2, c = 3 };

template <typename>
struct S { enum Inner { a = 100, c = 102 }; };

template <int>
struct G { typedef int b; };

int main()
{
    return S<G< 0 >>::c>::b>::a;
}
```

- Valid in both C++03 and C++11, but completely different meaning!
  - C++03 returns `100`
  - C++11 returns `0`

- Unlikely to happen in practice
  - Example of something "innocent" hiding a pitfall
- How about...
  - Attributes that can make your code ill-formed NDR?
  - `extern template` not improving compilation time or code size at all?
  - Destruction order UB with Meyers Singletons?
  - Encoding of whitespace within raw string literals?
- Almost every feature has a... "dark side"

Modern C++ at scale

- What is the best way of teaching C++11/14?
    - What features should be prioritized/avoided?
- Diversity of skill and seniority
- Impact of style guides

- Age range: 21-70+
- Prior C++ experience
- Prior development experience
- Experience with other languages
- "Interest" in Modern C++
- Application/library development goals

- Companies can have thousands of engineers
- Not every company has fancy code governance tools
- A style guide is essential to promote consistency and discoverability

- Who writes the style guide?
- What is the "input" to a style guide?

"Safety" of a feature

- Every C++ feature is "safe" when used correctly...

- But what is the likelihood that it is used correctly?
- Does the feature have any "attractive nuisance"?
- What are the advantages of using a feature compared to its risks?
- Is it worth teaching to a new hire? To an experienced hire?

- From our book:

> *"The degree of safety of a given feature is the relative likelihood that widespread use of that feature will have positive impact and no adverse effect on a large software company's codebase."*

- Not an exact science
- Relies on teaching and usage experience
- Useful metric to decide what to teach or to focus on

- **Safe**: add considerable value, easy to use, hard to misuse
  - Ubiquitous adoption of such features is productive
- **Conditionally Safe**: add considerable value, but prone to misuse
  - Require in-depth training and additional care
- **Unsafe**: provide value only in the hands of an "expert", and prone to misuse
  - Wouldn't teach these as part of a general C++11/14 course
  - Require explicit training on their use cases and pitfalls

- The `override` keyword is the prime example of a **safe** feature

```
class MockConnection : Connection
{
    void connect(IPV4Address ip) override;
};
```

- Prevents bugs
- Makes code self-explanatory
- No real technical downsides

- Only pitfall: overreliance without enforcement

- Range-based `for` loops are often great... until they aren't

```
for(Combo& c : keyboardTriggerGetters[bindID]().getCombos())
{
    // ...
}
```

```
class TriggerGetter
{
public:
    std::vector<Combo> getCombos() const;
};
```

- **Q:** Any issue? Is the code above OK?


```
for(Combo& c : keyboardTriggerGetters[bindID]().getCombos())
{
    // ...
}
```

- The code above was OK for months...
- Until an "optimization" was implemented!

```
class TriggerGetter
{
    std::vector<Combo> cachedCombos;

public:
    const std::vector<Combo>& getCombos() const;
};
```


- Range-based `for` loops are a fantastic tool
- But you need to be aware of their pitfalls
- Hence, additional training is required (compared to `override`)
- This is why they are a **conditionally safe** feature

- Categorization might change in the future, see:
    - **P2012**: "Fix the range-based `for` loop"
      (N. Josuttis, V. Zverovich, F. Mulonde, A. O'Dwyer)

- `decltype(auto)` has some very important use cases
  - Yet, it is often misused without proper training and care
- *Example:* higher-order functions

```
template <typename F>
decltype(auto) logAndCall(F&& f)
{
    log("invoking function ", nameOf<F>());
    return std::forward<F>(f)();
}
```


- I used to teach `decltype(auto)` right after `auto` and `decltype`
  - Train of thought: provide a complete overview of type inference
  - Actual results: overuse of `decltype(auto)`

- Some students thought:

> If `decltype(auto)` does everything `auto` does and more, why not use it all the time?

> If `decltype(auto)` is more flexible, why not use it when I'm not sure when to choose between `auto` and `auto&`?

- In order to understand when `decltype(auto)` is appropriate, you need to:
  - Have a solid grasp on type inference and value categories
  - Be somewhat experienced and familiar with both `decltype` and `auto`
  - Have some metaprogramming experience (e.g. SFINAE)

- I couldn't find valid use cases for `decltype(auto)` in variable position
- Only real use cases are as a return type placeholder
  - And those have to be compared against a trailing return type
- `decltype(auto)` is far from trivial!

- **Safe:** attributes, `nullptr`, `static_assert`, digit separators, ...
- **Conditionally Safe:** `auto`, `constexpr`, rvalue references, ...
- **Unsafe:** `[[carries_dependency]]`, `final`, `inline namespace`, ...

	Safe	Cond. Safe	Unsafe
C++11	18	21	7
C++14	5	3	2

- Teach **safe** features *early* and *quickly*
  - Most of them are QoL improvements or hard to misuse
  - Trust your students!
- Teach **conditionally safe** features by building on top of **safe** knowledge
  - They require more time and examples
  - Show how they can backfire
  - Have exercises that make students question whether to use a feature or not
- Leave a subset of **unsafe** features for self-contained CE courses
  - E.g. "*Library API and ABI version with* `inline` *namespaces*"

Case study: extended `friend` declarations

- Prior to C++11, `friend` declarations require an *elaborated type specifier*
  - Syntactical element having the form `<class|struct|union> <identifier>`

```
struct S;

struct Example
{
    friend class S;             // OK
    friend class NonExistent;   // OK
};
```

- This restriction prevents other entities to be designated as friends
  - E.g. type aliases, template parameters

```
using WindowManager = UnixWindowManager;

template <typename T>
struct Example
{
    friend class WindowManager;  // Error
    friend class T;              // Error
};
```


- Use of C++03 `friend` can sometimes be surprising

```
struct S; // This S resides in the global namespace

namespace ns
{
    class X3
    {
        friend struct S;
            // OK, declares a new `ns::S` instead of referring to `::S`
    };
}
```

- C++11 extended `friend` declarations lift all the aformentioned limitations

```
struct S;
typedef S SAlias;

namespace ns
{
    template <typename T>
    struct X4
    {
        friend T;           // OK, refers to template parameter
        friend S;           // OK, refers to `::S`
        friend SAlias;      // OK, refers to `::S`
        friend decltype(0); // OK, equivalent to `friend int;`
        friend C;           // Error, `C` does not name a type.
    };
}
```


- However, we categorize this feature as **unsafe** -- why?
  - It is rarely useful in practice, like C++03 `friend`
  - Promotes *long-distance friendship* (!)

- When a type `X` befriends a type `Y` which lives in a separate component...
  - `X` and `Y` cannot be thoroughly tested independently anymore
  - Physical coupling occurs between `X` and `Y`'s components
  - Possible physical design cycles can happen

- However, even an **unsafe** feature can have some compelling use cases
  - For example, avoiding typos

```
struct Container;

struct ContainerIterator
{
    friend class Contianer;
        // Whoops, compiles!
};
```

```
struct Container;

struct ContainerIterator
{
    friend Contianer;
        // Error, no such type!
};
```

- Other interesting use cases: type alias customization points, `PassKey` idiom, ...
  - However, let's focus on CRTP

- CRTP stands for "curiously recurring template pattern"

```
template <typename T>
class Base
{
    // ...
};

class Derived : public Base<Derived>
{
    // ...
};
```

- `Base` knows who derives from it, thanks to `T`
- Useful to implement *mixins* and factor out copy-pasted code

- *Example use case:* instance counter

```
class A
{
    static int s_count; // declaration
    // ...

public:
    static int count() { return s_count; }

    A()           { ++s_count; }
    A(const A&)   { ++s_count; }
    A(const A&&)  { ++s_count; }
    ~A()          { --s_count; }
};
```

```
int A::s_count;  // definition (in .cpp file)
```


- Factor out the counter, using `protected` access specifier

```
template <typename T>
class InstanceCounter
{
protected:
    static int s_count; // declaration

public:
    static int count() { return s_count; }
};

template <typename T>
int InstanceCounter<T>::s_count; // definition (in the same file)
```

- Let's use it!

```
struct A : InstanceCounter<A>
{
    A() { ++s_count; }
};

struct B : InstanceCounter<A>
{
    B() { ++s_count; }
};
```

- **Q:** Any issue?

- Also, a class further down the hierarchy tree could mess with `s_count`

```
struct AA : A
{
    AA() { s_count = -1; } // Oops! *Hyrum's Law* is at work again!
};
```

- We'd like to prevent mistakes and hijacking of the counter
  - Turns out, extended `friend` declarations solve both issues!

- Turn `s_count` from `protected` into `private`
- Befriend `T`

```
template <typename T>
class InstanceCounter
{
    static int s_count; // Make this static data member `private`.
    friend T;           // Allow access only from the derived `T`.

public:
    static int count() { return s_count; }
};
```



```
struct B : InstanceCounter<A>
{
    B() { ++s_count; }
        // error: 's_count' is private within this context
};
```

```
struct AA : A
{
    AA() { s_count = -1; }
        // error: 's_count' is private within this context
};
```

- Extended `friend` declarations seem of limited use at first
- They also promote bad design (physical coupling, long-distance friendship)
- However, they have some nice properties
    - Avoidance of typos/mistakes
    - Great synergy with CRTP

- Due to their niche nature, we categorize them as **unsafe**
    - Significant training and experience is required to avoid misuse

Conclusion

- C++11/14 at scale are still an open research area
  - "Human cost" of a feature is not easy to quantify
- Categorizing features by "safety" helps with devising learning paths
  - For productivity and stability, it is important to prioritize what to teach
- All features have good use cases and nasty pitfalls
  - "*Knowledge is power*"

- **"Embracing Modern C++ Safely"**
    - John Lakos
    - Vittorio Romeo
    - Rostislav Klebnikov
    - Alisdair Meredith
    - ...and many others
- Out in Q2 2021 -- emcpps.com
- Follow me on Twitter for updates: @supahvee1234

Thanks!

https://vittorioromeo.info

https://github.com/SuperV1234/accu2021

vittorio.romeo@outlook.com

vromeo5@bloomberg.net

@supahvee1234

Bloomberg

Chapter 1 Safe Features

1.1 C++11

- Attribute Syntax
- Consecutive >s
- **decltype**
- Defaulted Functions
- Delegating Ctors
- Deleted Functions
- **explicit** Operators
- Function **static** '11
- Local Types '11
- **long long**
- **noreturn**
- **nullptr**
- **override**
- Raw String Literals
- **static_assert**
- Trailing Return
- Unicode Literals
- **using** Aliases

1.2 C++14

- Aggregate Init '14
- Binary Literals
- **deprecated**
- Digit Separators
- Variable Templates

Chapter 2 Conditionally Safe Features

2.1 C++11

- **alignas**
- **alignof**
- **auto** Variables
- Braced Init
- **constexpr** Functions
- **constexpr** Variables
- Default Member Init
- **enum class**
- **extern template**
- Forwarding References
- Generalized PODs '11
- Inheriting Ctors
- **initializer_list**
- Lambdas
- **noexcept** Operator
- Opaque **enums**
- Range **for**
- *rvalue* References
- Underlying Type '11
- User-Defined Literals
- Variadic Templates

2.2 C++14

- **constexpr** Functions '14
- Generic Lambdas
- Lambda Captures

Chapter 3 Unsafe Features

3.1 C++11

- **carries_dependency**
- **final**
- **inline namespace**
- **noexcept** Specifier
- Ref-Qualifiers
- **union** '11
- **friend** '11

3.2 C++14

- **decltype(auto)**
- Deduced Return Type

emcpps.com