

Debug C++

Without Running

(Thoughtful Reading of C++ code)

Anastasia Kazakova

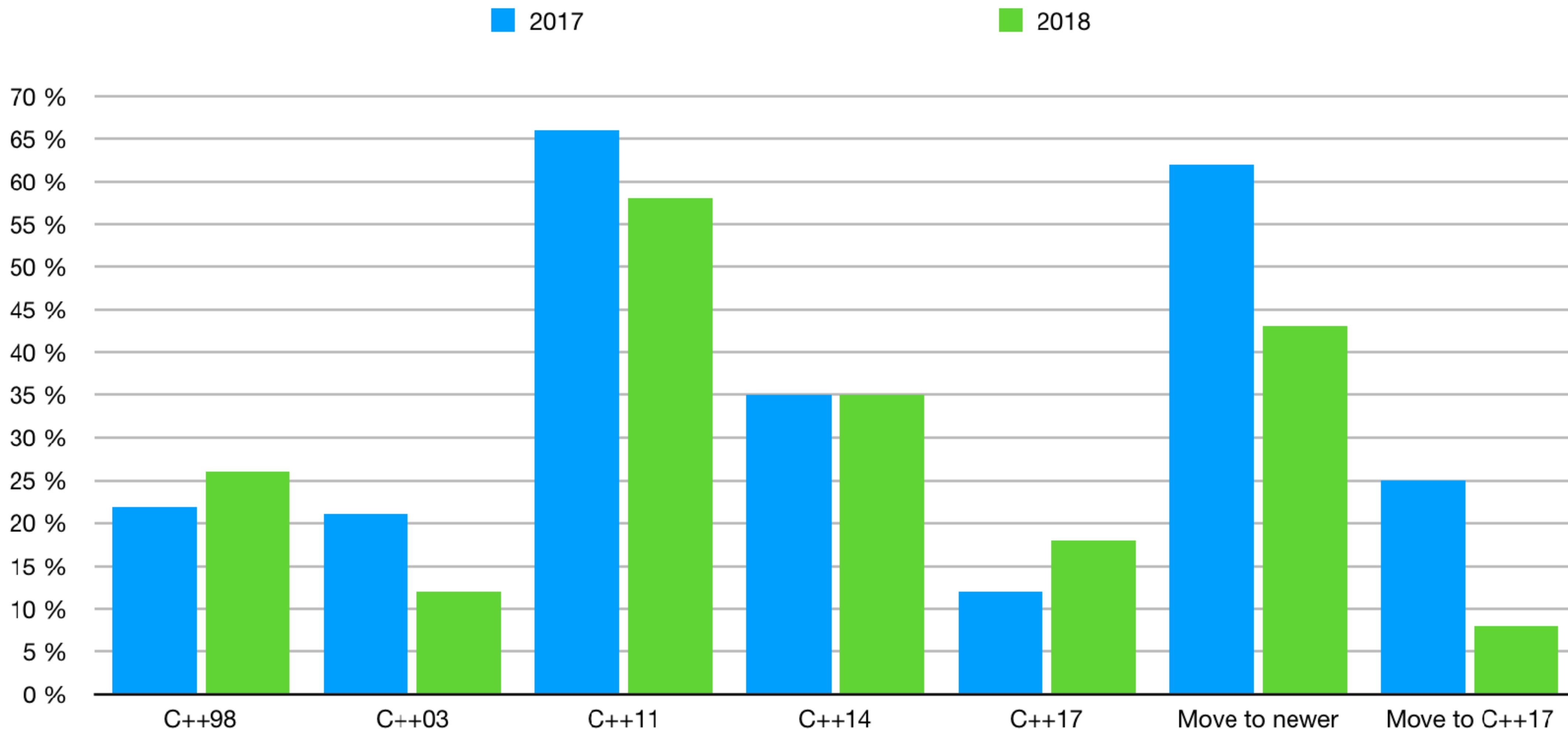
JetBrains

@anastasiak2512

C++ on Sea 2019

JetBrains Dev Ecosystem survey 2017/2018

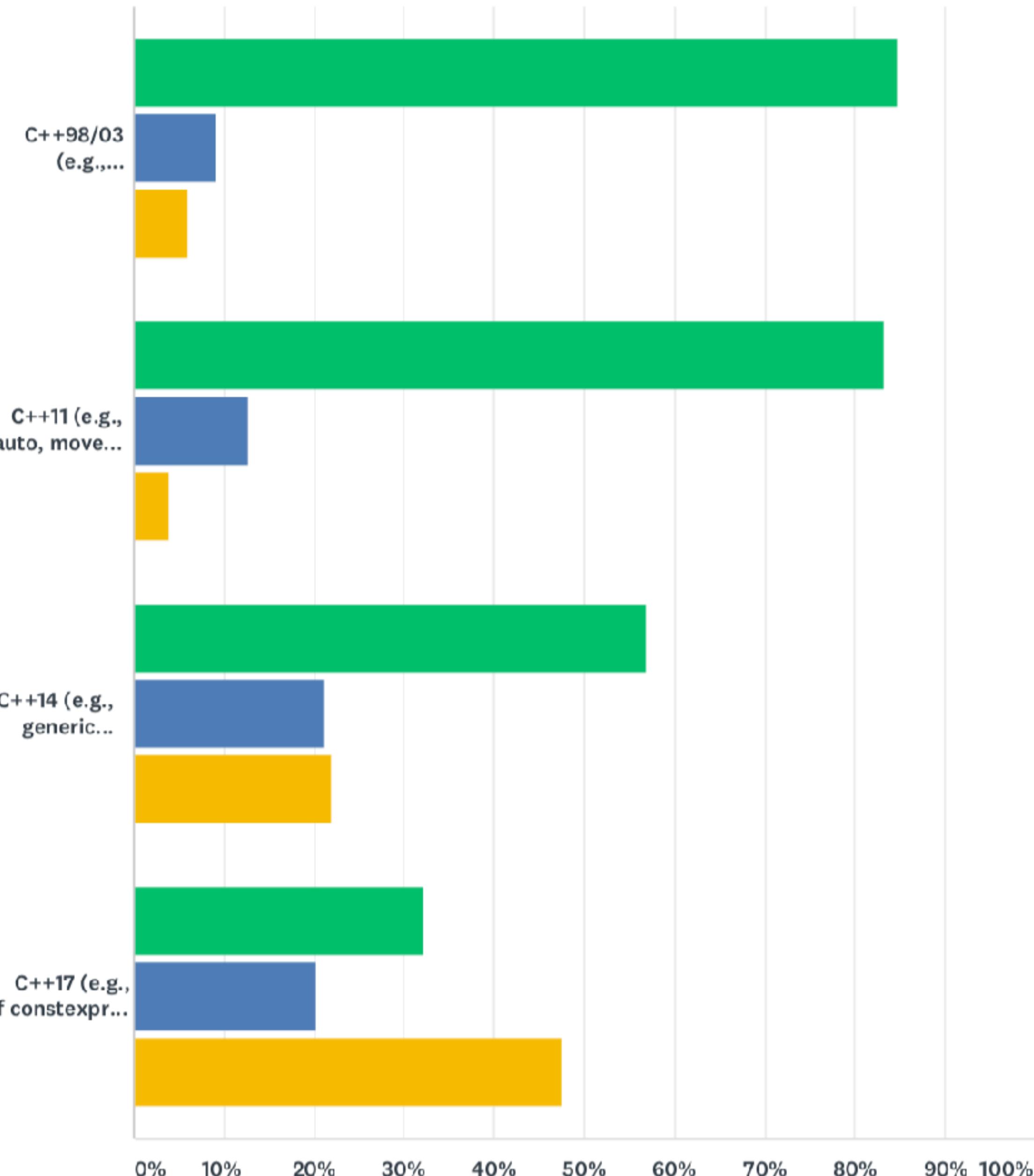
<https://www.jetbrains.com/research/devcosystem-2019> – Up & Running!



C++ Foundation Developer Survey 2018

Constexpr Edge References Resources Toolchain Impossible
Practices MSVC Learn Nope Code Modules Compiler
Hard to Understand New Features Colleagues
Standard Amount Language Tools Difficulty Evolves
New Stuff Dependencies Older Past Books Difficult to Understand

■ Yes: Pretty much all features ■ Partial: Just a few selected features
■ No: Not allowed



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



Eric Niebler
@ericniebler

Follow

New blog post! Ranges are coming in C++20. Read all about it.

ericniebler.com/2018/12/05/sta...

#cpp

Standard Ranges

As you may have heard by now, Ranges got merged and will be part of C++20. This is huge news and represents probably the biggest shift the Standard Library has seen since it was first st...
ericniebler.com

4:59 AM - 6 Dec 2018

82 Retweets 196 Likes



7

82

196



Aras Pranckevičius
@aras_p

Follow

That example for Pythagorean Triples using C++20 ranges and other features sounds terrible to me.

ericniebler.com/2018/12/05/sta...

And yes I get that ranges can be useful, projections can be useful etc. Still, a terrible example! Why would anyone want to code like that?!

Standard Ranges

As you may have heard by now, Ranges got merged and will be part of C++20. This is huge news and represents probably the biggest shift the Standard Library has seen since it was first st...
ericniebler.com

12:07 AM - 24 Dec 2018

53 Retweets 257 Likes



40

53

257



Raging at ranges



Eric Smolikowski @esmolikowski · 24 Dec 2018

Replies 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

1

9

✉

[1 more reply](#)



Mike Nicolella @MikeNicolella · 24 Dec 2018

Replies to [@aras_p](#)

And worse, who would want to debug that?

1

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

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.

1

1

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

```



NICOLAI JOSUTTIS

The Nightmare of Initialization in C++

	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



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>;
}
```

x86-64 gcc 7.3 (Editor #1, Compiler #1) C++ x

x86-64 clang 6.0.0 (Editor #1, Compiler #1) C++ x

x86-64 clang 6.0.0 -std=c++14

A 11010 .LX0: .text // \s+ Intel Demangle Libraries Add new...

```
1 main:                                # @main
2     push    rbp
3     mov     rbp, rsp
4     mov     dword ptr [rbp - 4], 0
5     mov     eax, dword ptr [pi<int, 42>]
6     pop     rbp
7     ret
8 pi<int, 42>:
9     .long   42                            # 0x2a
```

x86-64 gcc 7.3 (Editor #1, Compiler #1) C++ x

x86-64 gcc 7.3 -std=c++14

A 11010 .LX0: .text // \s+ Intel Demangle Libraries Add new...

```
1 main:
2     push    rbp
3     mov     rbp, rsp
4     mov     eax, DWORD PTR pi<int, 42>[rip]
5     pop    rbp
6     ret
7 pi<int, 42>:
8     .long   42
```

42 sample

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

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

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

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

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

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

Reading macro sample

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

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
#ifndef 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){...}

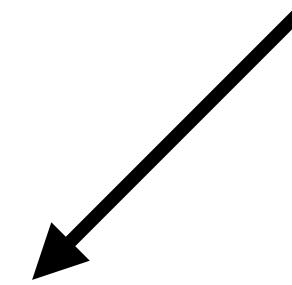
Fraction operator*(Fraction lhs, const Fraction& rhs){...}

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

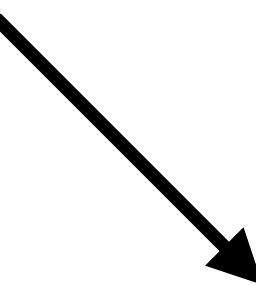
    std::cout << f1 << " * " << f2 << " = " << f1 * f2 << '\n';
}
```

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 hidars
- Abstractions need tool support
- Good abstractions do need to be toolable

Herb's list for toolable and debuggable C++

⇒ Abstractions need **tool support**.

C

C++98

C++17

proposed

Variables: hide values ⇒ need watch windows (debug)

Functions: hide code ⇒ need Go To Definition (IDE) / Step Into (debug)

Pointers: hide indirection ⇒ need visualizers (debug)

#includes: hide dependencies ⇒ need file “touch”-aware build (build)

Classes: hide code/data, encapsulate behavior ⇒ need most of the above

Overloads: hide static polymorphism ⇒ need better warning/error msgs

Virtuals: hide dynamic polymorphism ⇒ need dynamic debug support

constexpr functions: hide computations ⇒ need compile-time debug

if constexpr: hide whether code even has to compile ⇒ need colorizers

Modules: hide dependencies ⇒ need module “touch”-aware build (build)

Compile-time variables: hide values ⇒ need compile-time watch

Compile-time code/functions: hide computation ⇒ need compile-time debug

Injection, metaclasses: generate entities ⇒ need to visualize them

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)
```



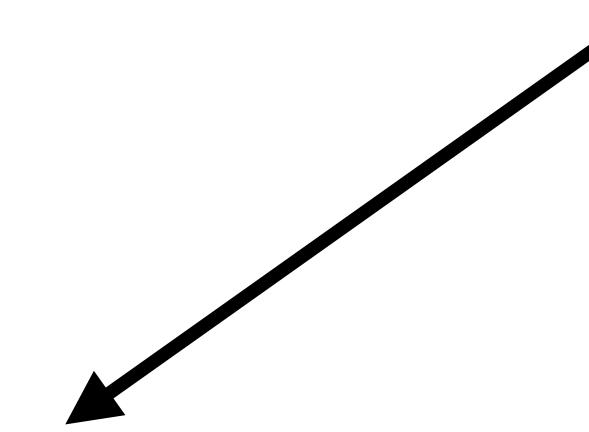
The power of tools: Macro debug



Substitute macro – practical sample

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

```
BOOST_PP_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!

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

```
#define FUNCM func
```

```
- void test()  
{
```

```
    FUNCM( i: 0 );  
    FUNCM( i: 1.2 );
```

```
    int func;  
    FUNCM;
```

```
}
```

Declaration of identifier 'func' 

func(int i) -> void RsCppDemo.cpp

func(double i) -> void RsCppDemo.cpp

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

```
14 template<typename T, typename U>
15 auto doOperation(T t, U u) -> decltype(t + u) {
16     return t + u;
17 }
18
19 void fun_type() {
20     auto op = doOperation(3.0, 0);
21     //... double op
22 }
23
24
```



```
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);
    //...
}
```

double op = doOperation(3.0, 0)



```
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);
    //... <anonymous>::op
}

(local variable) double op
```



go to

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:

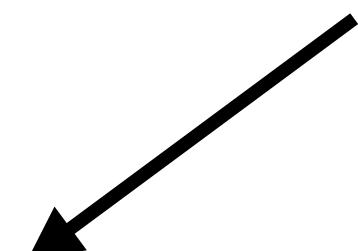
- Show inferred type
- Substitute typedef (one step)



```
#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;
```

A diagram illustrating the flow of type inference. Two arrows point from the code 'boost::mpl::vector5<A, B, C, D, E>::item3 hi;' to the 'D' in 'D hi;'. One arrow points from the 'item3' part to the 'D', and another arrow points directly from the 'hi;' part to the 'D'.



The power of tools: Meta info debug

Debug the abstractions

- Instantiating templates

A screenshot of a C++ code editor interface. At the top, there's a toolbar with icons for file operations. Below it, a status bar shows the word "handle". The main area displays the following code:

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

The code editor highlights the template parameters `T1 = int` and `Types = float` in blue, indicating they are being analyzed or resolved.

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

A tooltip window is open over the third overload of `handle`, showing the signature `(function) void handle<T1, Types...>(Tuple<T1, Types&...>)` and the label "Third overload". A cursor arrow points to the third overload line in the code.

A screenshot of a debugger interface showing function details. The current focus is on the `check()` function, which contains a call to `handle`. The tooltip window from the previous slide is still visible, providing detailed information about the `handle` function's signature and overloads.

```
void check() {  
    handle(Tuple<>()); // -> 1  
    handle(Tuple<int, float>()); // -> 2  
    handle(Tuple<int, float&>()); // -> 3  
    handle(Tuple<T1, Types...>()); // -> 3  
    //Third overload  
    (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

template <typename T = int, typename C = std::vector<T>>

Instantiation Arguments ×

Enter a sample argument for each parameter

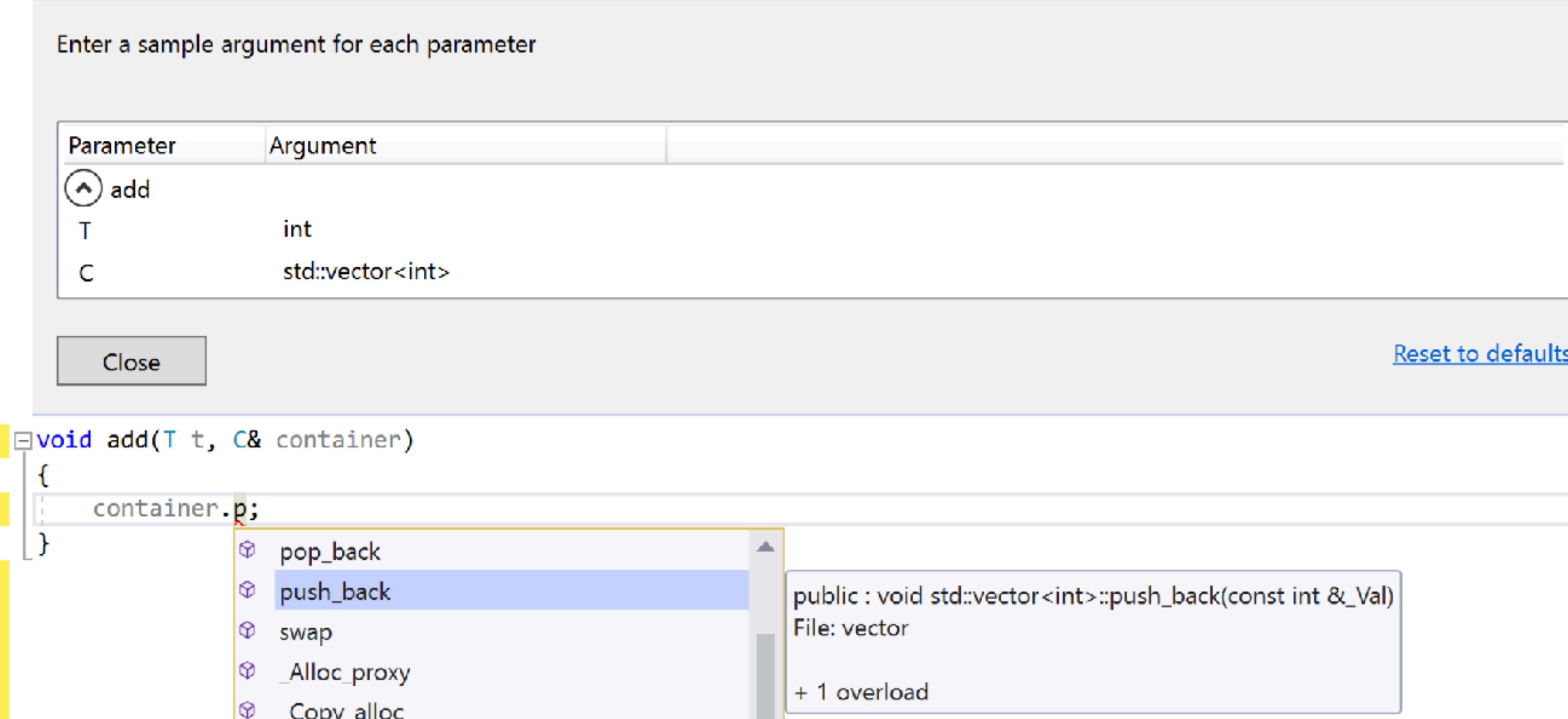
Parameter	Argument
add	
T	int
C	std::vector<int>

[Close](#) [Reset to defaults](#)

```
void add(T t, C& container)
{
    container.p;
}
```

pop_back
push_back
swap
_Alloc_proxy
_Copy_alloc

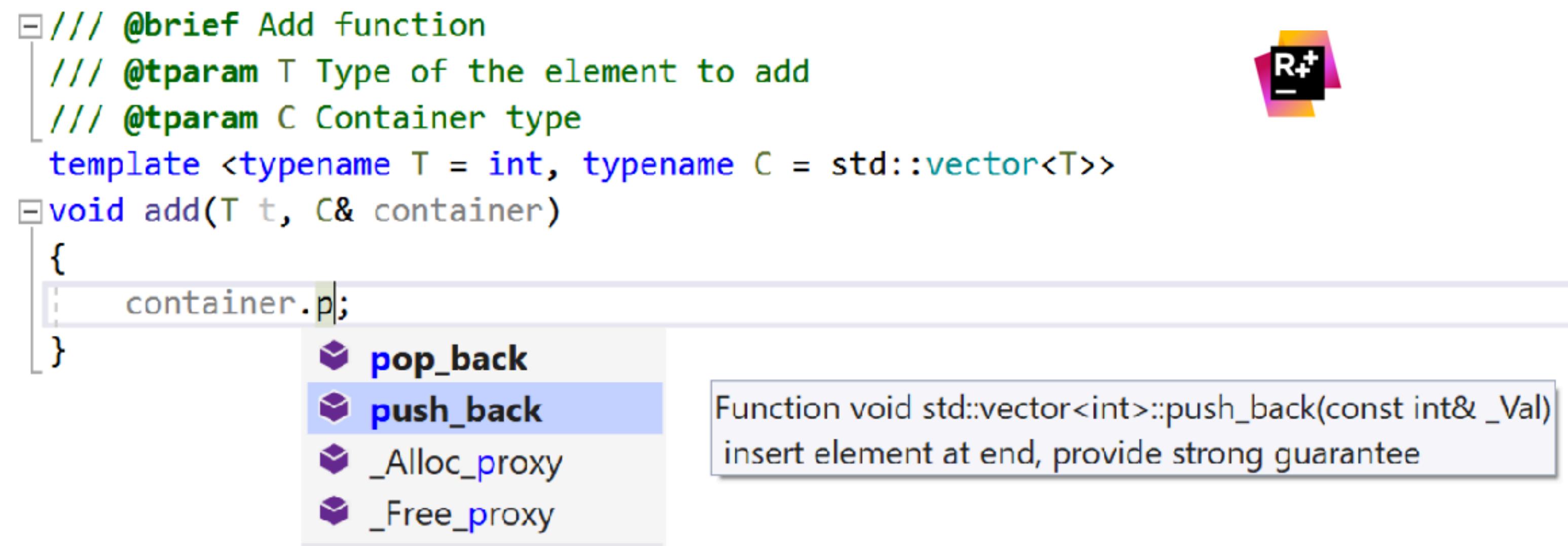
public : void std::vector<int>::push_back(const int &_Val)
File: vector
+ 1 overload



The power of tools: Meta info debug

Debug the abstractions

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



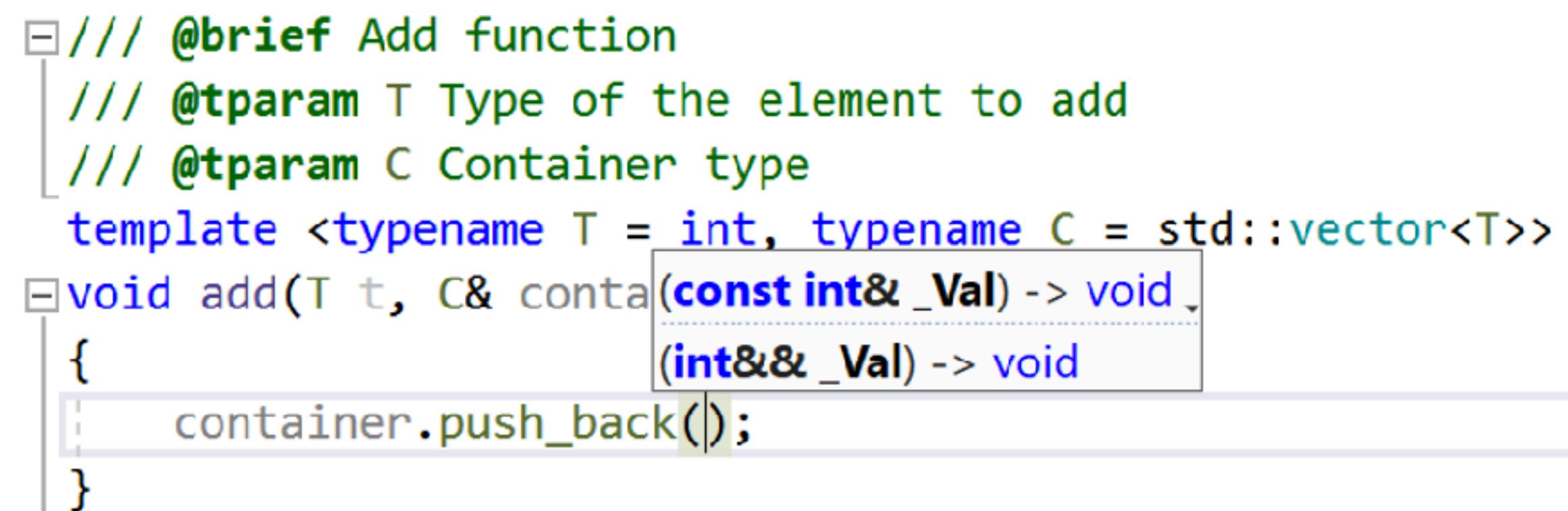
A screenshot of the Resharper (R#) IDE interface. On the right, there's a floating window showing intellisense options for the code. The code snippet is:

```
/// @brief Add function
/// @tparam T Type of the element to add
/// @tparam C Container type
template <typename T = int, typename C = std::vector<T>>
void add(T t, C& container)
{
    container.p;
}
```

The intellisense dropdown shows five options: `pop_back`, `push_back` (which is highlighted in blue), `_Alloc_proxy`, and `_Free_proxy`. To the right of the dropdown, a tooltip provides the documentation for `push_back`:

Function void std::vector<int>::push_back(const int& _Val)
insert element at end, provide strong guarantee

The R# logo is visible in the top right corner.



A screenshot of the Resharper (R#) IDE interface. The intellisense dropdown is shown over the parameter `const int& _Val` in the `add` function signature. The code snippet is identical to the one above:

```
/// @brief Add function
/// @tparam T Type of the element to add
/// @tparam C Container type
template <typename T = int, typename C = std::vector<T>>
void add(T t, C& container(const int& _Val) -> void)
```

The intellisense dropdown shows two options: `(const int& _Val) -> void` (highlighted in blue) and `(int&& _Val) -> void`.

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



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

▲ 6 of 8 ▼ void foo<int>(int)



```
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

```
int main() {  
    foo(1);  
}  
■ foo(void) : void  
■ foo(int) : void  
■ foo(T) : void  
■ foo(S) : void  
■ foo(ConvertibleToInt) : int
```



```
void f(<no parameters>): void  
void f(int): void  
void f  
    foo function  
    (S): void  
    (ConvertibleToInt): int  
int ma  
    (int a, int b): void  
        foo(1);  
    }
```

The power of tools: Overloads debug



Show candidates set via
parameter info

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

```
void foo() { std::cout << "1\n"; }
///foo function
void foo(int id) { std::cout << "2\n"; }
template<typename T> void foo(T t) { std::cout << "3\n"; }
template<> void foo(int ti) { std::cout << "4\n"; }
struct S {};
void foo(S 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"; } } namespace M
    void foo(double) { std::cout << "8\n"; }
} namespace N
```

A screenshot of an IDE interface showing code completion for the function 'foo'. A tooltip box displays five overload signatures:

- void f(int id) -> void
- void f(S s) -> void
- void f(ConvertibleToInt) -> int
- void t(int a, int b) -> void
- void t(int a, double d) -> void = delete

The last signature, 'void t(int a, double d) -> void = delete', is highlighted with a red underline. Below the tooltip, a line of code is shown: 'foo(a: 5, d: 1.7)'. The variable 'a' has a red wavy underline, while 'd' has a red arrow pointing to it.

The power of tools: Overloads debug

- Show candidates set
- Show explanations
 - When overload fails



```
3     template<typename T, typename = decltype(T().method())>
4     void bar(T);
5
6     struct X {
7         void method() {}
8     };
9
10    struct Y {};
11
12    void overload_check_3(X x, Y y) {
13        bar(x);
14        bar(y);
15    }
16
17    No matching function for call to 'bar'
18    candidate template ignored: substitution failure [with T = Y]: no member named 'method' in 'Y'
19
20 }
```



```
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



A screenshot of an IDE interface showing the file `Measurement.h`. The code defines a class `Measurement` with a constructor and two methods: `measureOnce` and `measureSet`. The code is annotated with several green vertical bars and arrows pointing to specific lines, likely indicating points of interest or errors in the code. The IDE has a dark theme with tabs for `Measurement.h` and `Measurement.cpp`, and a status bar showing the project name `RSCPP_2018_3_demo` and the scope `(Global Scope)`.

```
1 #pragma once
2 #include <vector>
3
4 class Measurement
5 {
6 public:
7     Measurement() = default;
8     ~Measurement() = default;
9
10    void measureOnce(int val);
11    void measureSet(std::vector<int> valVec);
12 }
13
14 
```

I

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 screenshot shows a Windows application window titled "Report" with a blue header bar. Below the header is a toolbar with four buttons: "Scan", "Report" (which is selected), "Includes", "Errors", and "Missing Files". The main area is titled "Report" and contains the following text:

Files:	923
Total Lines:	171 117
Total Parsed:	6 046 220
Blowup Factor:	35,33

Below this, the title "Biggest Contributors" is displayed, followed by a list of files and their counts:

263 840	map.inl
199 626	set.inl
138 093	string.h
131 670	vector.inl
126 315	hash_map.inl
123 519	shader.h
118 872	sort_map.inl
117 384	deque.inl
107 672	vector.h
106 720	array.inl
102 365	matrix4x4.inl
100 360	critical_section.h
94 180	file_system.h
...	...

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

Branch: master [include-what-you-use / docs / WhyIWYUIsDifficult.md](#) [Find file](#) [Copy path](#)

 Scott Ramsby Add custom markdownlint config and update docs to be markdownlint clean d1babfe on May 29, 2018

1 contributor

164 lines (96 sloc) | 11.4 KB [Raw](#) [Blame](#) [History](#)   

Why Include What You Use Is Difficult

This section is informational, for folks who are wondering why include-what-you-use requires so much code and yet still has so many errors.

Include-what-you-use has the most problems with templates and macros. If your code doesn't use either, IWYU will probably do great. And, you're probably not actually programming in C++...

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=7DTIWPGX6zs>
- 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?