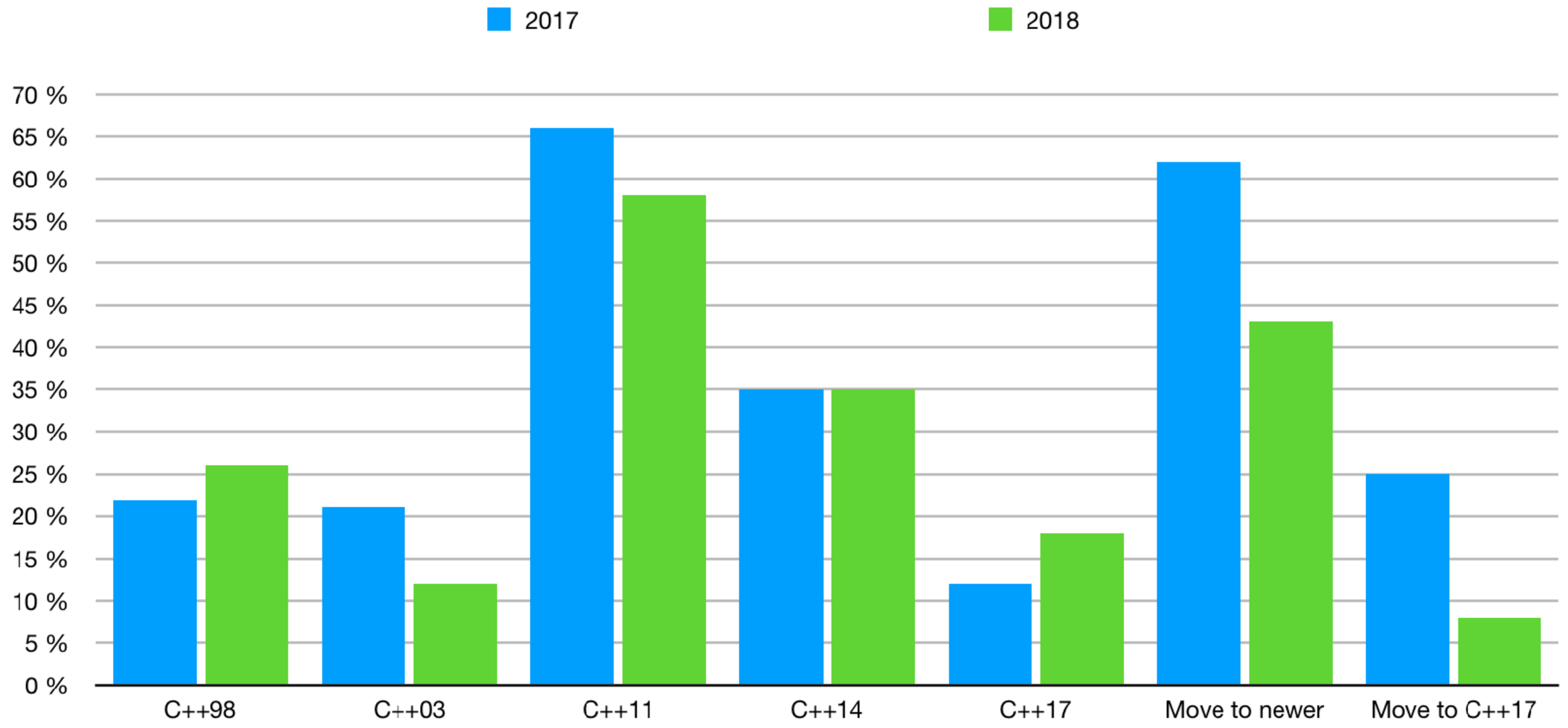
Debug C++ Without Running (Thoughtful Reading of C++ code)

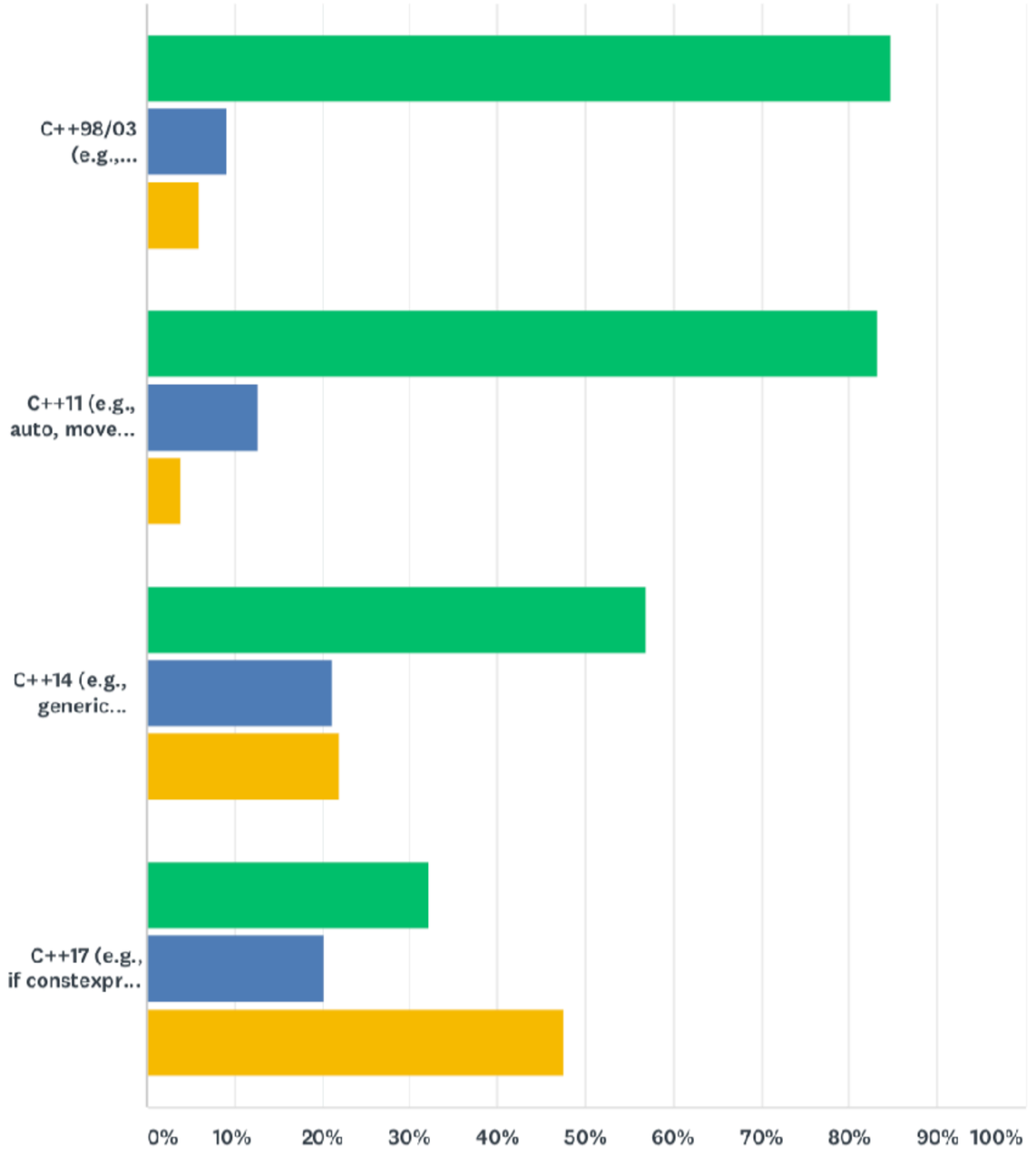
Anastasia Kazakova
JetBrains
@anastasiak2512
C++ on Sea 2019

JetBrains Dev Ecosystem survey 2017/2018

https://www.jetbrains.com/research/devecosystem-2019 – Up & Running!



C++ Foundation Developer Survey 2018



How do C++ developer feel about the language?

1. “C++ developers must suffer!”

   (someone at ACCU 2018, JB booth)

2. “Cognitive overhead”

   (in comments to Sean Parent "Modern" C++ Ruminations)

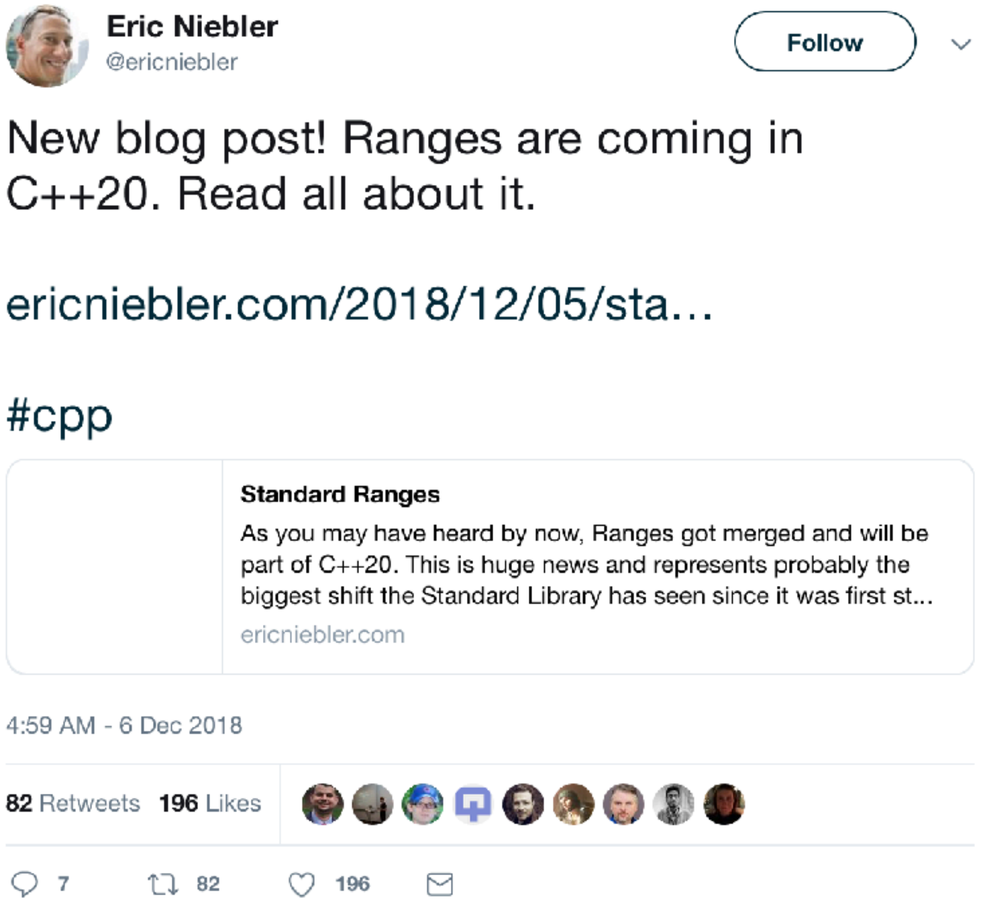
3. “Need naive simple and debuggable code”

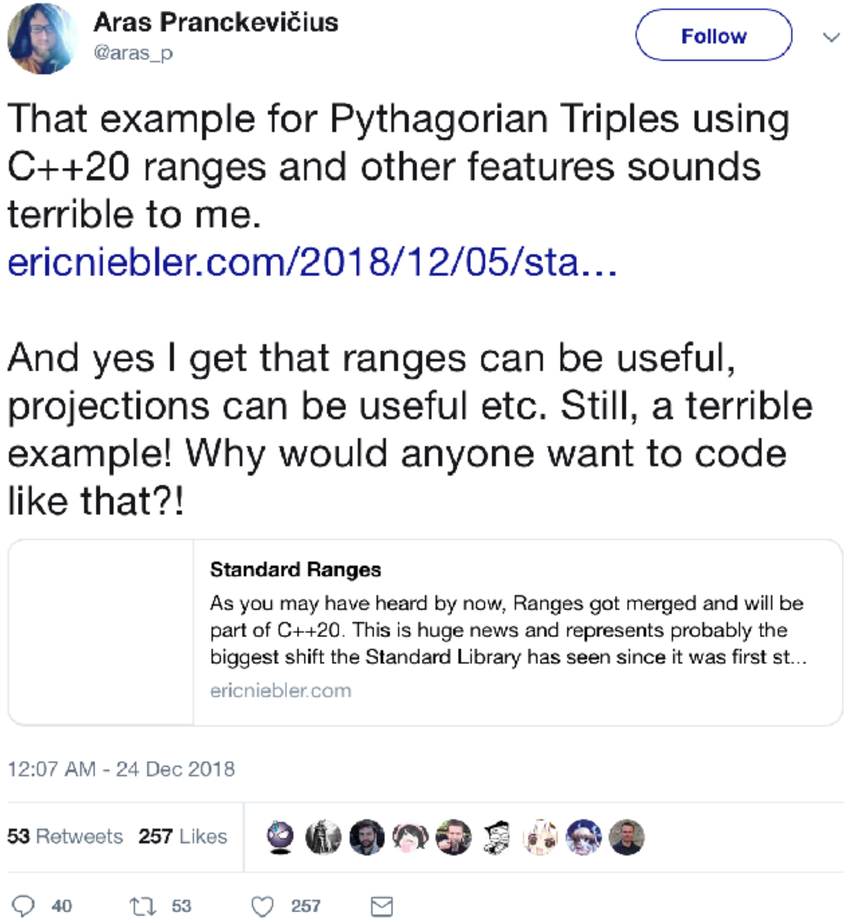
   (from comments to @aras_p tweet)

4. “Towards a more powerful and simpler C++”

   (Herb Sutter’s direction)

Raging at ranges





Raging at ranges

Eric Smolikowski @esmolikowski · 24 Dec 2018

Replying to @aras_p

The way c++ is changing is scary. I liked how it was object oriented while still gave the C system programming at its core, but it's now becoming a mash of unrecognizable gibberish...

1 9

1 more reply

Mike Nicolella @MikeNicolella · 24 Dec 2018

Replying to @aras_p

And worse, who would want to debug that?

1 9

1 more reply

Wojciech Muła @pshufb · 24 Dec 2018

Nobody is forced to abuse language or use its corner case. The discussed code is a great example of over-engineering. Production-level C++ code is boring.

1 2

Michael Lyashenko @ArenMook · 24 Dec 2018

Problem is, just because the "features" are there, some people will use them. If you're coding alone, all is peachy. But working in a team? 10 ways of doing 1 thing != good language.

4

10 ways of doing 1 thing

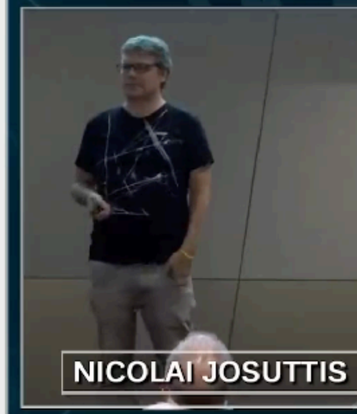
Consequences of Uniform Initialization

- Couple of ways to initialize an int:

```
int i1;                    // undefined value
int i2 = 42;               // note: inits with 42
int i3(42);                // inits with 42
int i4 = int();            // inits with 0
int i5{42};                // inits with 42
int i7{};                  // inits with 0
int i6 = {42};             // inits with 42
int i8 = {};               // inits with 0
auto i9 = 42;              // inits int with 42
auto i10{42};              // C++11: std::initializer_list<int> , C++14: int
auto i11 = {42};           // inits std::initializer_list<int> with 42
auto i12 = int{42};        // inits int with 42
int i13();                 // declares a function
int i14(7, 9);             // compile-time error
int i15 = (7, 9);          // OK, inits int with 9 (comma operator)
int i16 = int(7, 9);       // compile-time error
auto i17(7, 9);            // compile-time error
auto i18 = (7, 9);         // OK, inits int with 9 (comma operator)
auto i19 = int(7, 9);      // compile-time error
```

tialization
IT-communication.com
8
josuttis | eckstein
IT communication

cppcon | 2018
THE C++ CONFERENCE • BELLEVUE, WASHINGTON



The Nightmare of Initialization in C++

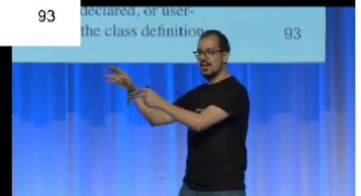
CppCon.org

	Default init	Copy init	Direct init	Value init	Direct list init	Copy list init
`Foo foo`	`;`	`= value;`	`(args);`	`();` `{};`	`{args};`	`= {args};`
Built-in types	Uninitialised	Initialised with value (via conversion sequence)	1 arg: Initialised with arg >1 arg: doesn't compile	Zero-initialised	1 arg: Initialised with arg >1 arg: doesn't compile	1 arg: Initialised with arg >1 arg: doesn't compile
Aggregates	Uninitialised	Doesn't compile	Doesn't compile (but will in C++20)	Aggregate init (= all elements are zero-initialised)	Aggregate init	Aggregate init
Types with std::initializer_list ctor	Default ctor	Non-explicit ctor matching value type (via conversion sequence)	Matching ctor	Default ctor if there is one, otherwise std::initializer_list ctor	std::initializer_list ctor if there is one, otherwise matching ctor	std::initializer_list ctor if there is one, otherwise matching ctor
Other types with no user-provided* default ctor	Members are default initialised	Non-explicit ctor matching value type (via conversion sequence)	Matching ctor	Members are zero-initialized	Matching ctor	1 arg: copy init. >1 arg: matching constructor
Other types	Calls default ctor	Non-explicit ctor matching value type (via conversion sequence)	Matching ctor	Default ctor	Matching ctor	1 arg: copy init. >1 arg: matching constructor

*not user-provided = not user-declared, or user-declared as `=default` *inside* the class definition

93

Meeting C++ 2018
Timur Doumler
Initialization in modern C++



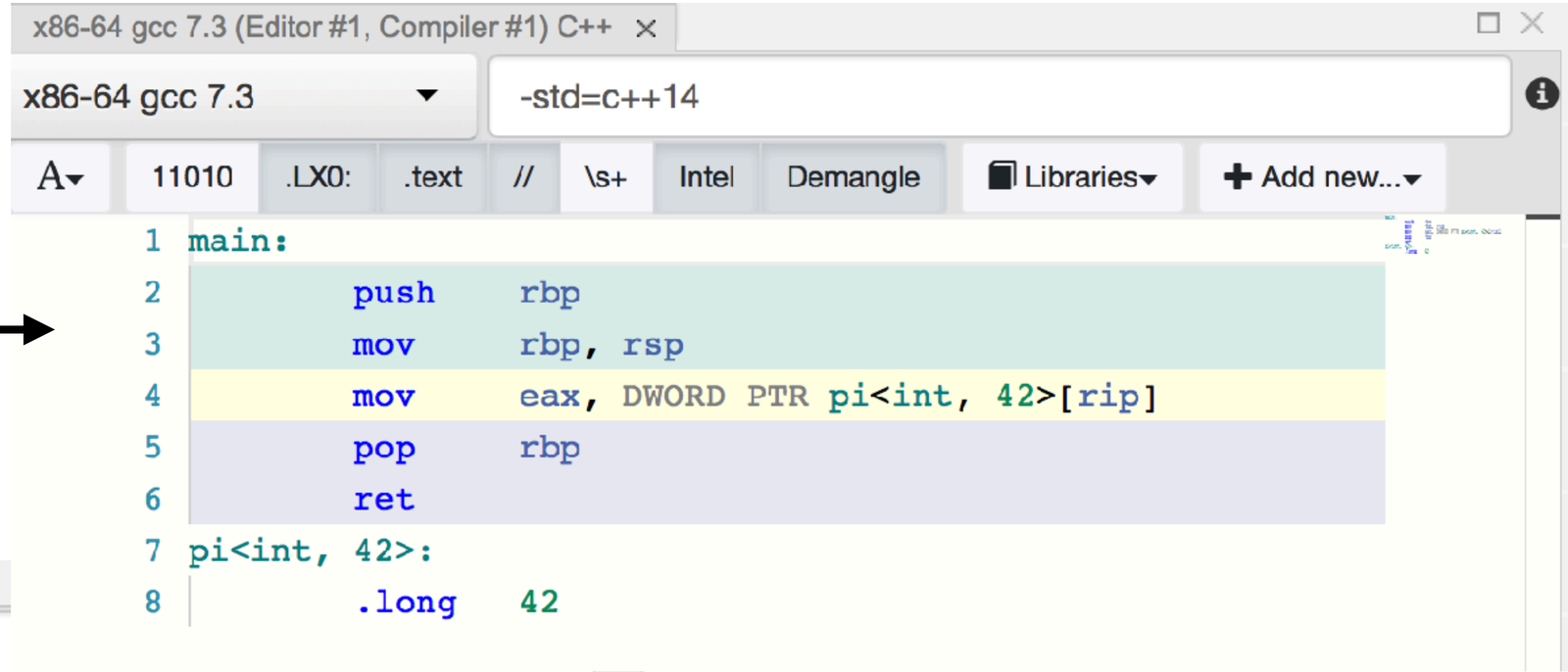
← Tomorrow, 14.45

10 ways of doing 1 thing: 42 sample

```
template<class T, int ... X>
T pi(T(X...));

int main() {
    return pi<int, 42>;
}
```

42 sample

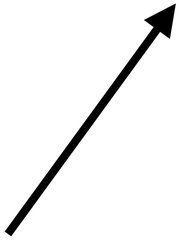
```
template<class T, int ... X>
T pi(T(X...));

int main() {
    return pi<int, 42>;
}
```



42 sample

```cpp
template<class T, int ... X>
T pi(T(X...));

int main() {
    return pi<int, 42>;
}
```

```cpp
template<class T, int ... X>
T pi = T(X...);

int main() {
    return pi<int, 42>;
}
```



```cpp
int main() {
    return int(42);
}
```

```cpp
int main() {
    return 42;
}
```

Reading macro sample

```
#define X(a) myVal_##a,
enum myShinyEnum {
#include "xmacro.txt"
};
#undef X

void foo(myShinyEnum en) {
    switch (en) {
        case myVal_a:break;
        case myVal_b:break;
        case myVal_c:break;
        case myVal_d:break;
    }
}
```

```
//xmacro.txt

X(a)
X(b)
X(c)
X(d)
```

Reading macro sample

```
#define MAGIC 100
#define CALL_DEF(val, class_name) int call_##class_name() \
                                      { return val; }

#define CLASS_DEF(class_name) class class_##class_name { \
                              public: \
                                  int count_##class_name; \
                                  CALL_DEF(MAGIC, class_name) \
                              };

CLASS_DEF(A)
CLASS_DEF(B)
CLASS_DEF(C)
```

Context sample

```
//foo.h
#ifdef MAGIC
template<int>
struct x {
    x(int i) { }
};
#else
int x = 100;
#endif
```

```
//foo.cpp
#include "foo.h"
void test(int y) {
    const int a = 100;

    auto k = x<a>(0);
}
```

Function overloads

```
void foo() { std::cout << "1\n"; }
void foo(int) { std::cout << "2\n"; }
template<typename T> void foo(T) { std::cout << "3\n"; }
template<> void foo(int) { std::cout << "4\n"; }
template<typename T> void foo(T*) { std::cout << "5\n"; }
struct S {};
void foo(S) { std::cout << "6\n"; }
struct ConvertibleToInt {ConvertibleToInt(int); };
void foo(ConvertibleToInt) { std::cout << "7\n"; }
namespace N {
    namespace M { void foo(char) { std::cout << "8\n"; } }
    void foo(double) { std::cout << "9\n"; }
}

int main() {
    foo(1);

    using namespace N::M;
    foo(1);
}
```

Operator overloads

```
class Fraction {…};

std::ostream& operator<<(std::ostream& out, const Fraction& f){…}

bool operator==(const Fraction& lhs, const Fraction& rhs){…}

bool operator!=(const Fraction& lhs, const Fraction& rhs){…}

Fraction operator*(Fraction lhs, const Fraction& rhs){…}

void fraction_sample()
{
    Fraction f1(3, 8), f2(1, 2);

    std::cout << f1 << " * " << f2 << " = " << f1 * f2 << '\n';
}
```

Compile-time generation & metaclasses

```
$class interface {
    constexpr {
        compiler.require($interface.variables().empty(),
                         "interfaces may not contain data");
        for... (auto f : $interface.functions()) {
            compiler.require(!f.is_copy() && !f.is_move(),
                "interfaces may not copy or move; consider a"
                " virtual clone() instead");
            if (!f.has_access()) f.make_public();
            compiler.require(f.is_public(),
                "interface functions must be public");
            f.make_pure_virtual();
        }
    }
    virtual ~interface() noexcept { }
};
```

```
interface Shape {
    int area() const;
    void scale_by(double factor);
};
```

```
struct Shape {
    virtual int area() const = 0;
    virtual void scale_by(double factor) = 0;
    virtual ~Shape() noexcept {
    }
}
```

Understanding the code

—

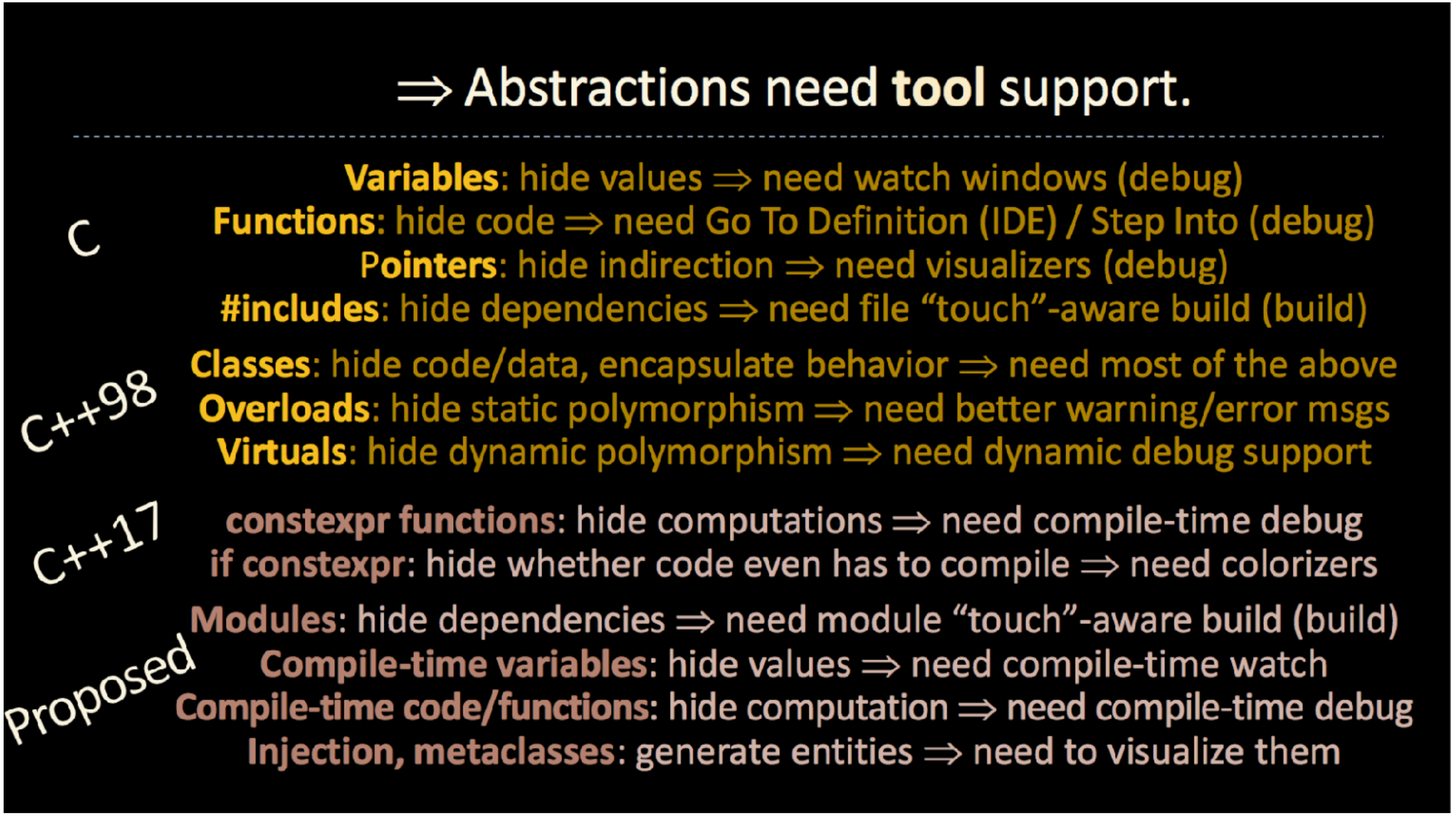
- Bug: Run / Debug / Test / Static Analysis
  - Long compilation time
  - Deploy required
  - Target platform is different
  - Incomplete code
- Flow in logic, bad design
- Code knowledge

Herb Sutter’s keynotes CppCon’17

Meta - Thoughts on Generative C++

- Abstractions are hiders
- Abstractions need tool support
- Good abstractions do need to be toolable

Herb's list for toolable and debuggable C++



General tool’s goal

Help C++ to be Debuggable

The power of tools:
Macro debug

—

Goal – *understand the substitution w/o running the preprocessor*

The power of tools: Macro debug

Existing options:

- Show final replacement

```
#define MAGIC 100
#define CALL_DEF(val, class_name) int call_##class_name() { return val; }

#define CLASS_DEF(class_name) class class_##class_name { \
                              public: \
                                  int count_##class_name; \
                                  CALL_DEF(MAGIC, class_name) \
                              };

CLASS_DEF(A)
CLASS_DEF(B)
CLASS_DEF(C)
```

Declared In: MacroReplacement.cpp

Definition:

```
#define CLASS_DEF(class_name) class class_##class_name { \
                              public: \
                                  int count_##class_name; \
                                  CALL_DEF(MAGIC, class_name) \
                              };
```

Replacement:

```
class class_C {
public:
    int count_C;
    int call_C() { return 100; }
};
```

The power of tools: Macro debug

—

Existing options:

- Show final replacement
- Substitute next step

```
#define MAGIC 100
#define CALL_DEF(val, class_name) int call_##class_name() { return val; }

#define CLASS_DEF(class_name) class class_##class_name { \
                              public: \
                                  int count_##class_name; \
                                  CALL_DEF(MAGIC, class_name) \
                              };

class class_A { public: int count_A; CALL_DEF(MAGIC, A) };
CLASS_DEF(B)
CLASS_DEF(C)
```

The power of tools: Macro debug

—

Existing options:

- Show final replacement
- Substitute next step
- Substitute all steps

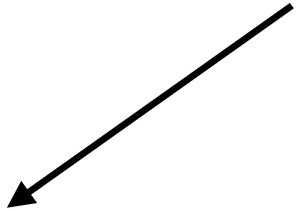
```
#define MAGIC 100
#define CALL_DEF(val, class_name) int call_##class_name() { return val; }

#define CLASS_DEF(class_name) class class_##class_name { \
                              public: \
                                  int count_##class_name; \
                                  CALL_DEF(MAGIC, class_name) \
                              };

class class_A { public: int count_A; int call_A() { return 100; } };
CLASS_DEF(B)
CLASS_DEF(C)
```

The power of tools: Macro debug

—

Substitute macro – practical sample

```
#define DECL(z, n, text) text ## n = n;

BOOST_PP_REPEAT(5, DECL, int x)
```



```
#define DECL(z, n, text) text ## n = n;

BOOST_PP_CAT(BOOST_PP_REPEAT_, BOOST_PP_AUTO_REC(BOOST_PP_REPEAT_P, 4))(5, DECL, int x)
```

```
#define DECL(z, n, text) text ## n = n;

int x0 = 0; int x1 = 1; int x2 = 2; int x3 = 3; int x4 = 4;
```

The power of tools: Macro debug

—

Be careful!

Code might be affected!

```
static int v;

#define __NEW_VAR(name, num) static void *__v_##num = (void *)&name
#define _NEW_VAR(name, num) __NEW_VAR(name, num)
#define NEW_VAR(name) _NEW_VAR(name, __COUNTER__)

void counter_macro_sample() {
    NEW_VAR(v);
    NEW_VAR(v);
    NEW_VAR(v);
}
```

The power of tools: Macro debug

—

Be careful!

Code might be affected!

```
static int v;

#define __NEW_VAR(name, num) static void *__v_##num = (void *)&name
#define _NEW_VAR(name, num) __NEW_VAR(name, num)
#define NEW_VAR(name) _NEW_VAR(name, __COUNTER__)

void counter_macro_sample() {
    NEW_VAR(v);
    static void *__v_1 = (void *)&v;
    NEW_VAR(v);
}
```

The power of tools: Macro debug

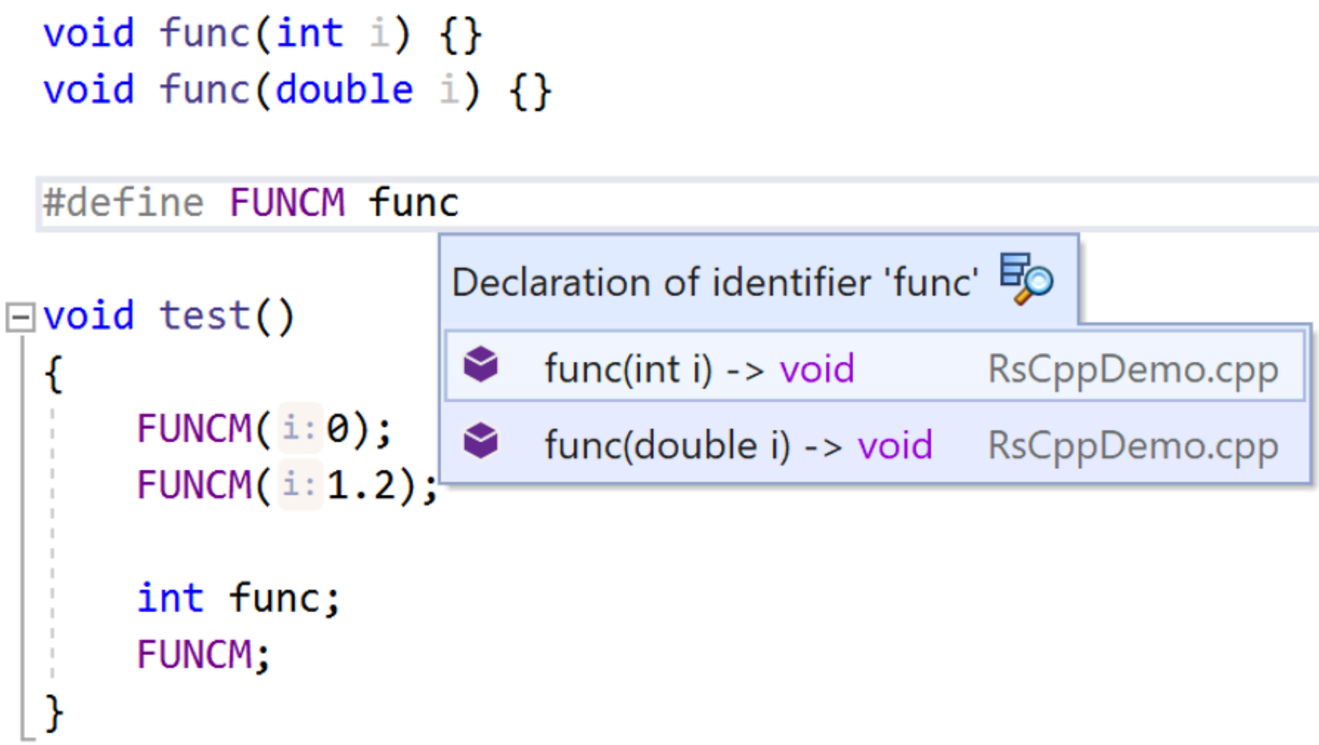
—

Macro debug requires ***all usages*** analysis!

```
void func(int i) {}
void func(double i) {}

#define FUNCM func

void test()
{
    FUNCM( i: 0);
    FUNCM( i: 1.2);

    int func;
    FUNCM;
}
```

The power of tools: Macro debug

Macro debug requires ***all usages*** analysis!



The power of tools:
Type info debug

—

Goal – *understand the final type*

The power of tools: Type info debug

—

Existing options:

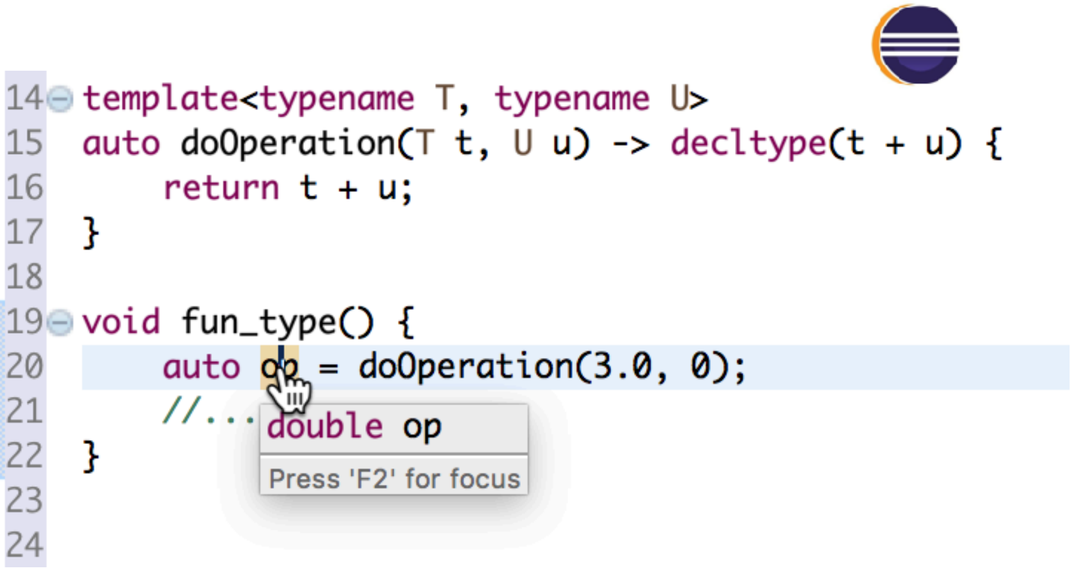
- Show inferred type

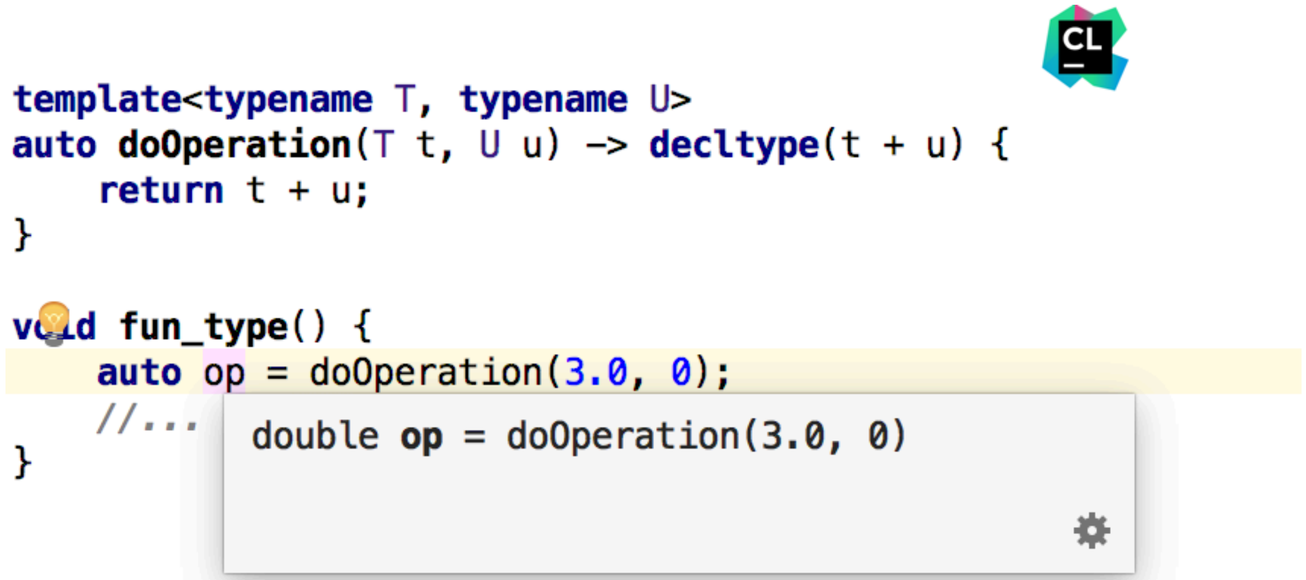
```
template<typename T, typename U>
auto doOperation(T t, U u) -> decltype(t + u) {
    return t + u;
}

void fun_type() {
    auto op = doOperation(3.0, 0);
    //...
}
```

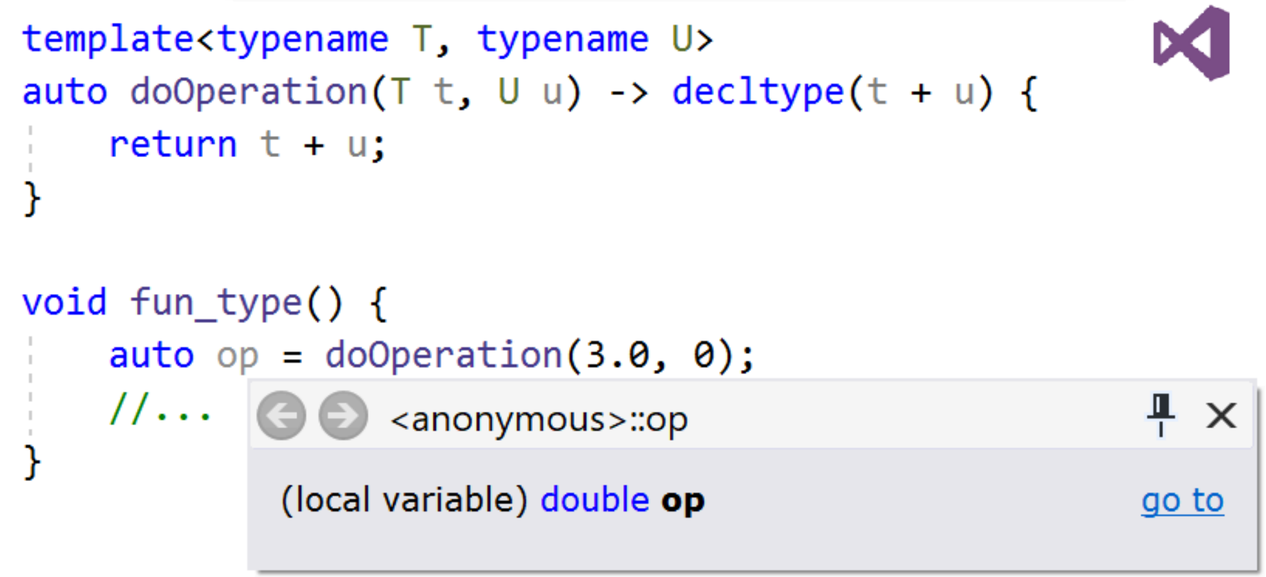
The power of tools: Type info debug

Existing options:

- Show inferred type







The power of tools: Type info debug

Existing options:

- Show inferred type
- Substitute typedef (one step)

```
#define MY_STRUCT(name) struct name {};

MY_STRUCT(A)
MY_STRUCT(B)
MY_STRUCT(C)
MY_STRUCT(D)
MY_STRUCT(E)

typedef boost::mpl::vector<A, B, C, D, E> myStructVec;
boost::mpl::at_c<myStructVec, 3>::type hi;
```

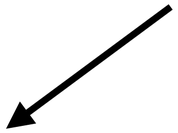
The power of tools: Type info debug

—

Existing options:

- Show inferred type
- Substitute typedef (one step)

```
#define MY_STRUCT(name) struct name {};

MY_STRUCT(A)
MY_STRUCT(B)
MY_STRUCT(C)
MY_STRUCT(D)
MY_STRUCT(E)

typedef boost::mpl::vector<A, B, C, D, E> myStructVec;
boost::mpl::at_c<myStructVec, 3>::type hi;
```



```
boost::mpl::vector5<A, B, C, D, E>::item3 hi;
```

The power of tools: Type info debug

Existing options:

- Show inferred type
- Substitute typedef (one step)
- Substitute typedef and all nested (all steps)

```
#define MY_STRUCT(name) struct name {};

MY_STRUCT(A)
MY_STRUCT(B)
MY_STRUCT(C)
MY_STRUCT(D)
MY_STRUCT(E)

typedef boost::mpl::vector<A, B, C, D, E> myStructVec;
boost::mpl::at_c<myStructVec, 3>::type hi;
```

```
boost::mpl::vector5<A, B, C, D, E>::item3 hi;
```

```
D hi;
```

The power of tools: Meta info debug

—

Debug the abstractions

- Instantiating templates

```
template<class...> struct Tuple { };
///First overload
template<class... Types>
void handle(Tuple<Types ...>) { std::cout << "1\n"; }
///Second overload
template<class T1, class... Types>
void handle(Tuple<T1, Types ...>) { std::cout << "2\n"; }
///Third overload
template<class T1, class... Types>
void handle(Tuple<T1, Types& ...>) { std::cout << "3\n"; }

void check() {
    handle(Tuple<>());              // -> 1
    handle(Tuple<int, float>());    // -> 2
    handle(Tuple<int, float&>());   // -> 3
}
```

```
///Third overload
template<class T1, class... Types>
void handle(Tuple<T1, Types& ...>) { std::cout << "3\n"; }
```

Press 'F2' for focus

handle

class T1 = int

class... Types = float

```
template<class T1 = int, class... Types>
void handle(Tuple<T1,Types...&>)
{
    std::cout << "3\n";
}
```

```
void check() {
    handle(Tuple<>());              // -> 1
    handle(Tuple<int, float>());    // -> 2
    handle(Tuple<int, float&>());   // -> 3
    h
}
```

(function) void **handle**<T1, Types...>(Tuple<T1, Types&...>)
Third overload

The power of tools: Meta info debug

—

Debug the abstractions

- Instantiating templates
- Constexpr evaluator

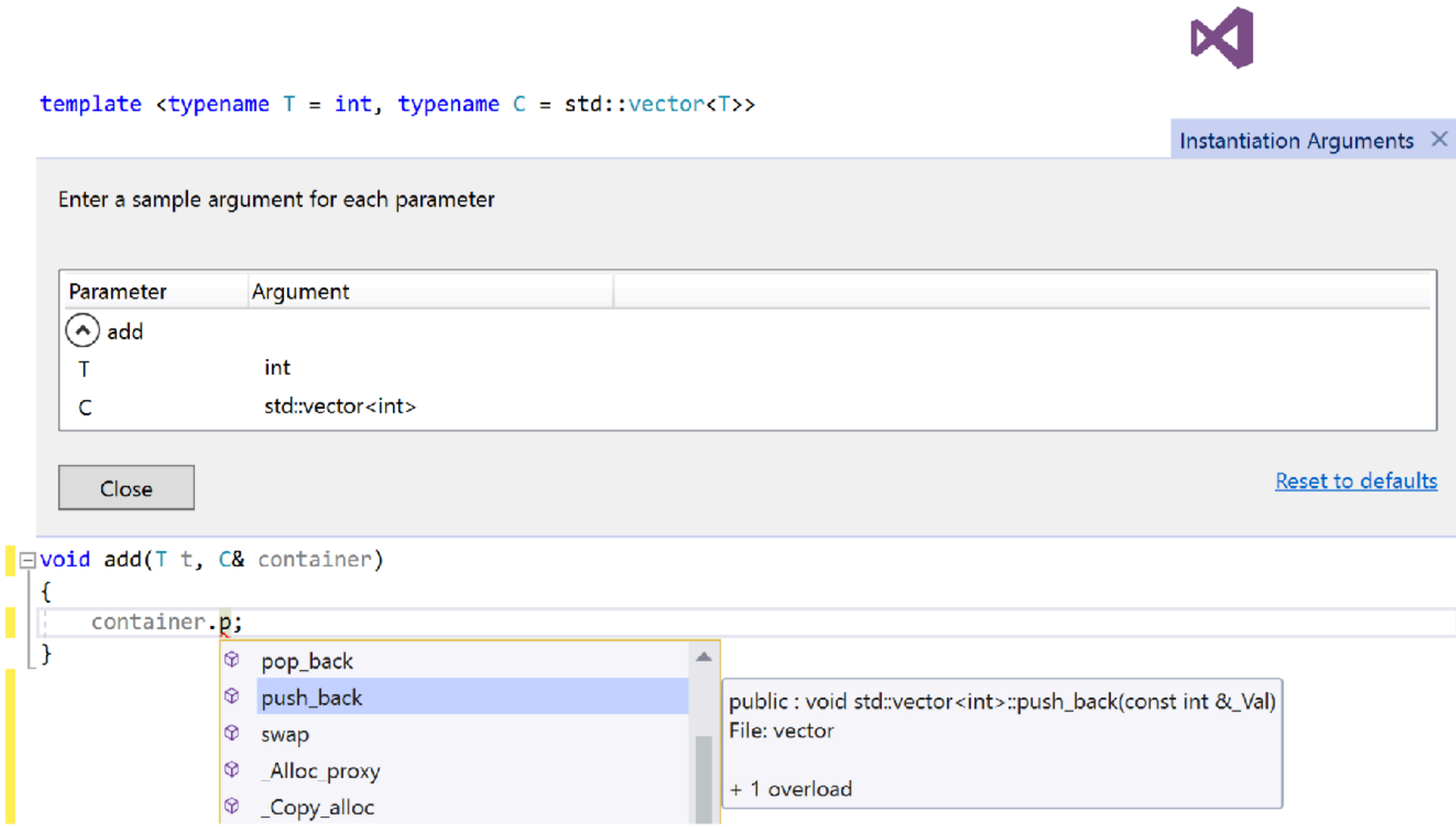
```
template <typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer<T>::value)
        return *t;
    else
        return t;
}

void test()
{
     auto pi = std::make_unique<int>(9);
     int i = 9;

     std::cout << get_value(pi.get()) << "\n";
     std::cout << get_value(i) << "\n";
}
```

The power of tools: Meta info debug

—

Debug the abstractions

- Instantiating template
- Constexpr evaluator
- Template intellisense
  - reactive

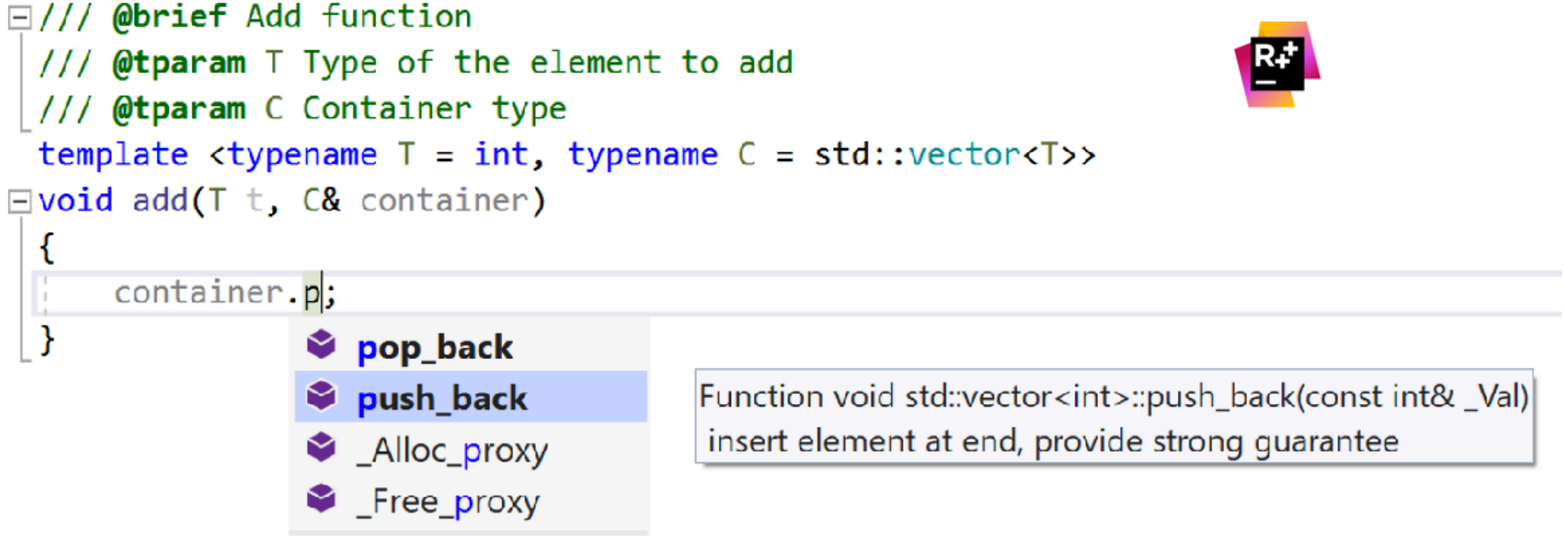


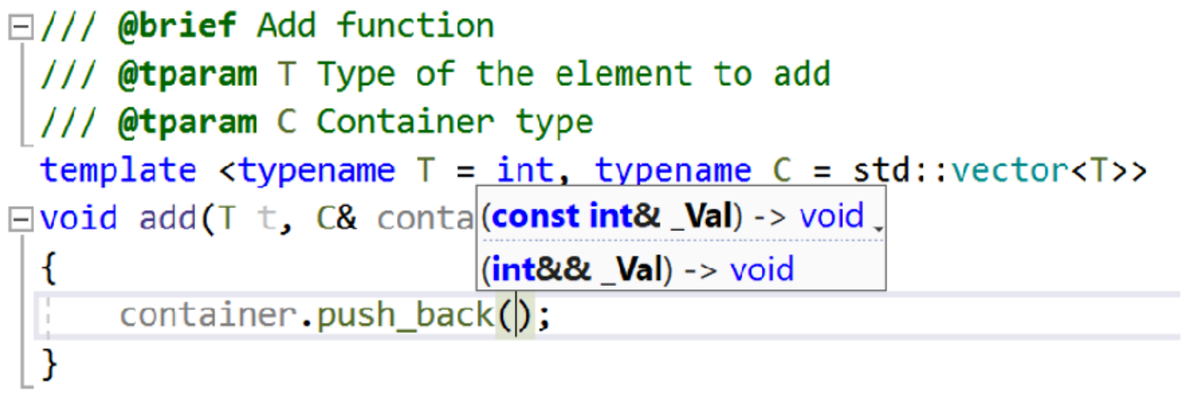
The power of tools: Meta info debug

—

Debug the abstractions

- Instantiating templates
- Constexpr evaluator
- Template intellisense
  - reactive
  - proactive





The power of tools: Overloads debug

—

Debug functions and operators overload

The power of tools: Overloads debug

—

Debug overloads:

- Distinguish overloaded operators

```
class Fraction {...};

std::ostream& operator<<(std::ostream& out, const Fraction& f)
{
    return out << f.num() << '/' << f.den() ;
}

bool operator==(const Fraction& lhs, const Fraction& rhs)
{...}

bool operator!=(const Fraction& lhs, const Fraction& rhs)
{...}

Fraction operator*(Fraction lhs, const Fraction& rhs)
{...}

void fraction_sample()
{
    Fraction f1(3, 8), f2(1, 2);

    std::cout << f1 << " * " << f2 << " = " << f1 * f2 << '\n';
}
```

The power of tools: Overloads debug

—

Debug overloads:

- Distinguish overloaded operators
- Explain overload resolution

Overload resolution:

1. Do name lookup
2. Do template argument deduction
3. Pick the candidate
4. Check access control

The power of tools: Overloads debug

—

Show candidates set via parameter info

- One-by-one or all together
- Parameters or full signature

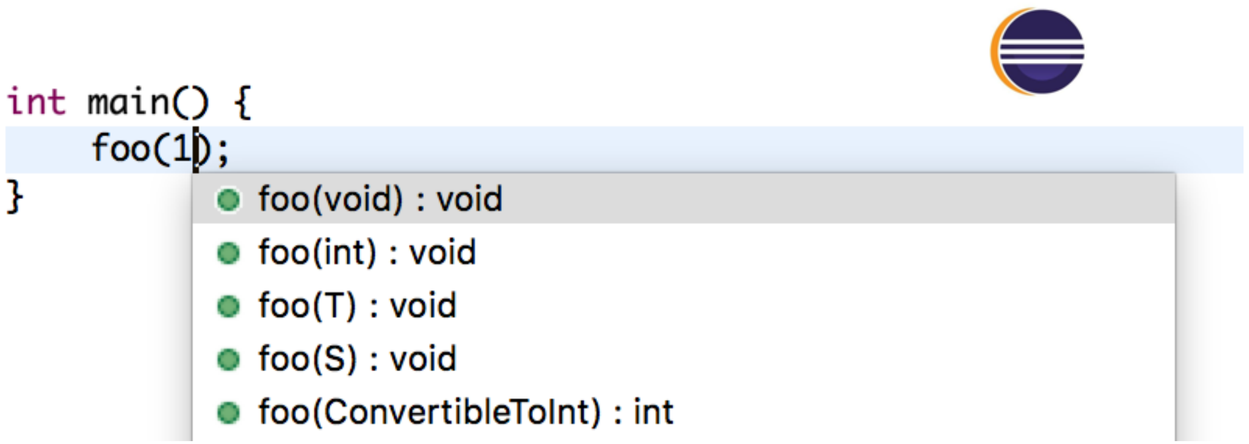

```
void foo() { std::cout << "1\n"; }
void foo(int) { std::cout << "2\n"; }
template<typename T> void foo(T) { std::cout << "3\n"; }
template<> void foo(int) { std::cout << "4\n"; }
struct S {};
void foo(S) { std::cout << "5\n"; }
struct ConvertibleToInt {ConvertibleToInt(int) {} };
int foo(ConvertibleToInt) { std::cout << "6\n"; return 0; }
namespace N {
    namespace M { void foo(char) { std::cout << "7\n"; } }
    void foo(double) { std::cout << "8\n"; }
}

void foo (int a, int b);
void foo (int a, double b);
void foo (int a, ConvertibleToInt b);

<no parameters>
int
T
S
ConvertibleToInt
int a, int b
int a, double b
int a, ConvertibleToInt b

int main() {
    foo(1);
}
```

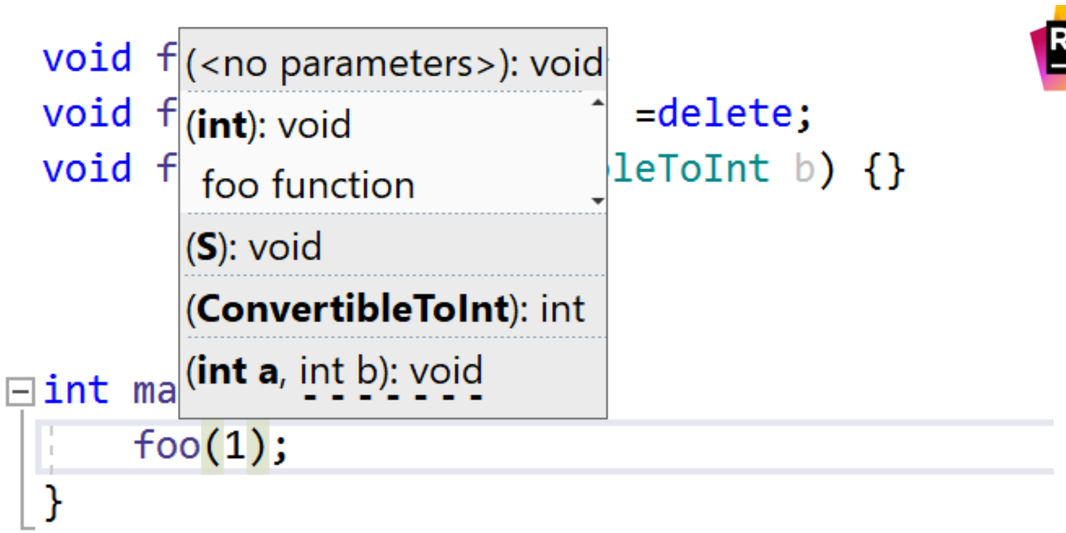
The power of tools: Overloads debug

—

Show candidates set via parameter info

- One-by-one or all together
- Parameters or full signature



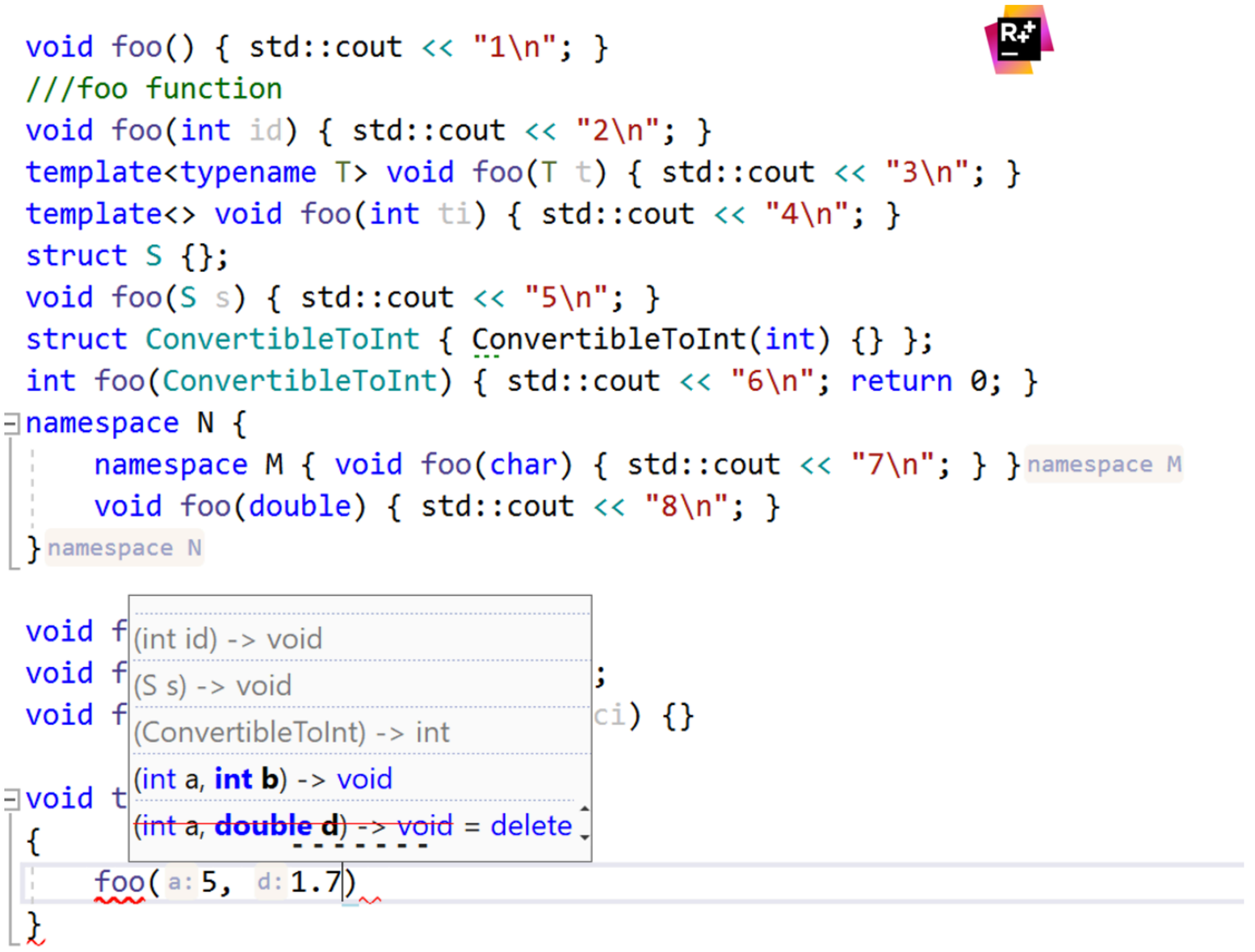


The power of tools: Overloads debug

—

Show candidates set via parameter info

- One-by-one or all together
- Parameters or full signature
- Name hints



The power of tools: Overloads debug

—

- Show candidates set
- Show explanations
  - When overload fails

```
3
4   template<typename T, typename = decltype(T().method())>
5   void bar(T);
6
7   struct X {
8       void method() {}
9   };
10
11  struct Y {};
12
13  void overload_check_3(X x, Y y) {
14      bar(x);
15      bar(y);
16  }
17
18
19
20  }
```

No matching function for call to 'bar'
candidate template ignored: substitution failure [with T = Y]: no member named 'method' in 'Y'

```
template<typename T, typename U,
    typename = std::enable_if_t<
        std::conjunction_v<std::is_integral<T>,
        std::is_floating_point<U>>
    >>
void do_stuff(T, U) {}

void overload_check_5() {
    do_stuff(1, 2);
}
```

Substitution failed: requirement 'is_floating_point<int>::value' is not satisfied

The power of tools: Overloads debug

—

- Show candidates set
- Show explanations
  - When overload fails
  - When overload works

The power of tools: Overloads debug

—

Debug overloads:

- Distinguish overloaded operators
- Explain overload resolution
- Navigate to similar/unmatched functions

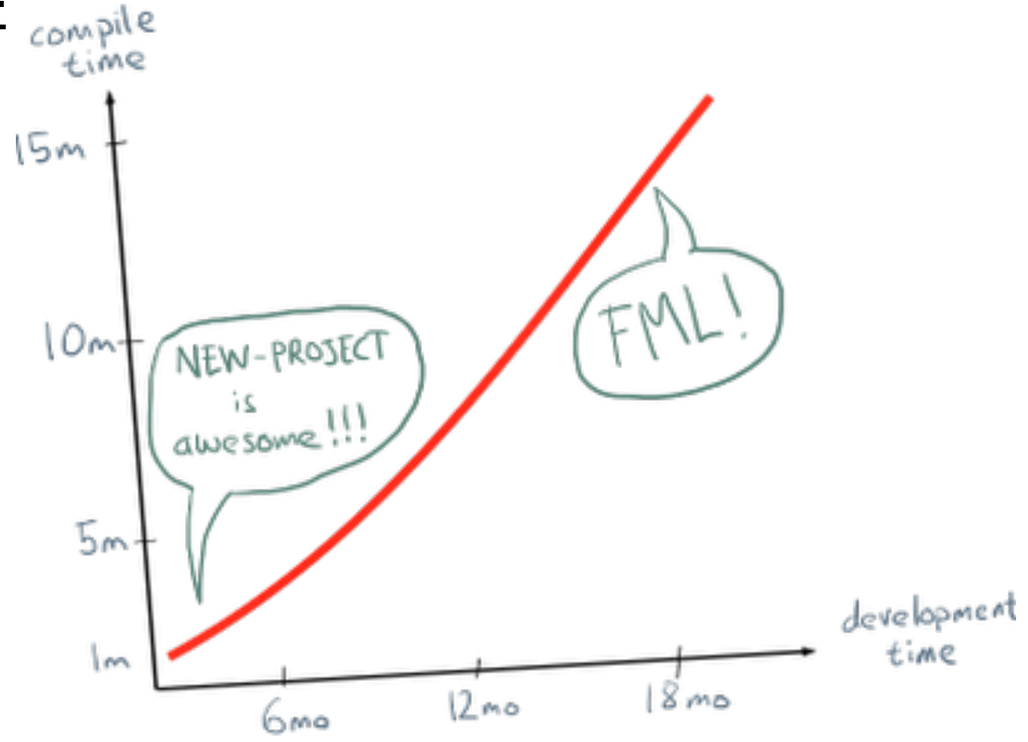
The power of tools: Overloads debug

The power of tools: Includes Header Hero

—

“Once an #include has been added, it stays”

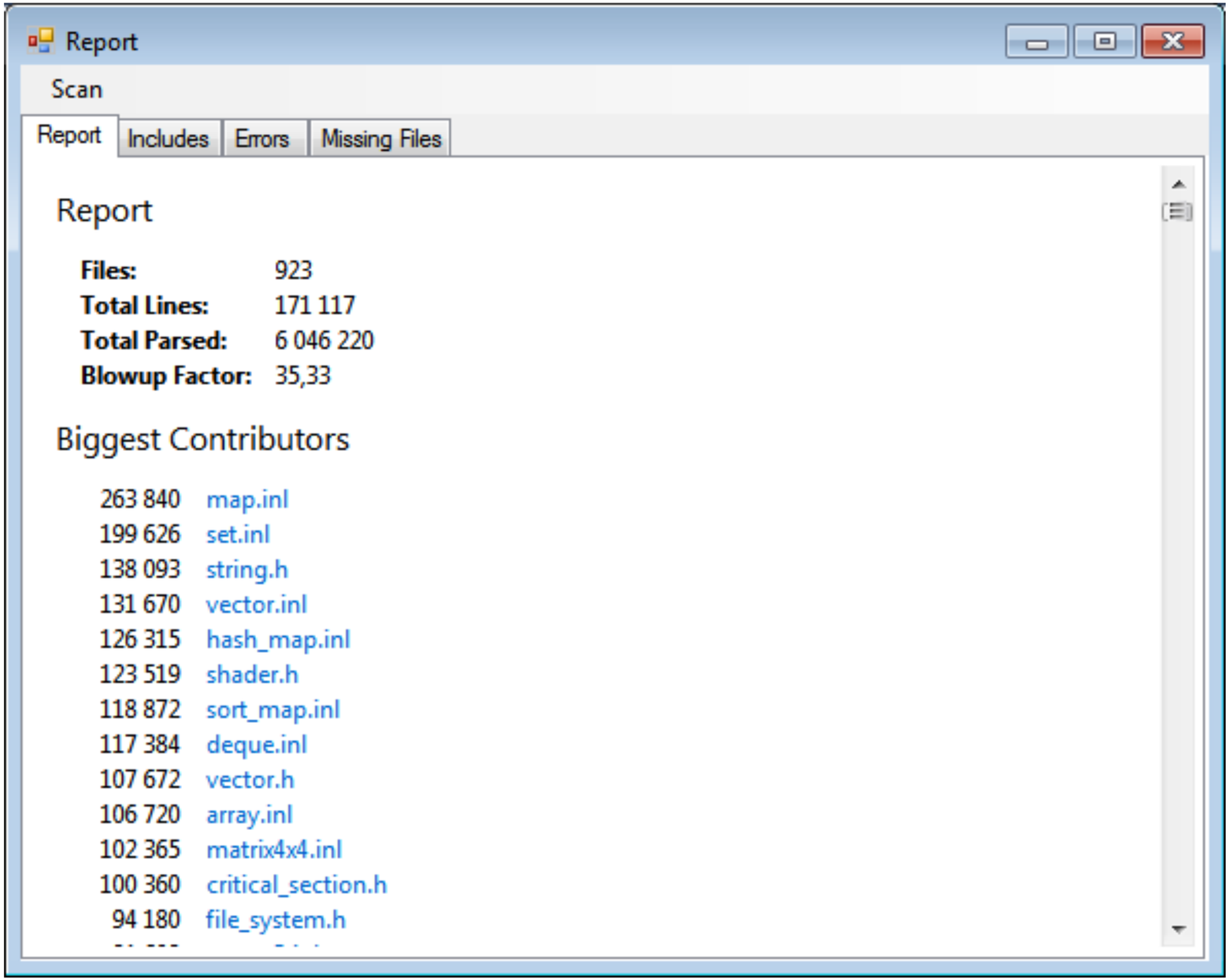
(http://bitsquid.blogspot.co.uk/2011/10/caring-by-sharing-header-hero.html)



The power of tools: Includes Header Hero

—

Blowup factor =
total parsed / total lines



The power of tools: Includes
Includes profiler

Includes profile of solution 'debuggerext'

Type to search

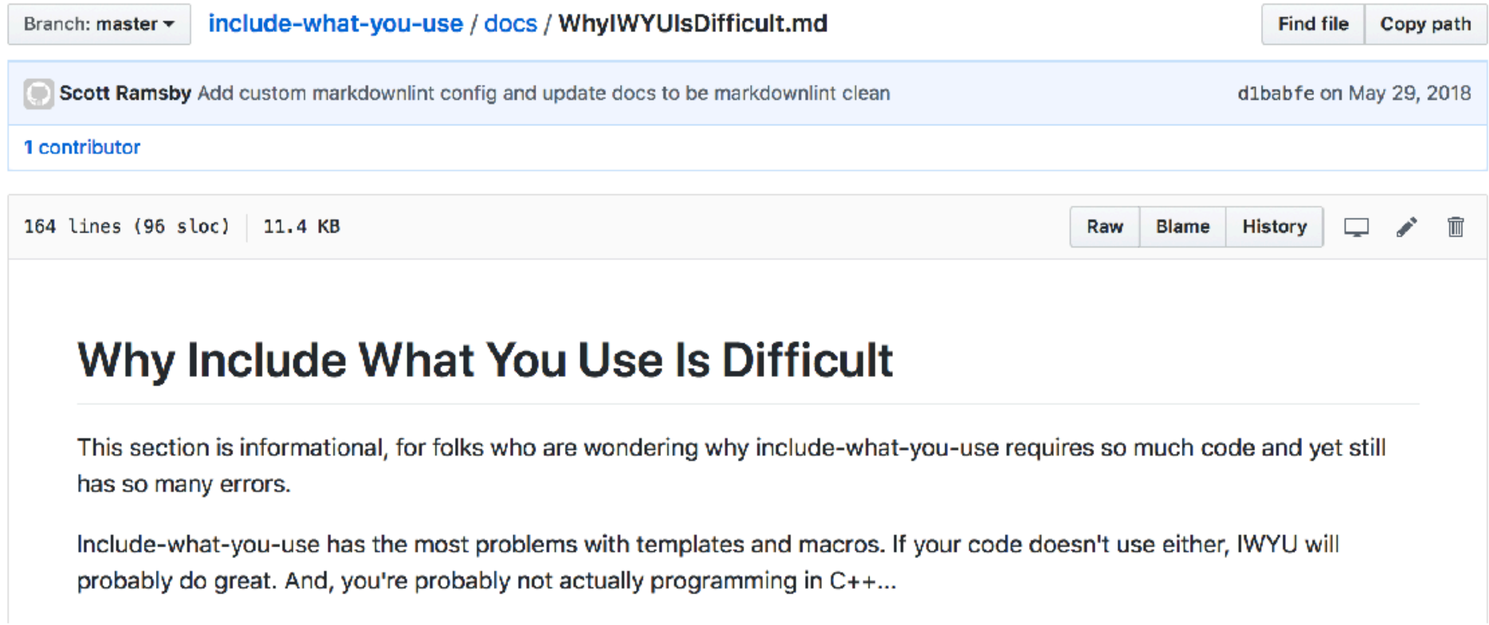
Includee file	Times included	Line contribution	Line contribution inclusive
debuggerext.cpp (debuggerext)	1	599	2675
▷ EventCallback.h (debuggerext)	3	294	1359
EventCallback.cpp (debuggerext)	1	279	1070
◢ DebugContext.h (debuggerext)	13	892	892
StackTrace.cpp (debuggerext)	1	223	223
debuggerext.cpp (debuggerext)	1	223	223
▷ OutputCallback.h (debuggerext)	2	223	223
▷ EventCallback.h (debuggerext)	2	223	223
▷ StackTrace.h (debuggerext)	2	0	0

The power of tools: Includes
Proactive optimization

—

Proactive optimization

- Precompiled headers
- Optimizers
  - "*Unused includes*" check
  - include-what-you-use (and don't include what you don't use)
  - Includator



References

- Eric Niebler, Standard Ranges
  - [Dec, 2018] http://ericniebler.com/2018/12/05/standard-ranges/
- Aras Pranckevičius, "Modern" C++ Lamentations
  - [Dec 2018] http://aras-p.info/blog/2018/12/28/Modern-C-Lamentations/
- Nicolai Josuttis, "The Nightmare of Initialization in C++
  - [CppCon 2018] https://www.youtube.com/watch?v=7DTlWPgX6zs
- Timur Doumler, “Initialization in modern C++”
  - [Meeting C++ 2018] https://www.youtube.com/watch?v=ZfP4VAK21zc
- Herb Sutter, Meta - Thoughts on Generative C++
  - [CppCon 2017] https://www.youtube.com/watch?v=4AfRAVcThyA
- Niklas, bitsquid blog, Caring by Sharing: Header Hero
  - [2011] http://bitsquid.blogspot.co.uk/2011/10/caring-by-sharing-header-hero.html
- C++ Foundation Developer Survey
  - [2018-2] https://isocpp.org/files/papers/CppDevSurvey-2018-02-summary.pdf
- The State of Developer Ecosystem Survey
  - [2017] https://www.jetbrains.com/research/devecosystem-2017/cpp/
  - [2018] https://www.jetbrains.com/research/devecosystem-2018/cpp/
  - [2019] https://www.jetbrains.com/research/devecosystem-2019 – up and running!

Thank you
for your attention
—

Use tools!

Questions?