Debug C++ Without Running

Anastasia Kazakova
JetBrains
@anastasiak2512

Agenda

1. Tricky C++ language. Show samples!

2. Seems to help but it doesn’t. Why?
   - Running / Debugging
   - Static / dynamic code analysis

3. Should help – IDEs! How?

Agenda

1. Tricky C++ language. Show samples!

2. Seems to help but it doesn’t. Why?
   - Running / Debugging
   - Static / dynamic code analysis

3. Should help – IDEs! How?

Time for a quote

“C makes it easy to shoot yourself in the foot;
C++ makes it harder, but when you do it blows your whole leg off”

- *Bjarne Stroustrup*

http://www.stroustrup.com/bs_faq.html#really-say-that

C++ difficulties: math

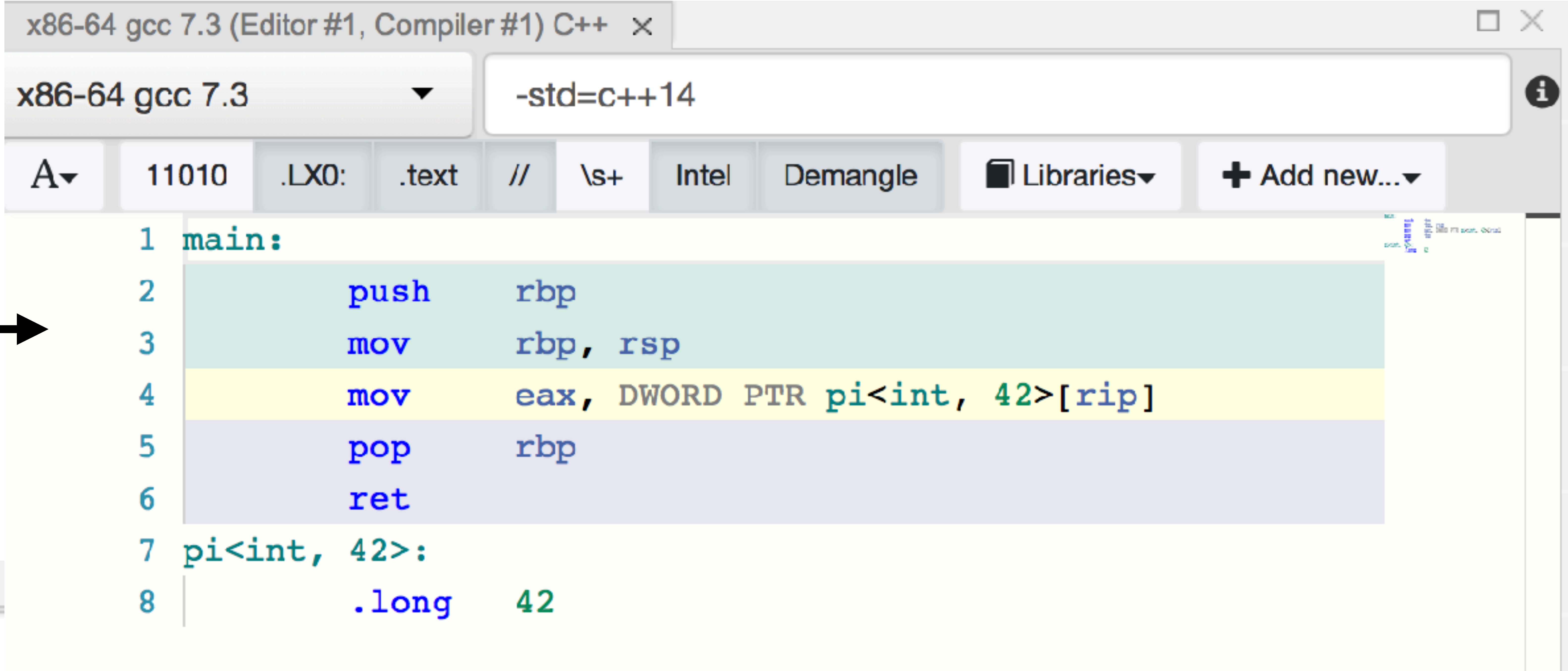
```
static_assert(-1 > 1u);
```

C++ difficulties: 42

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

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

C++ difficulties: 42

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

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

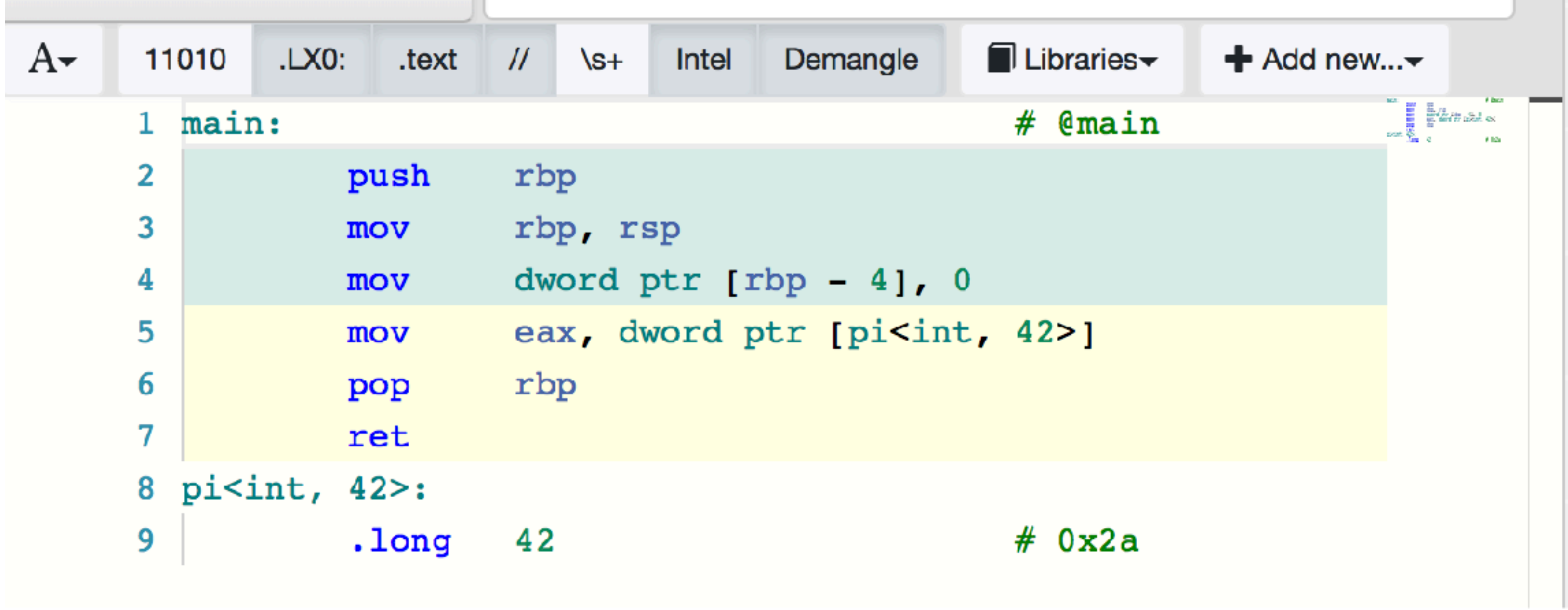


x86-64 gcc 7.3 (Editor #1, Compiler #1) C++ ×

x86-64 gcc 7.3

-std=c++14

```asm
main:
        push    rbp
        mov     rbp, rsp
        mov     eax, DWORD PTR pi<int, 42>[rip]
        pop     rbp
        ret
pi<int, 42>:
        .long   42
```



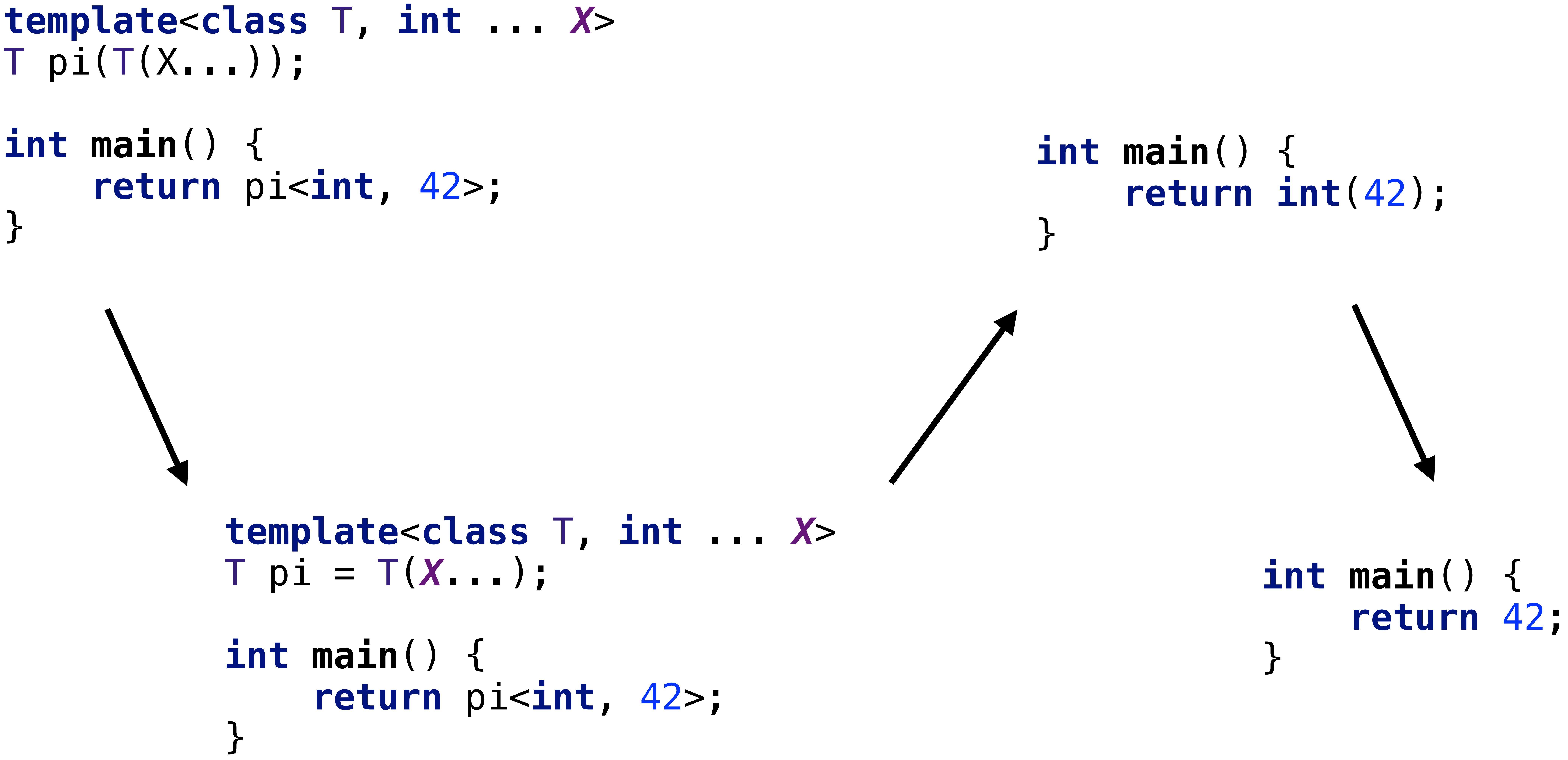
x86-64 clang 6.0.0 (Editor #1, Compiler #1) C++ ×

x86-64 clang 6.0.0

-std=c++14

```asm
main:                                   # @main
        push    rbp
        mov     rbp, rsp
        mov     dword ptr [rbp - 4], 0
        mov     eax, dword ptr [pi<int, 42>]
        pop     rbp
        ret
pi<int, 42>:
        .long   42                      # 0x2a
```

C++ difficulties: 42



```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;
}
```

C++ difficulties: macro

```
#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;
    }
}
```

C++ difficulties: macro

```
#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)
```

C++ difficulties: context

```
//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);
}
```

C++ difficulties: compile-time generation

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

```
$class interface {
    constexpr {
        compiler.require($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 { }
};
```

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

C++ difficulties: 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';
}
```

C++ difficulties: overloads

```cpp
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);
}
```

C++ difficulties: even more

- Constexpr
- Injected code
- …

Agenda

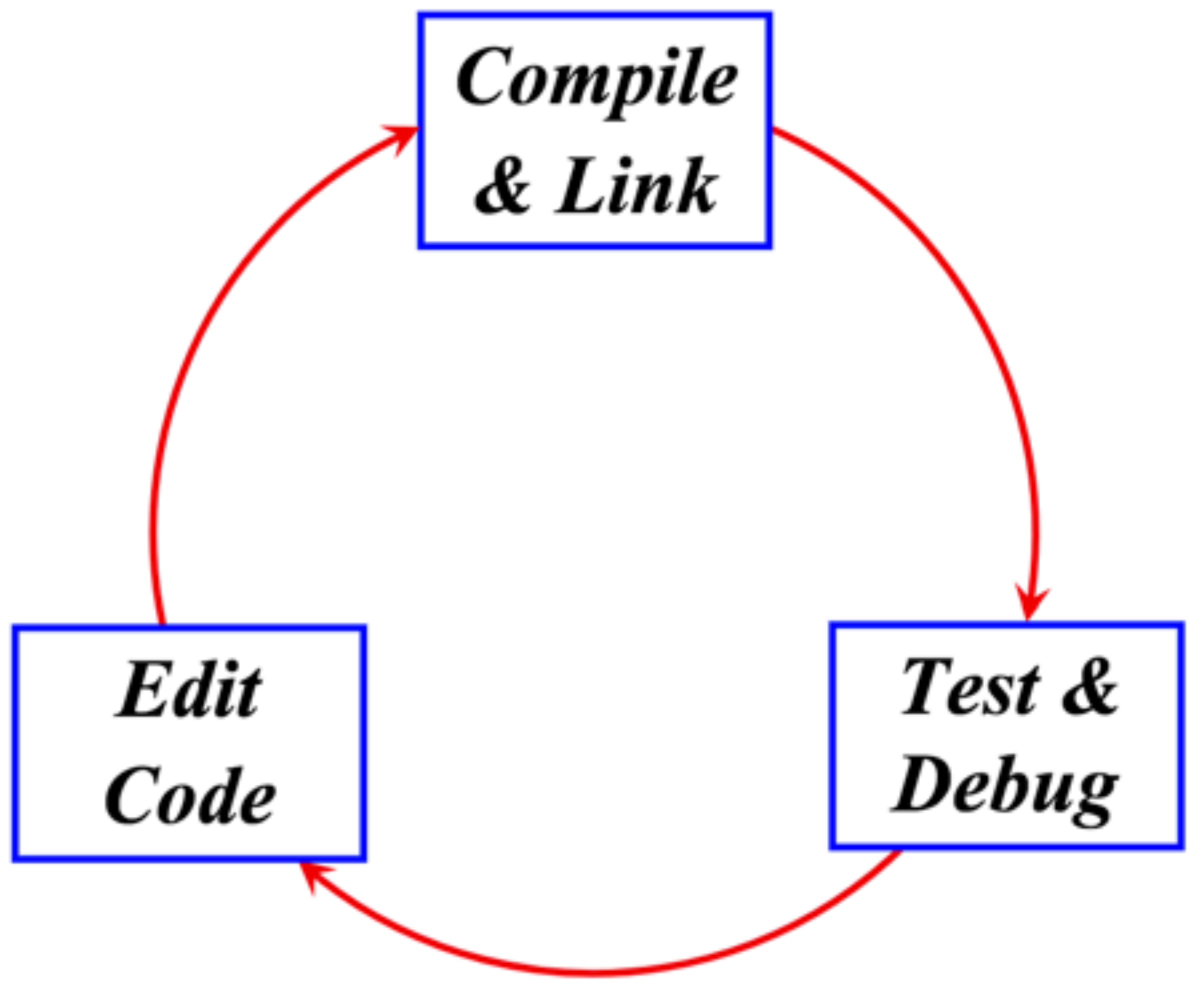
1. Tricky C++ language. Show samples!

2. Seems to help but it doesn’t. Why?
   - Run / Debug
   - Static / dynamic code analysis

3. Should help – IDEs! How?

Do these help?

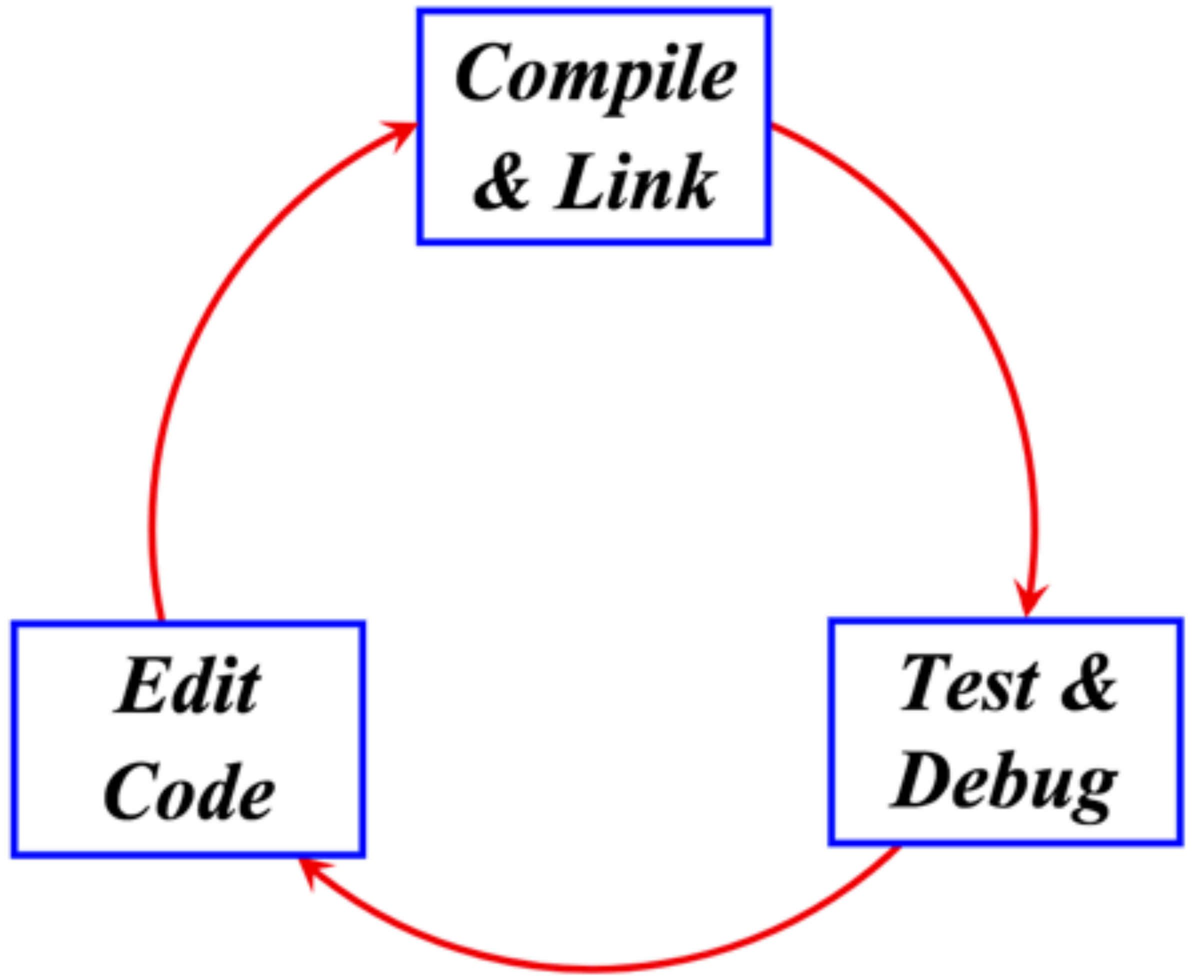
- Read-fix-run / read-fix-print-run and check results
- Debug
- Use static or dynamic code analysis



Do these help?

- Read-fix-run / read-fix-print-run and check results
- Debug
- Use static or dynamic code analysis

No!
(not always)



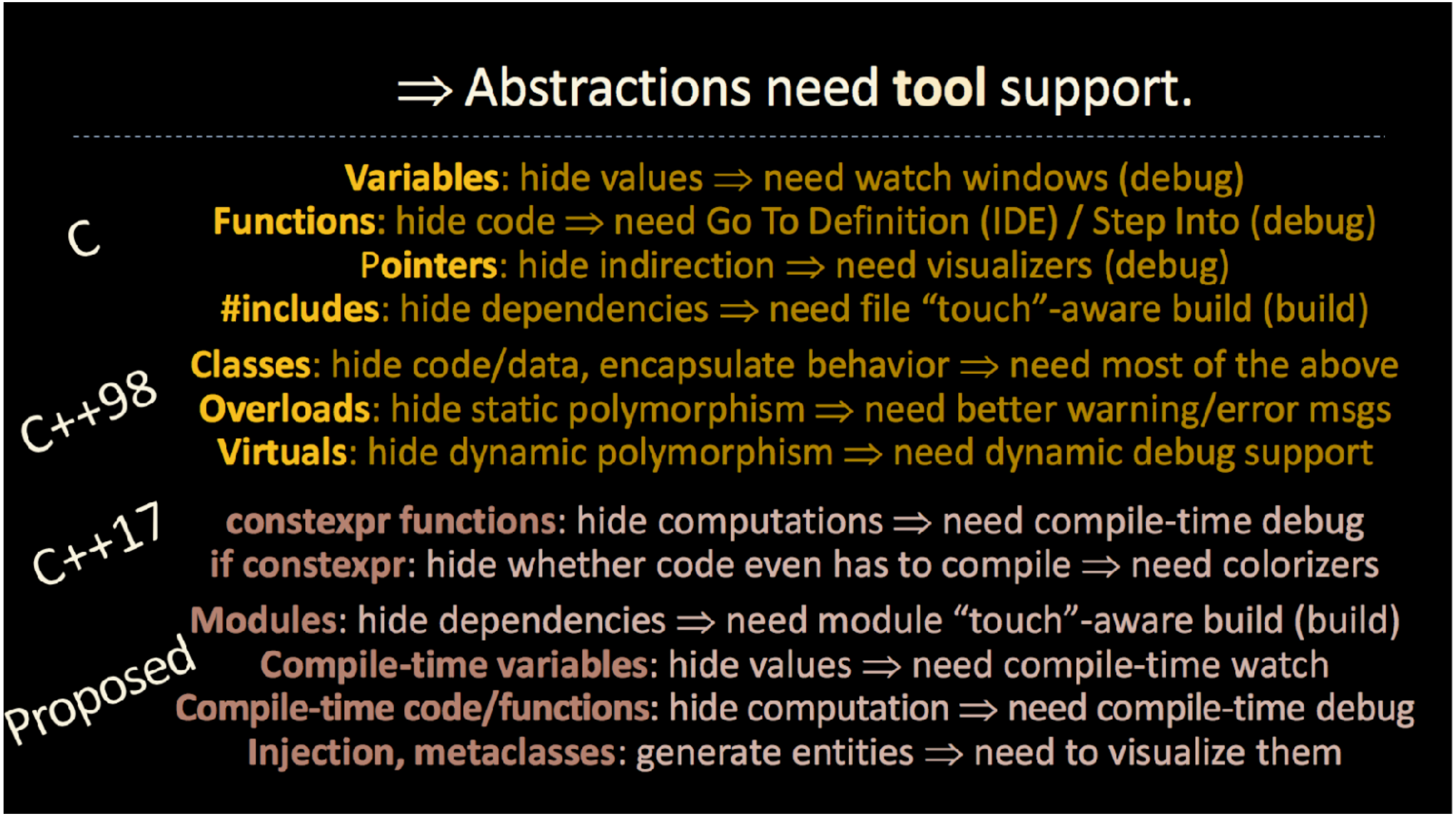
Herb Sutter’s keynotes
CppCon’17

—

Meta - Thoughts on Generative C++

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

Herb Sutter’s keynotes CppCon’17



⇒ Abstractions need **tool** support.

C

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)

C++98

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

C++17

constexpr functions: hide computations ⇒ need compile-time debug
if constexpr: hide whether code even has to compile ⇒ need colorizers

Proposed

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

Agenda

1. Tricky C++ language. Show samples!

2. Seems to help but it doesn’t. Why?
   - Running / Debugging
   - Static / dynamic code analysis

3. Should help – IDEs! How?

The power of tools:
Macro debug

—

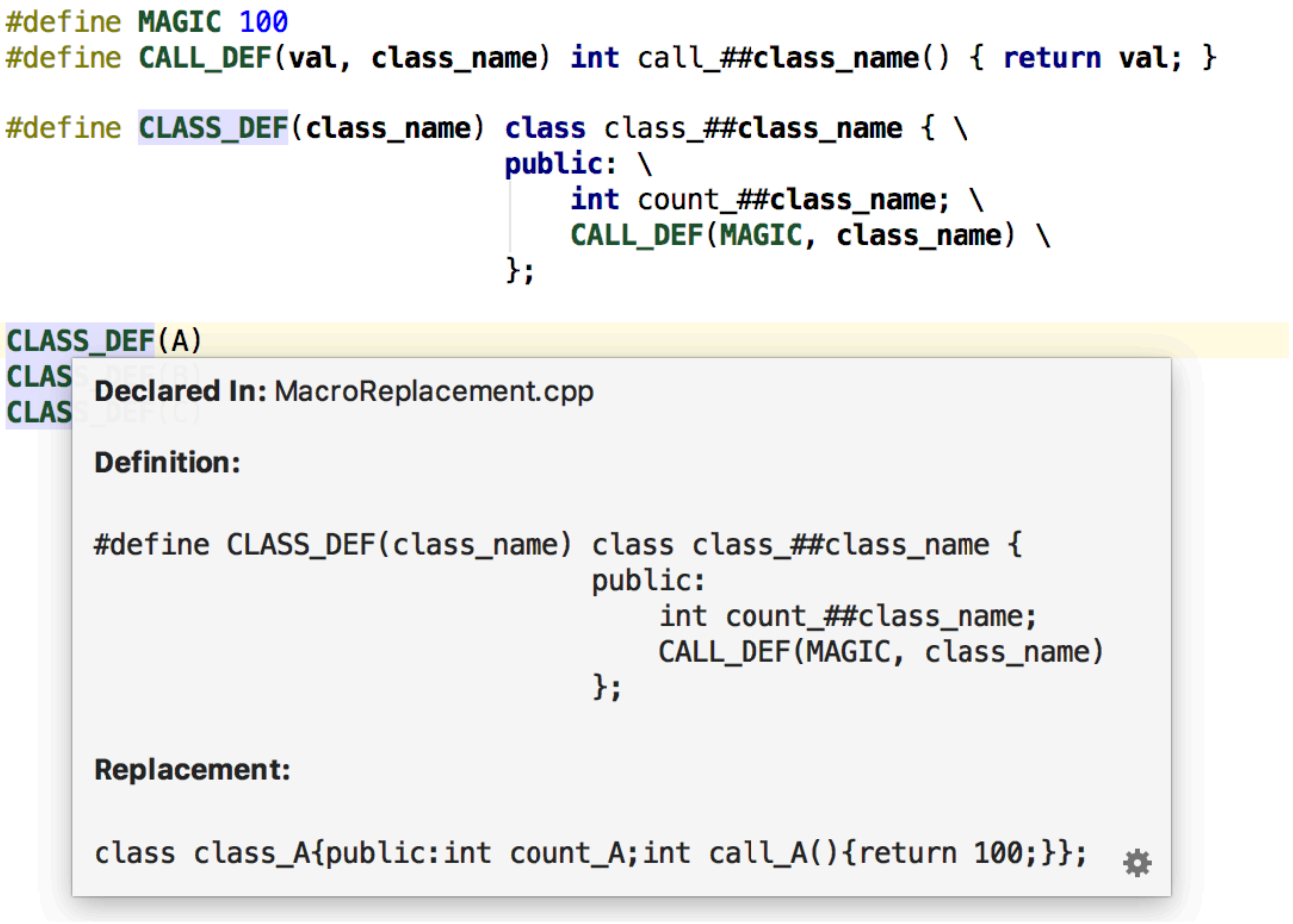
Goal – understand the substitution w/o running the preprocessor:

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

The power of tools: Macro debug

—

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:

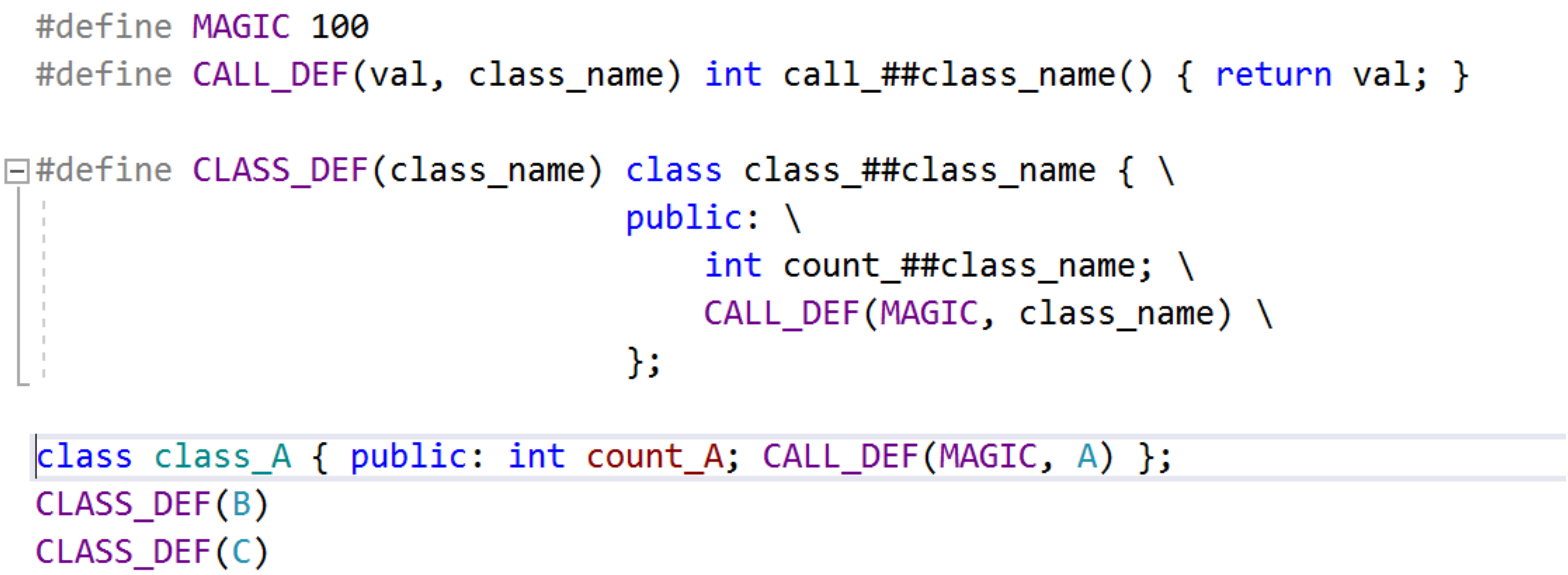
```
#define CLASS_DEF(class_name) class class_##class_name {
                              public:
                                  int count_##class_name;
                                  CALL_DEF(MAGIC, class_name)
                              };
```

Replacement:

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

The power of tools: Macro debug

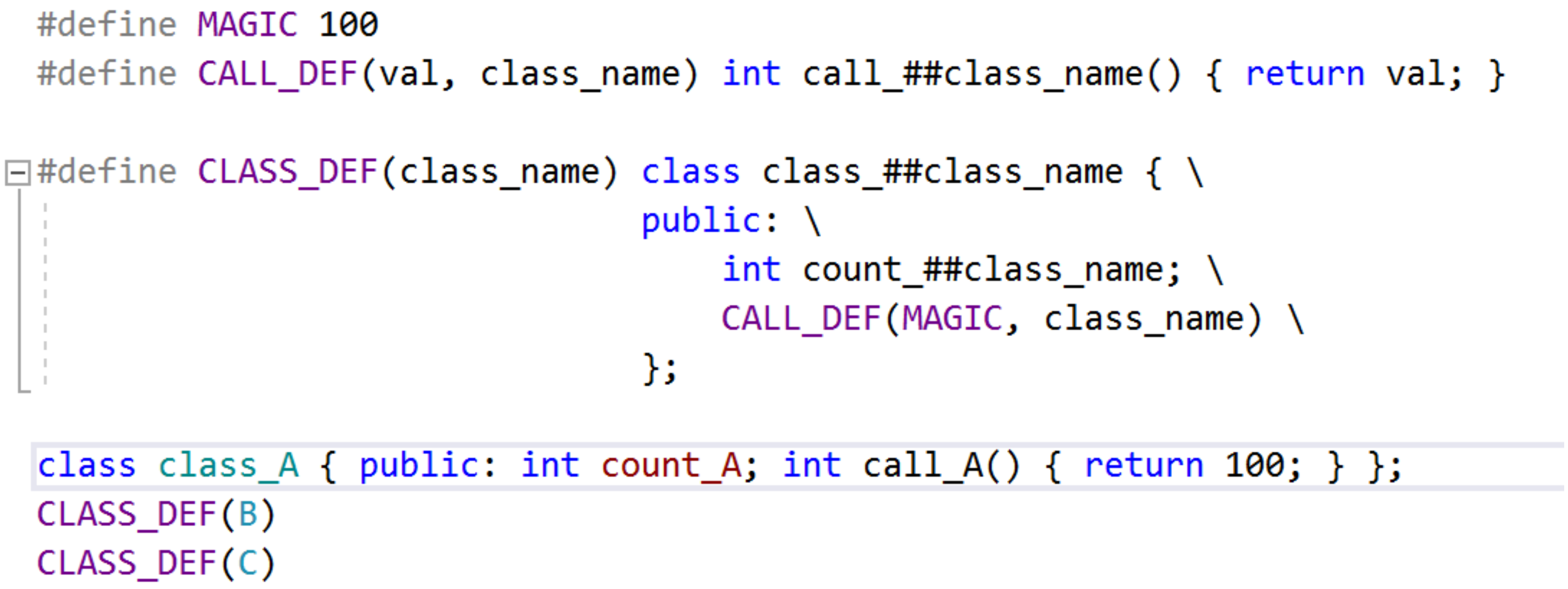
Substitute next step



```
#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

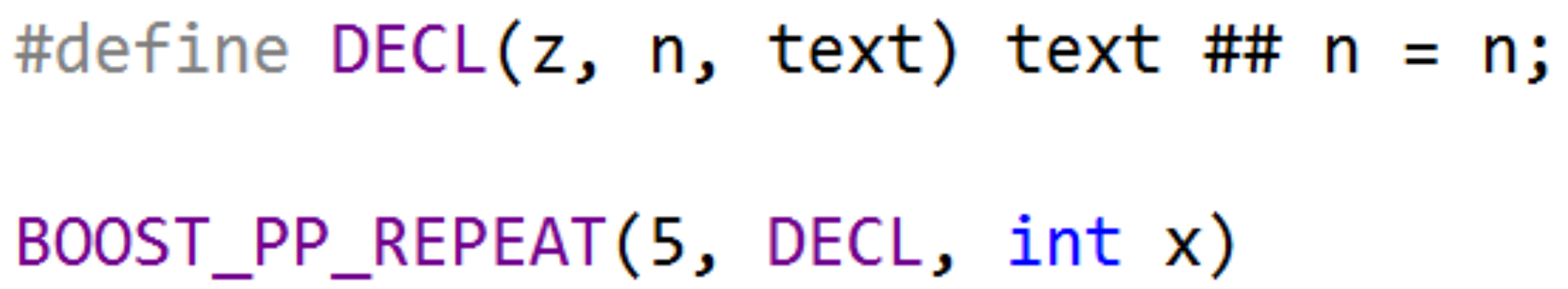
Substitute all steps



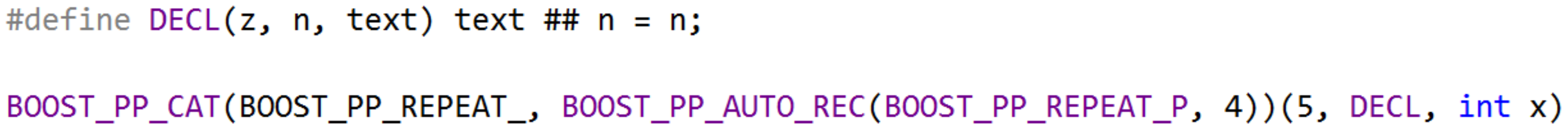
```
#define MAGIC 100
#define CALL_DEF(val, class_name) int call_##class_name() { return val; }

#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

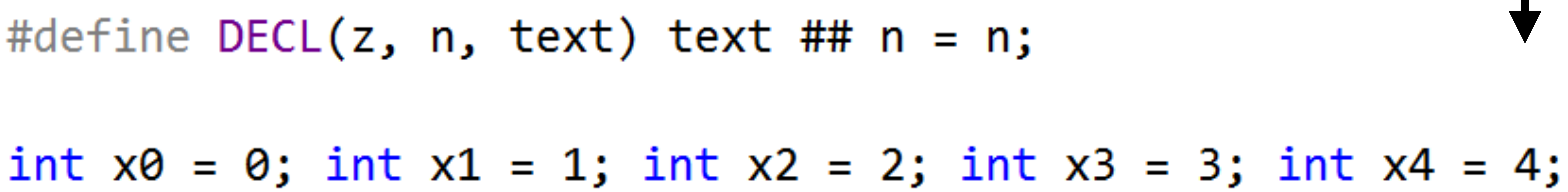
Substitute macro – practical sample



```
#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)
```



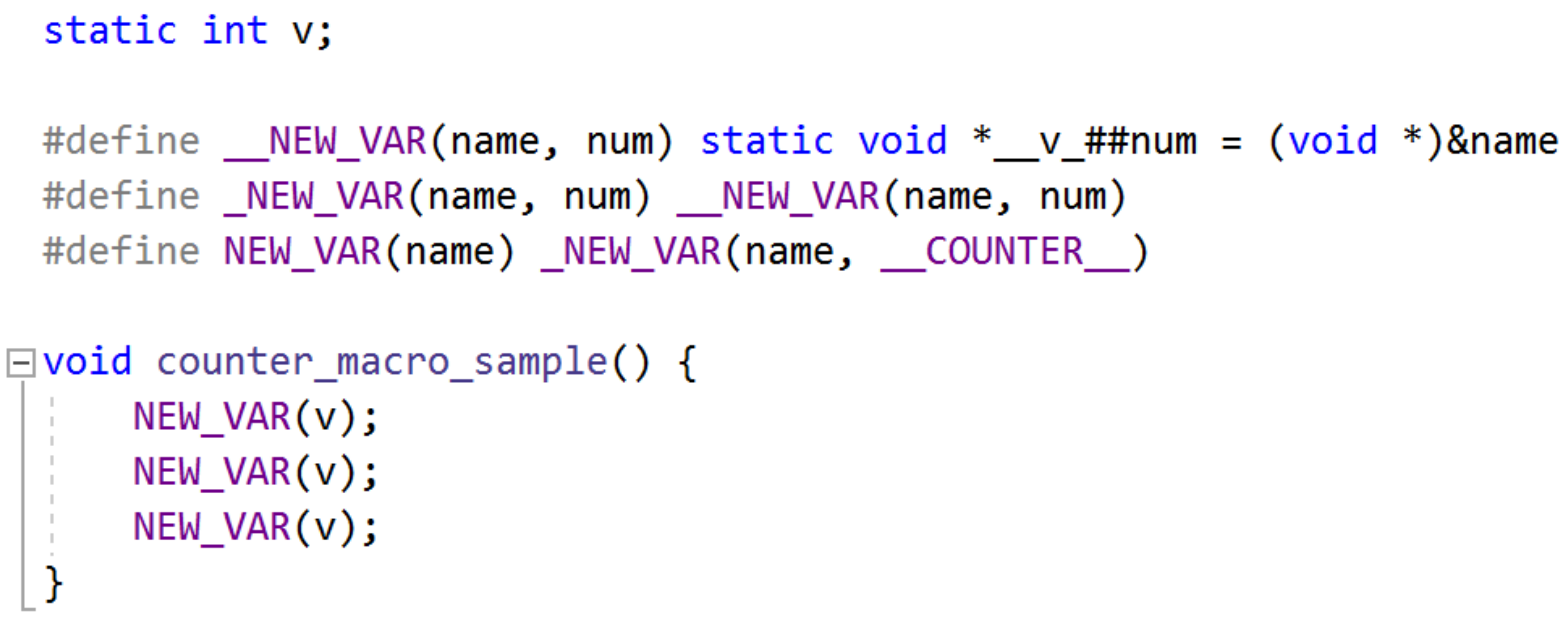
```
#define DECL(z, n, text) text ## n = n;

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!

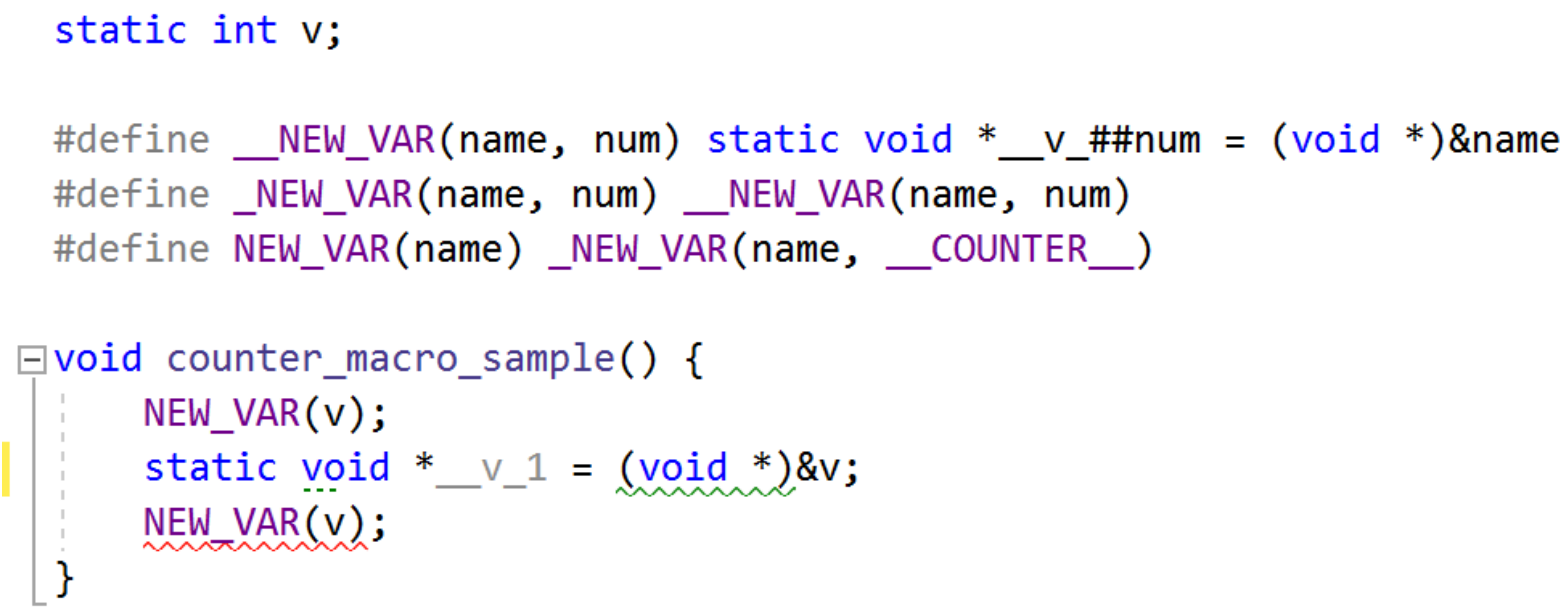


```
static int v;

#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!

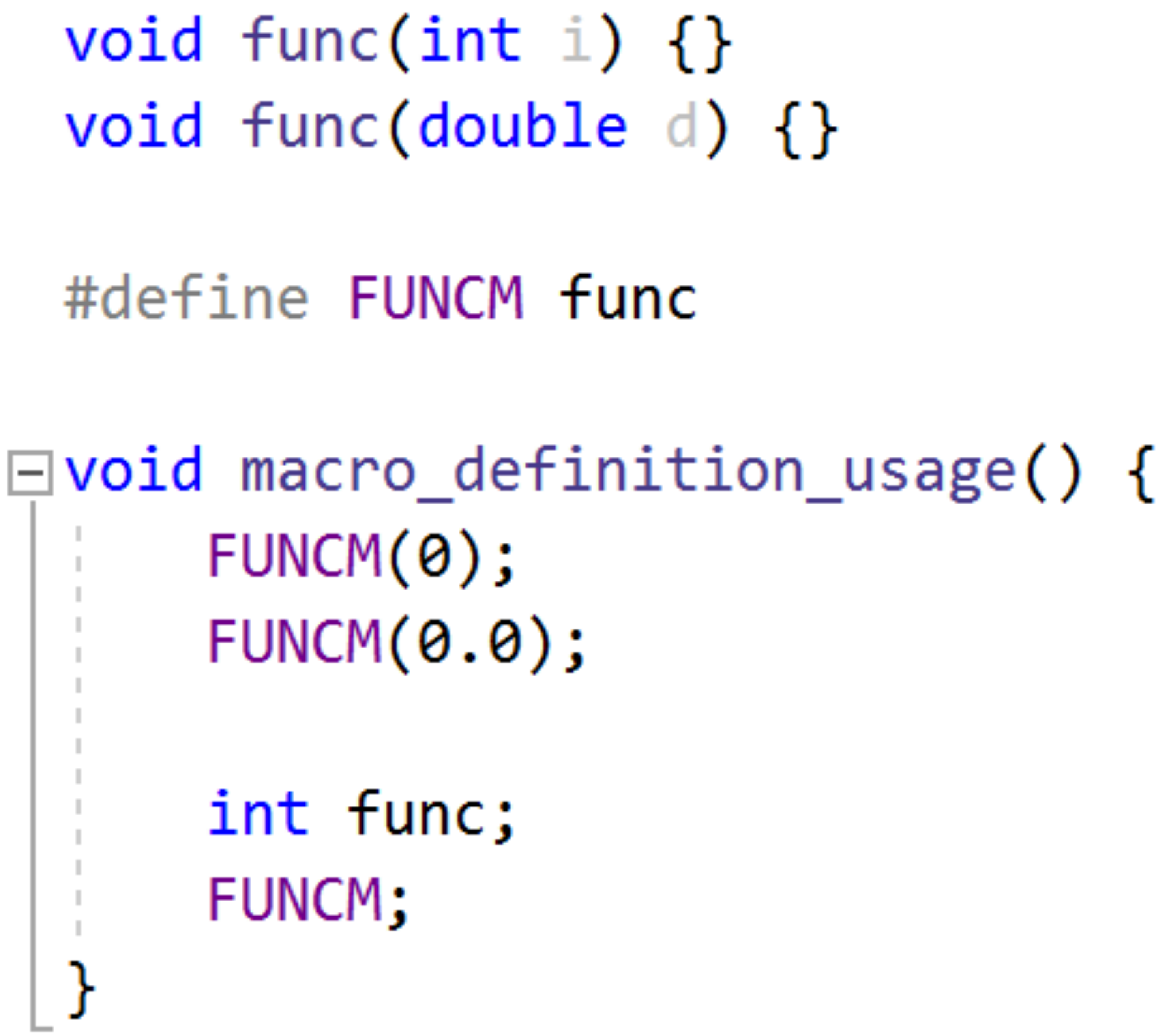
Code might be affected!



```
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!

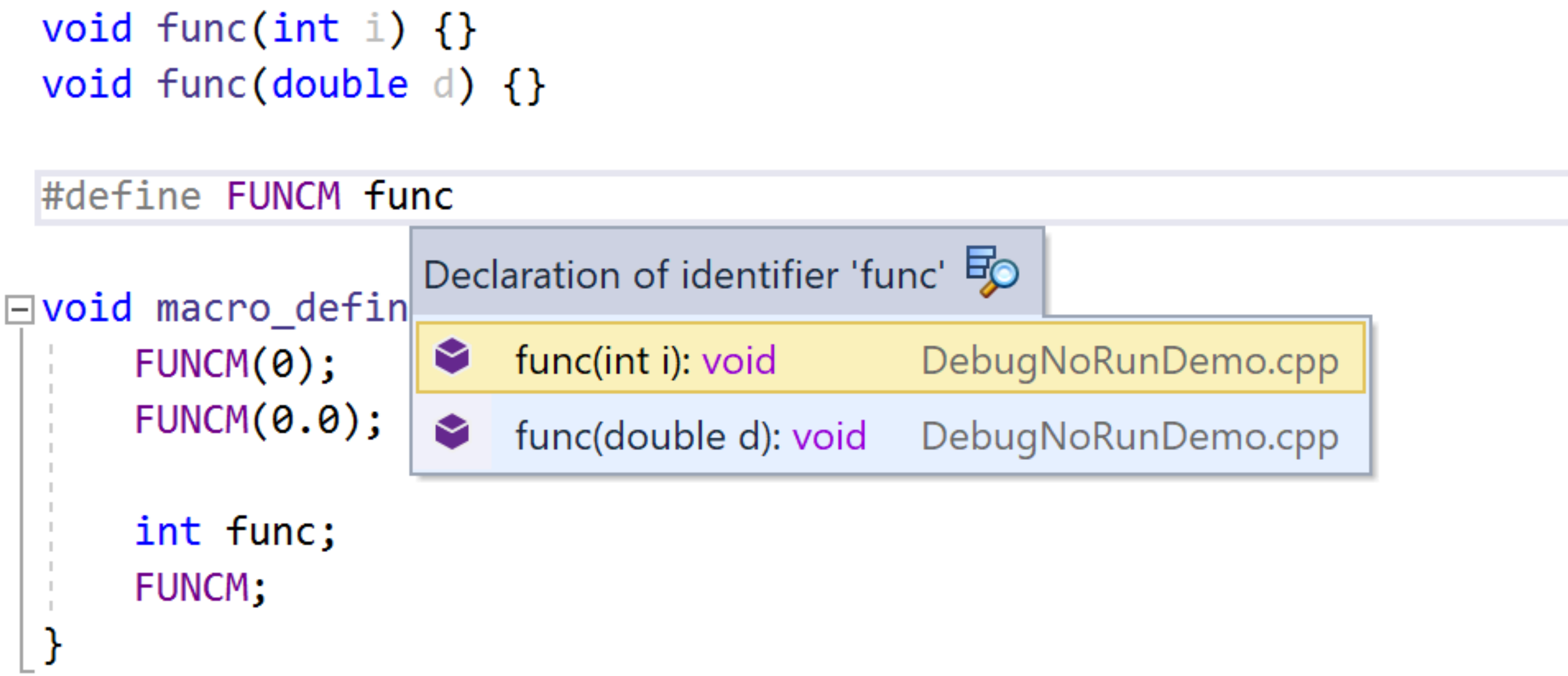


```
void func(int i) {}
void func(double d) {}

#define FUNCM func

void macro_definition_usage() {
    FUNCM(0);
    FUNCM(0.0);

    int func;
    FUNCM;
}
```

The power of tools: Macro debug

—

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



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

#define FUNCM func

void macro_defin
    FUNCM(0);
    FUNCM(0.0);

    int func;
    FUNCM;
}
```

Declaration of identifier 'func'

func(int i): void	DebugNoRunDemo.cpp
func(double d): void	DebugNoRunDemo.cpp

The power of tools:
Type info debug

—

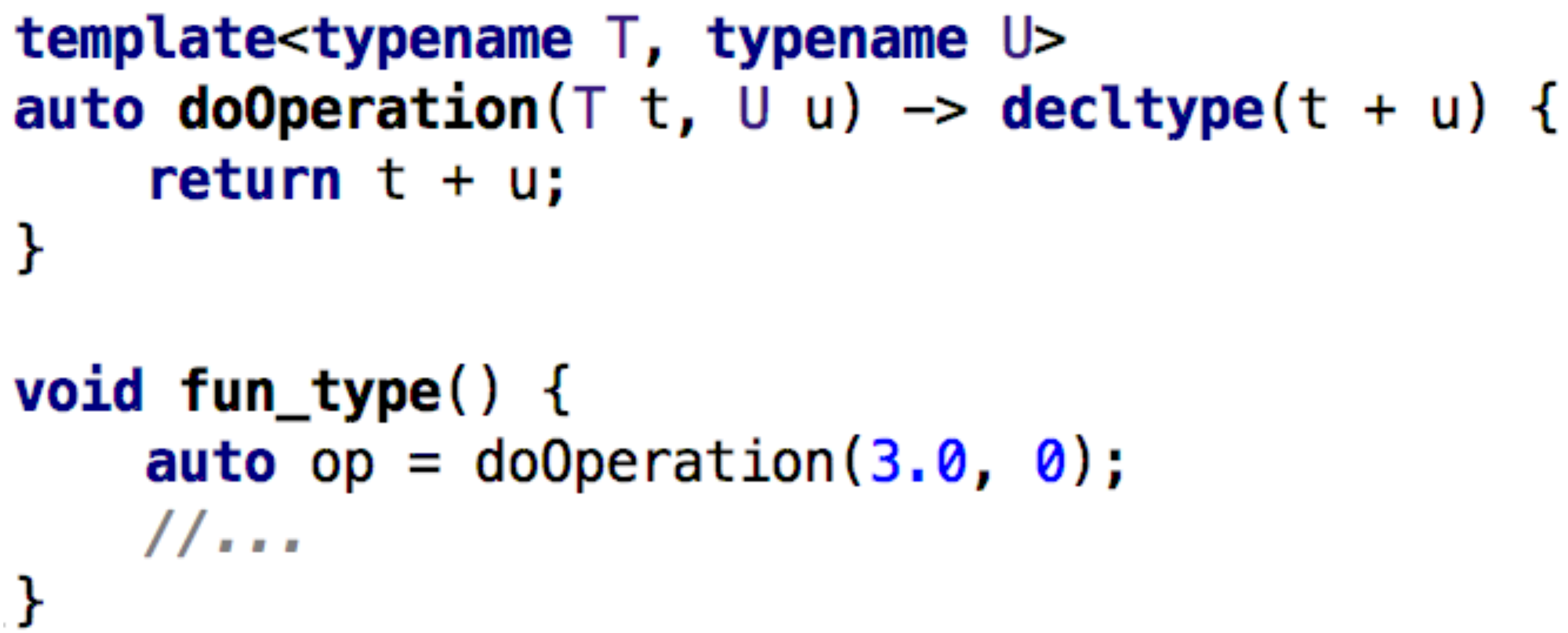
Goal – understand the final type

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

The power of tools:
Type info debug

—

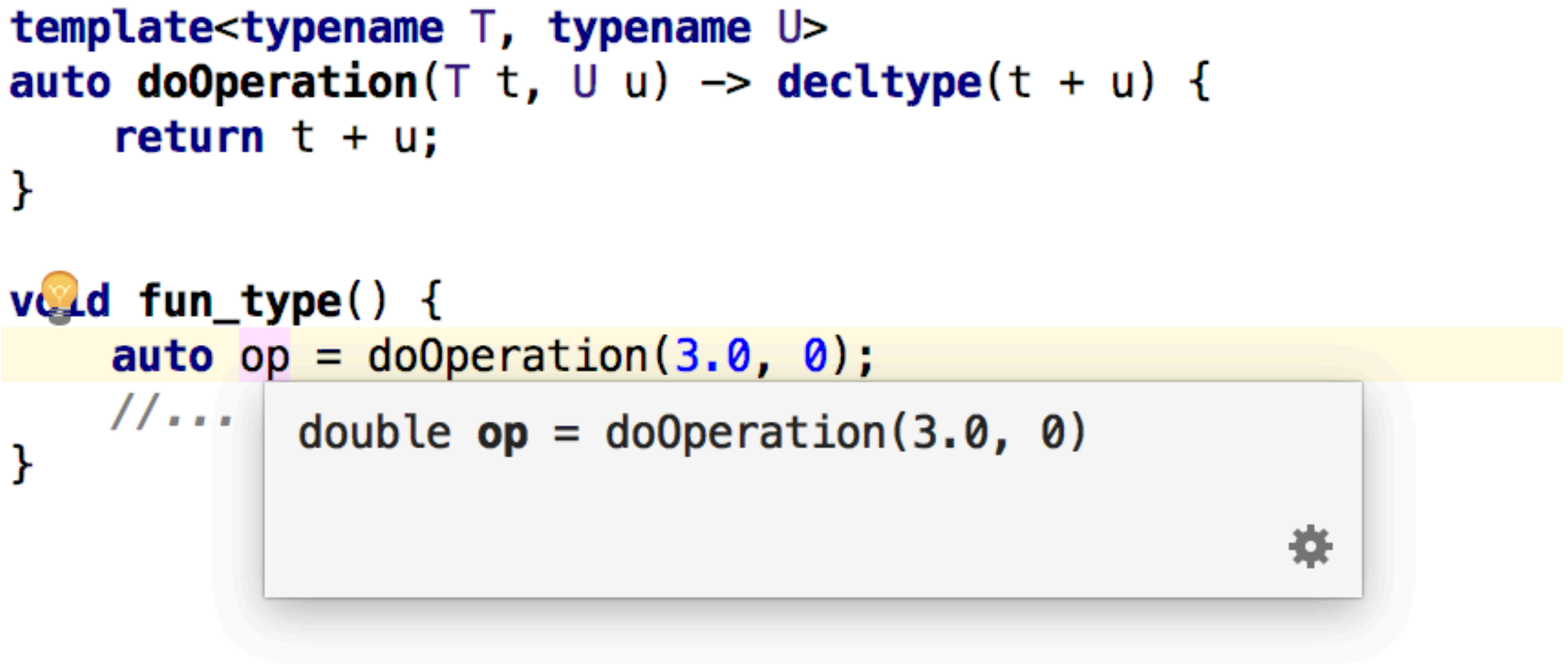
Show inferred type



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

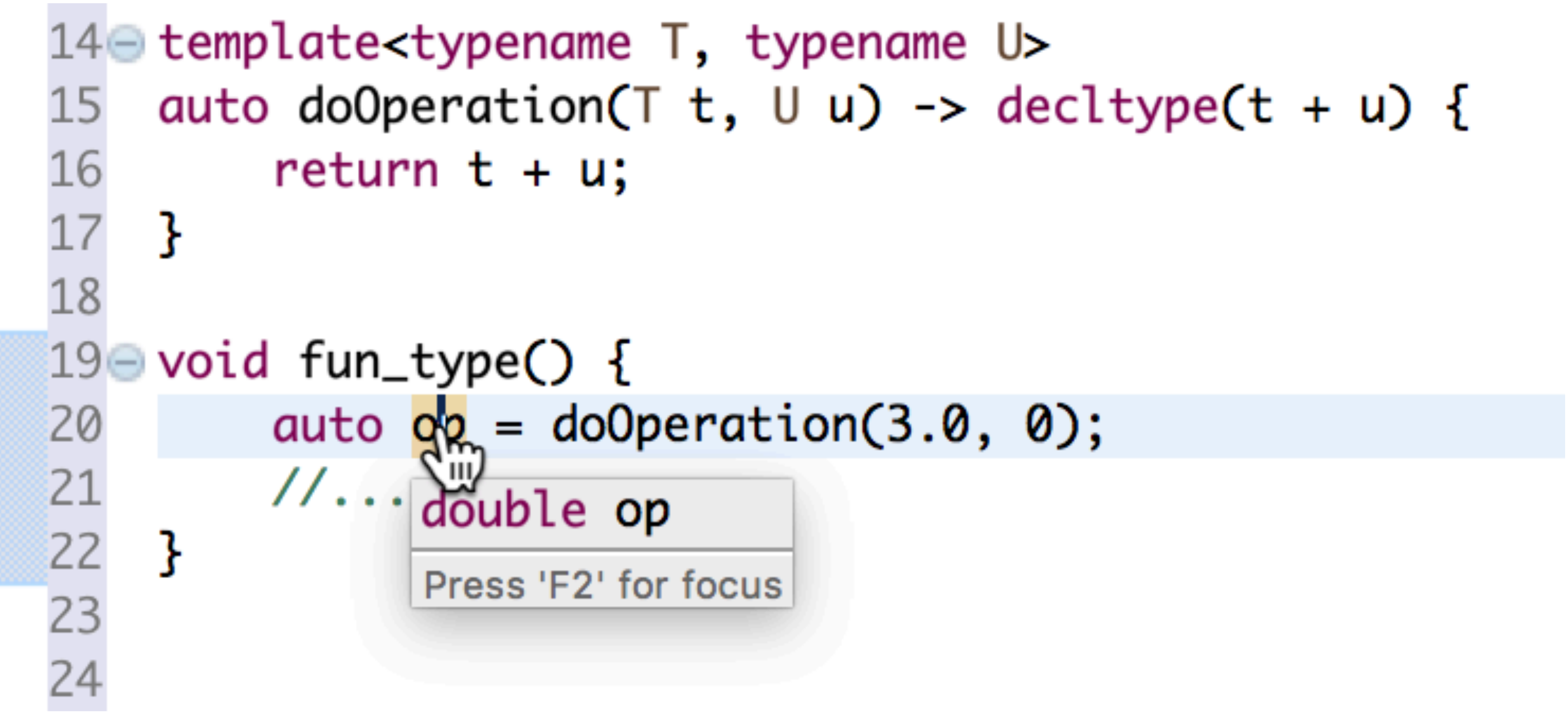
The power of tools: Type info debug

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);
    //...
}
```

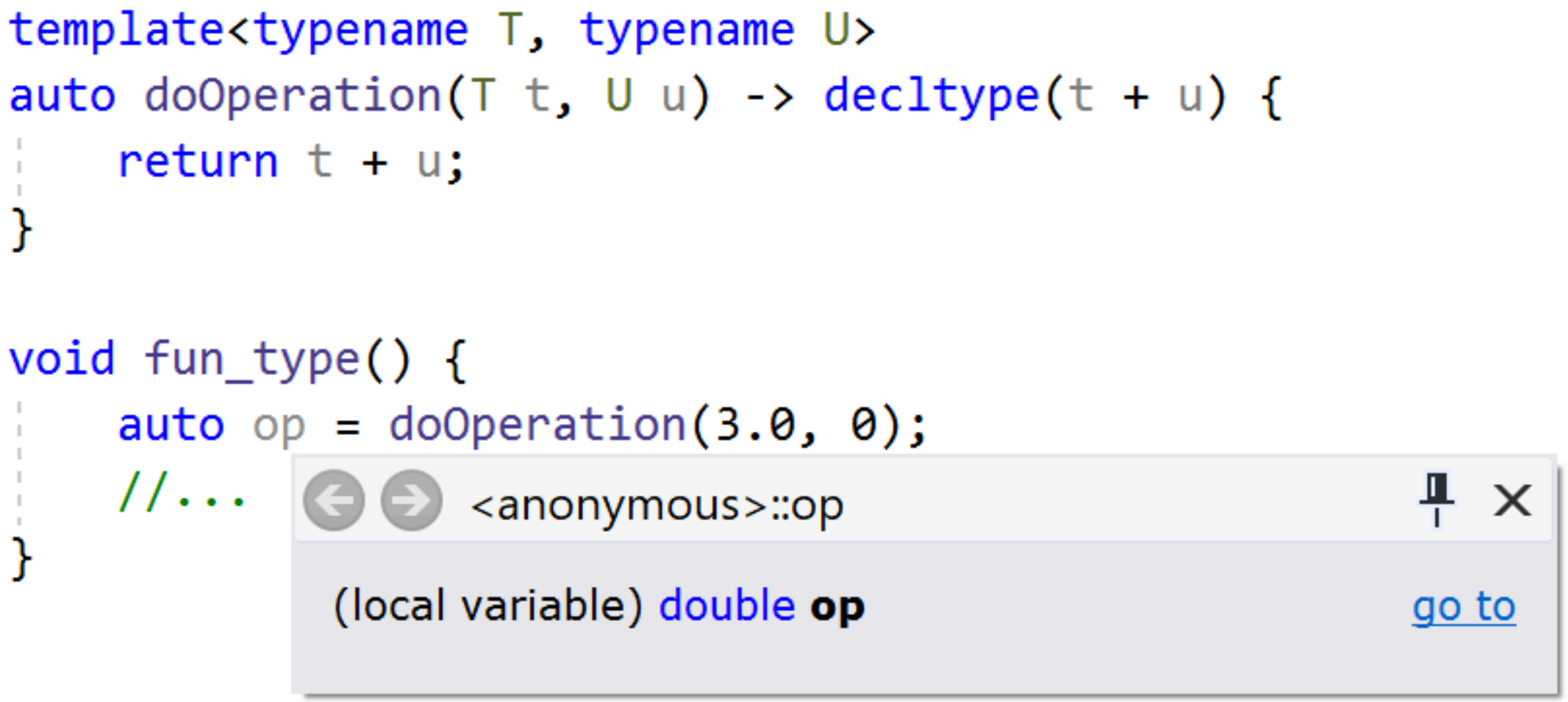
```
double op = doOperation(3.0, 0)
```



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

double op

Press 'F2' for focus



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

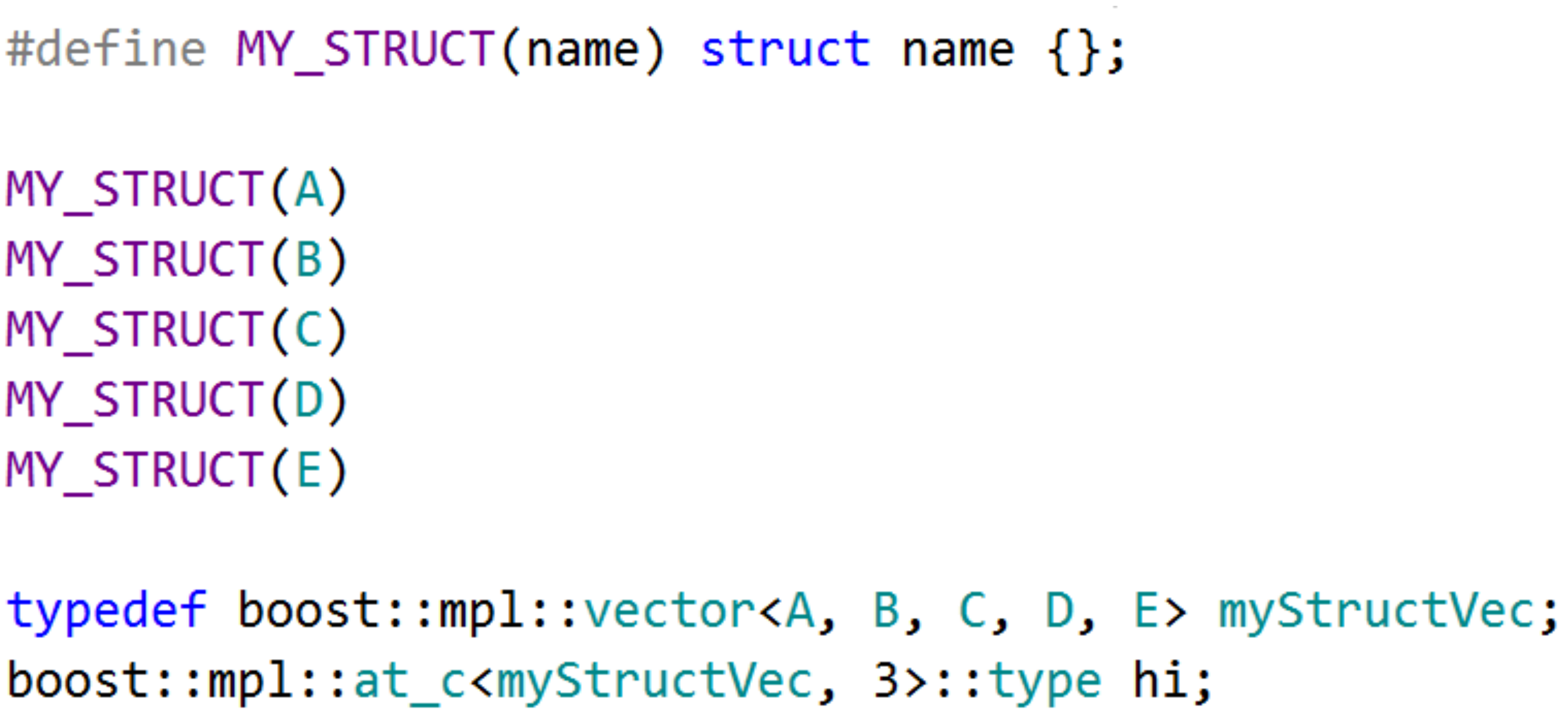
<anonymous>::op

(local variable) double **op**

go to

The power of tools:
Type info debug

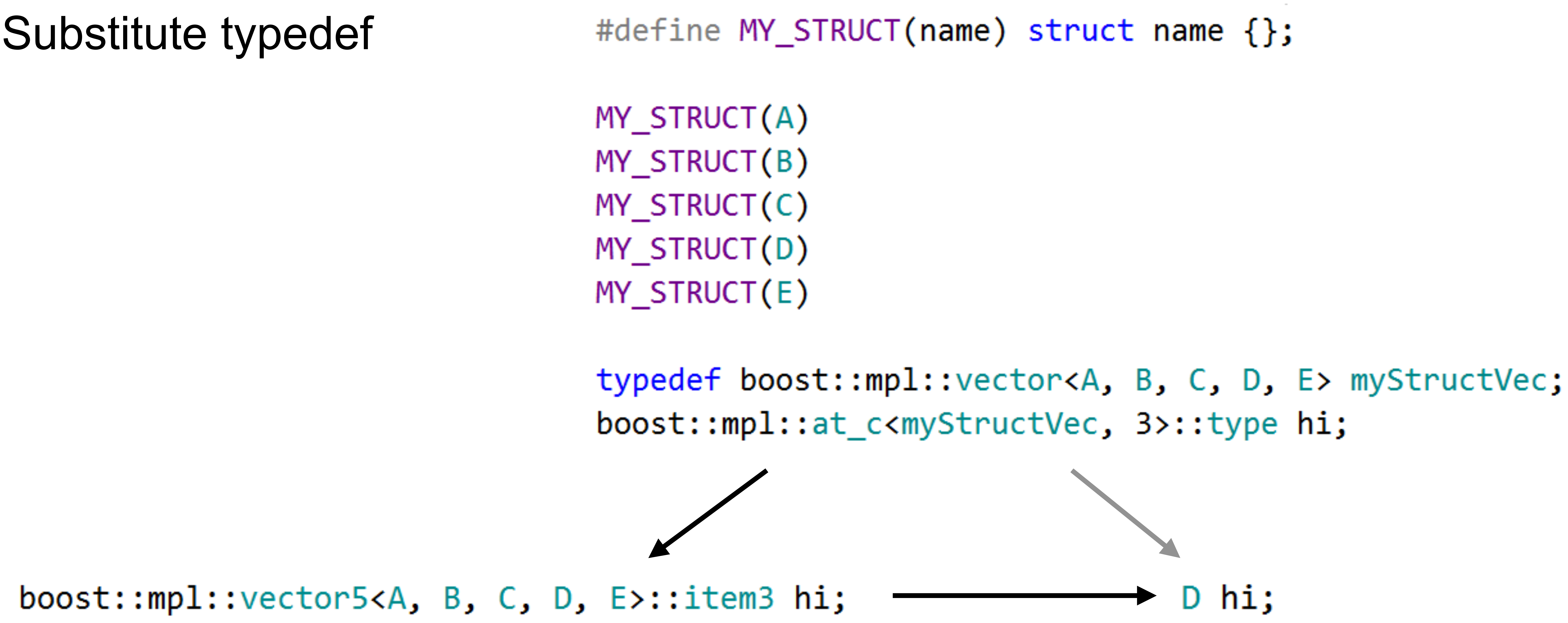
Substitute typedef



```
#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

Substitute typedef



```
#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;
```

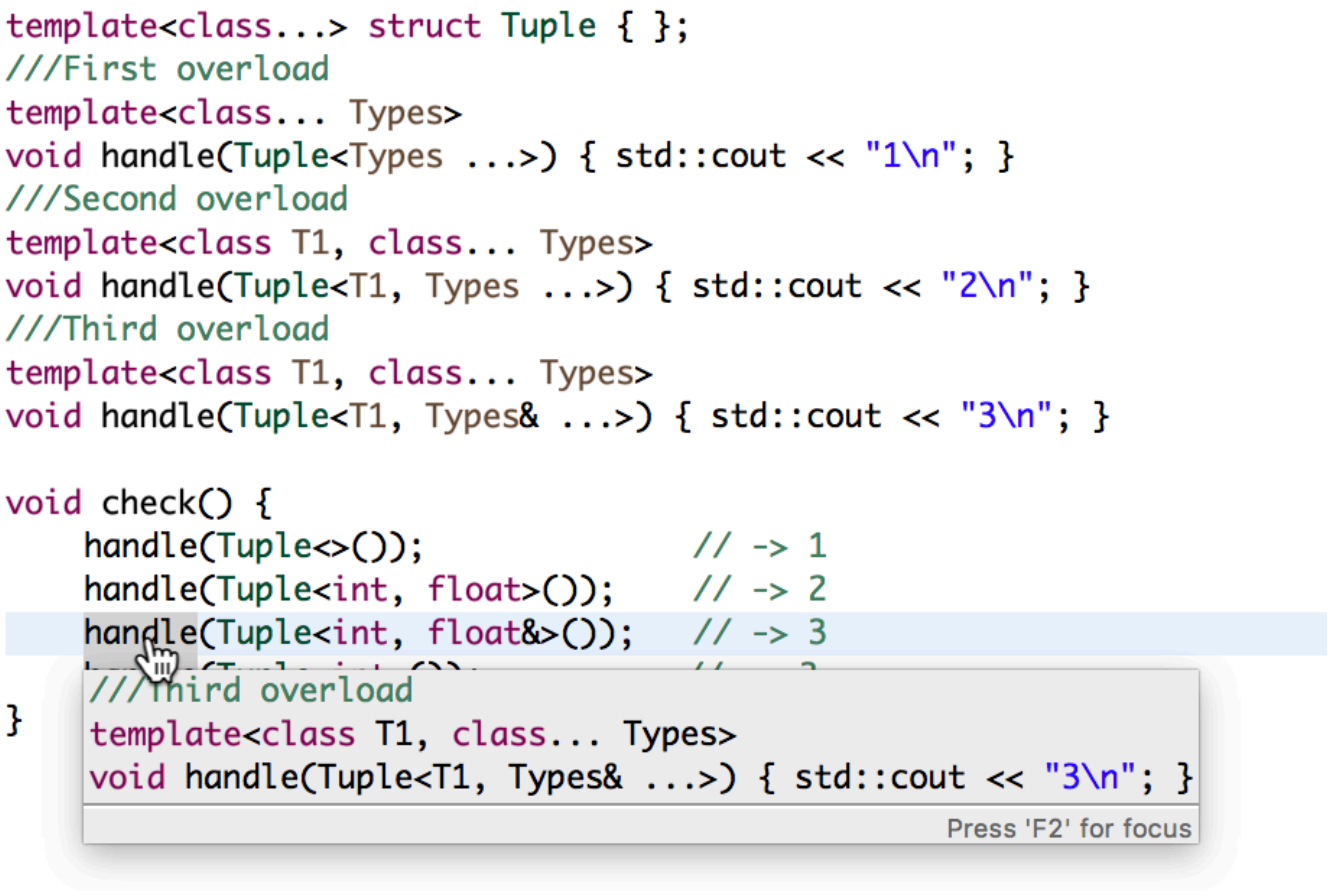
```
D hi;
```

The power of tools: Meta info debug

—

Debug the abstractions

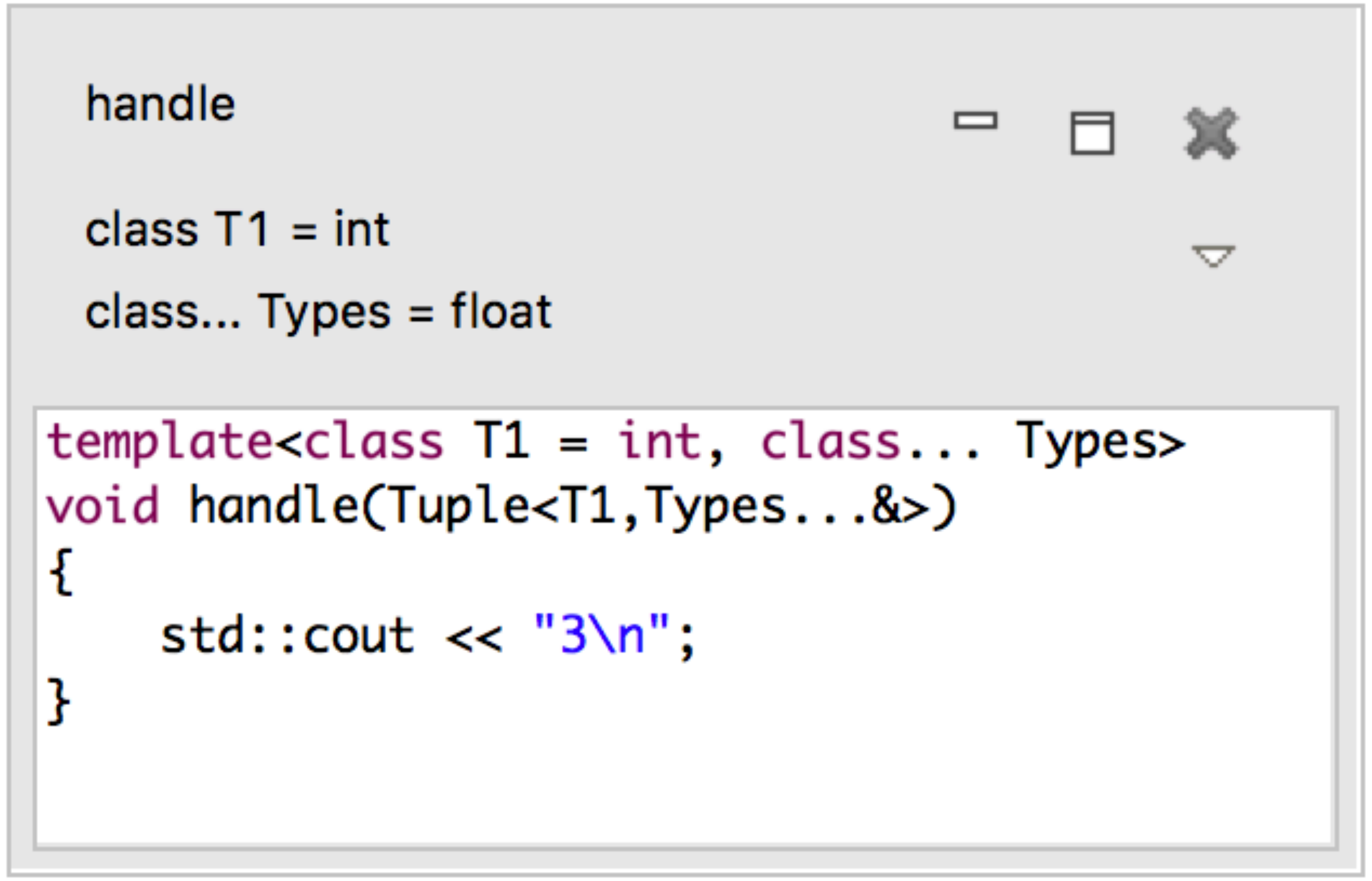
- Instantiating templates
- Constexpr evaluator
- Injection evaluator



```
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

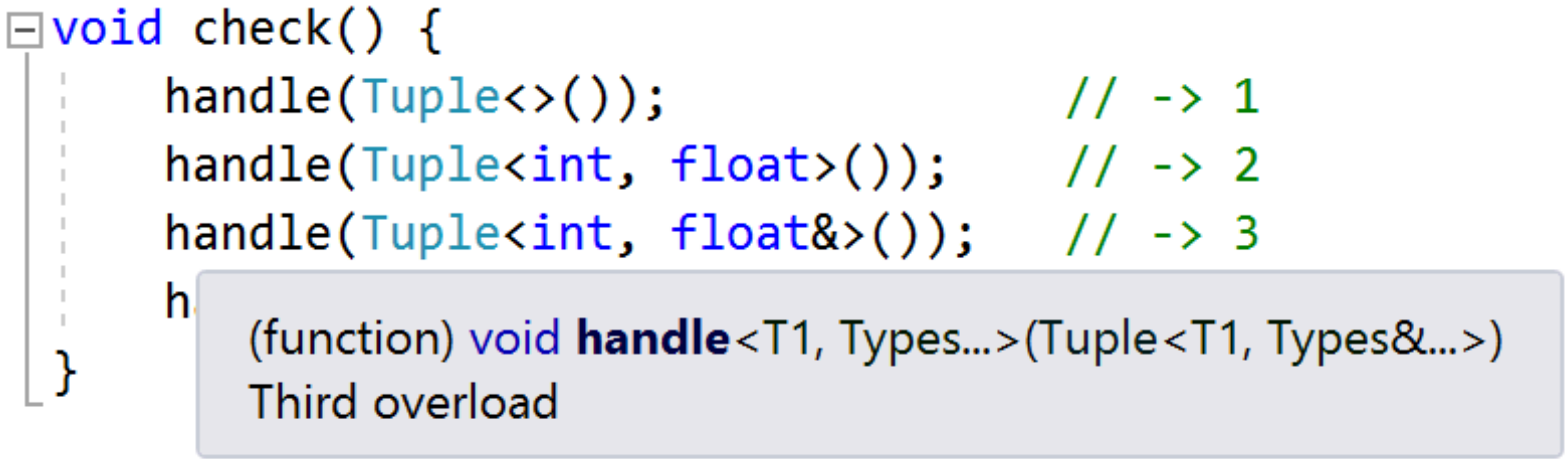


handle

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

Constexpr evaluator +
Template instantiation

```cpp
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: Overloads debug

—

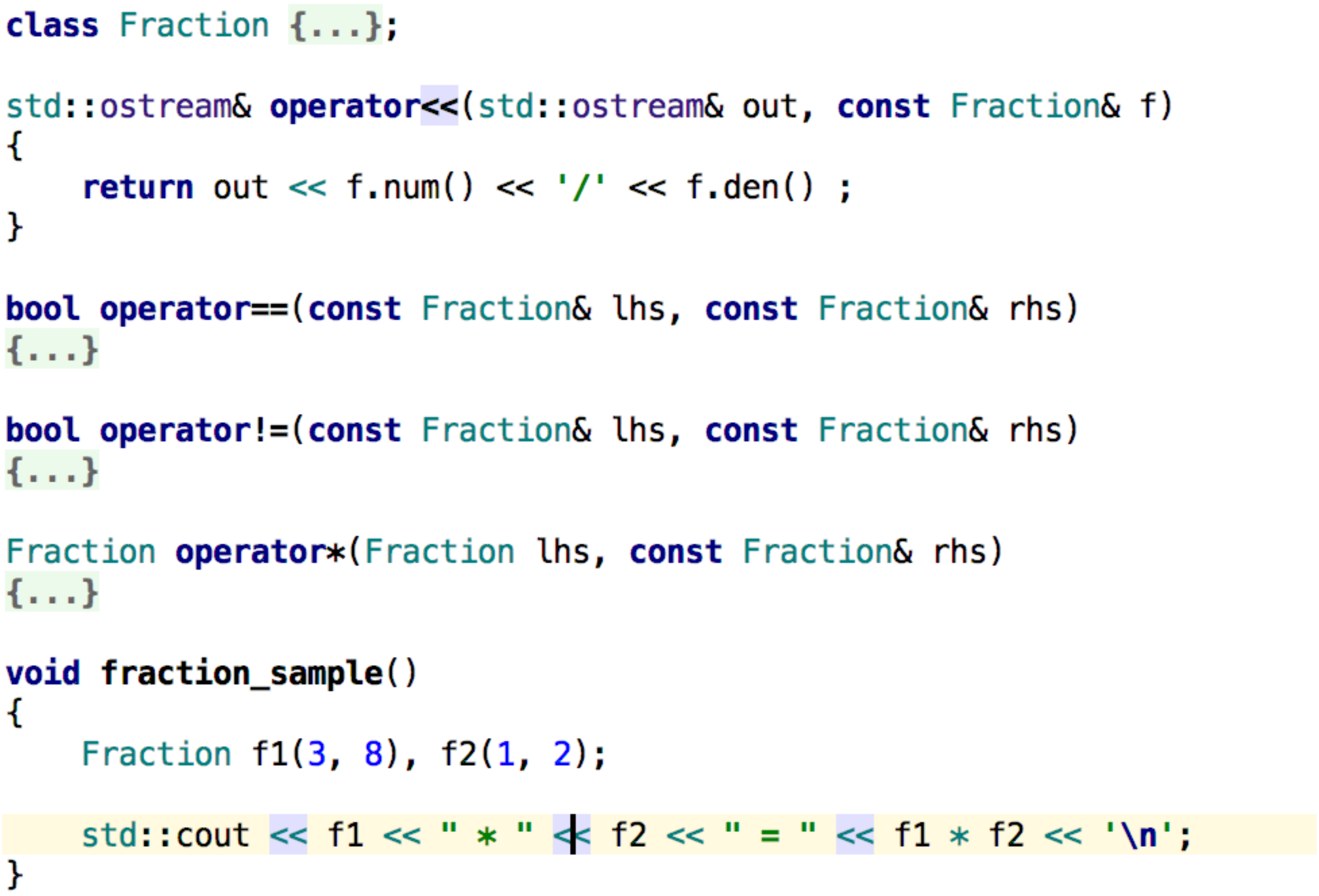
Debug functions and operators overloads:

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

The power of tools: Overloads debug

—

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

—

Overload resolution:

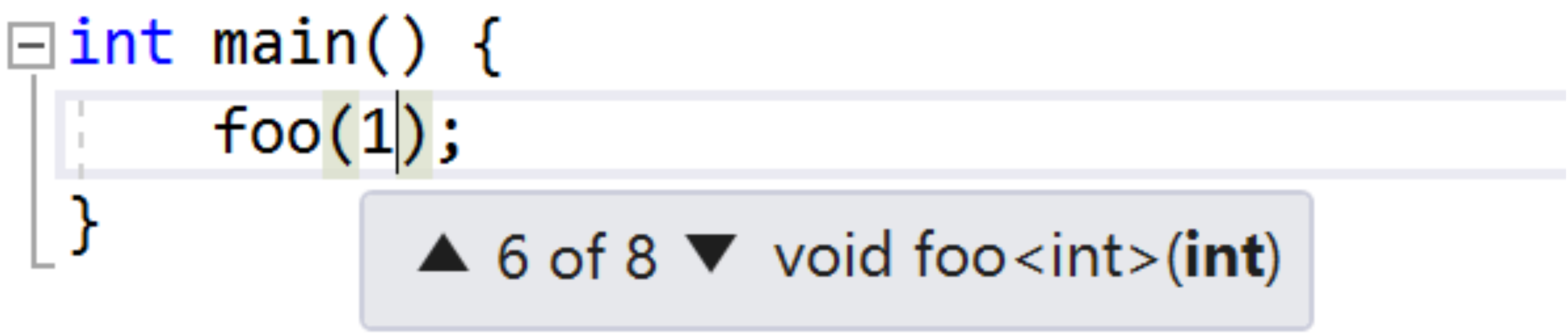
1. Do name lookup
2. Do template argument deduction
3. Pick the candidate
4. Check access control

The power of tools: Overloads debug

—

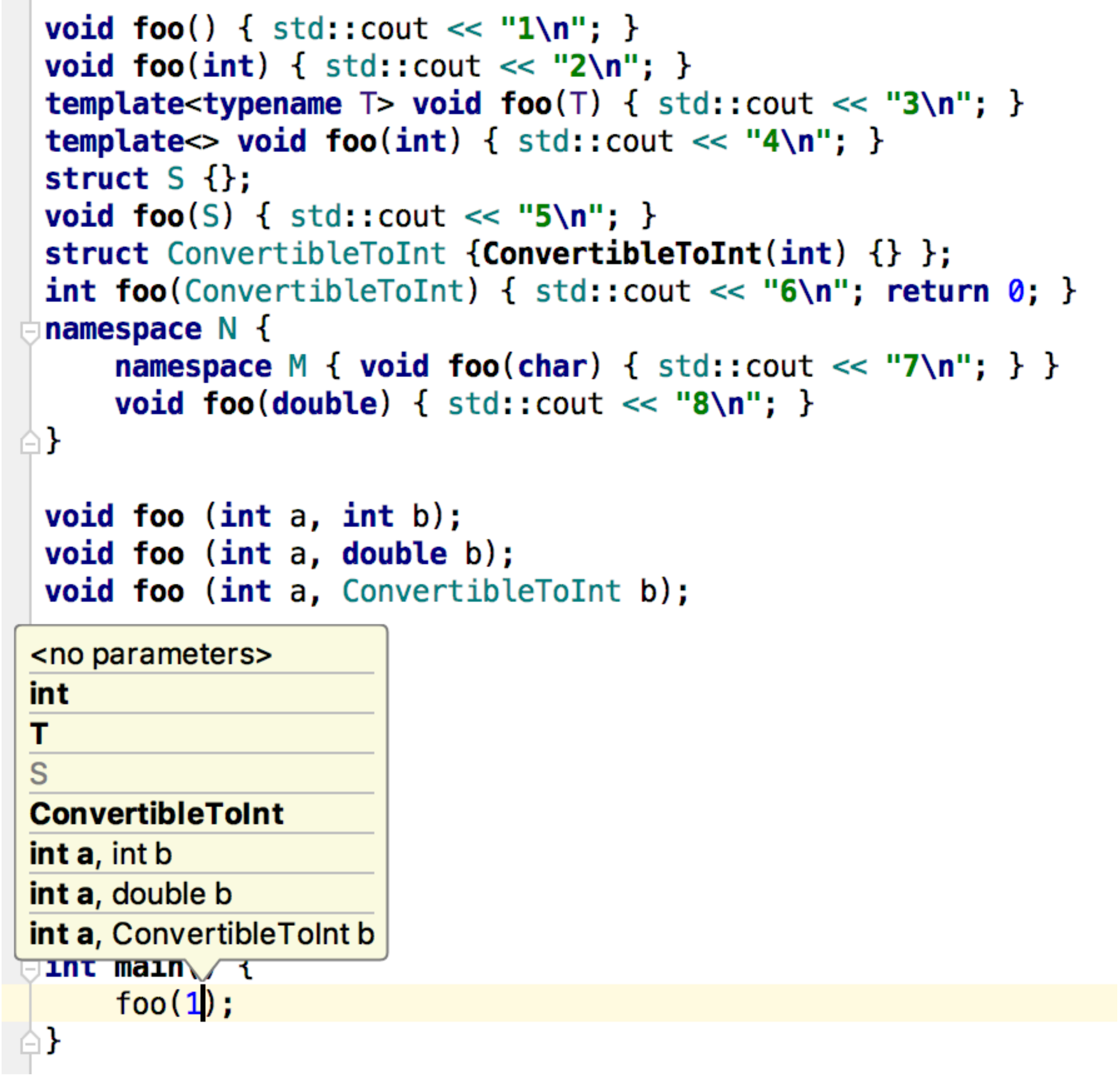
Show candidates set – parameter info

1. One-by-one or all together
2. 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);

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

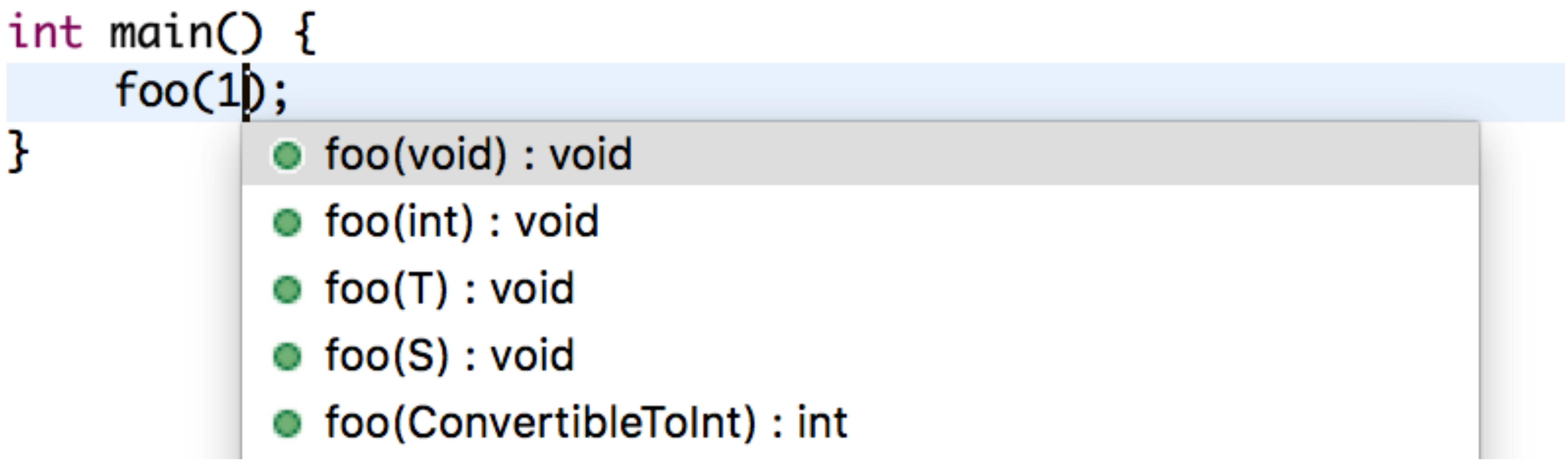
<no parameters>
int
T
S
ConvertibleToInt
int a, int b
int a, double b
int a, ConvertibleToInt b

The power of tools: Overloads debug

—

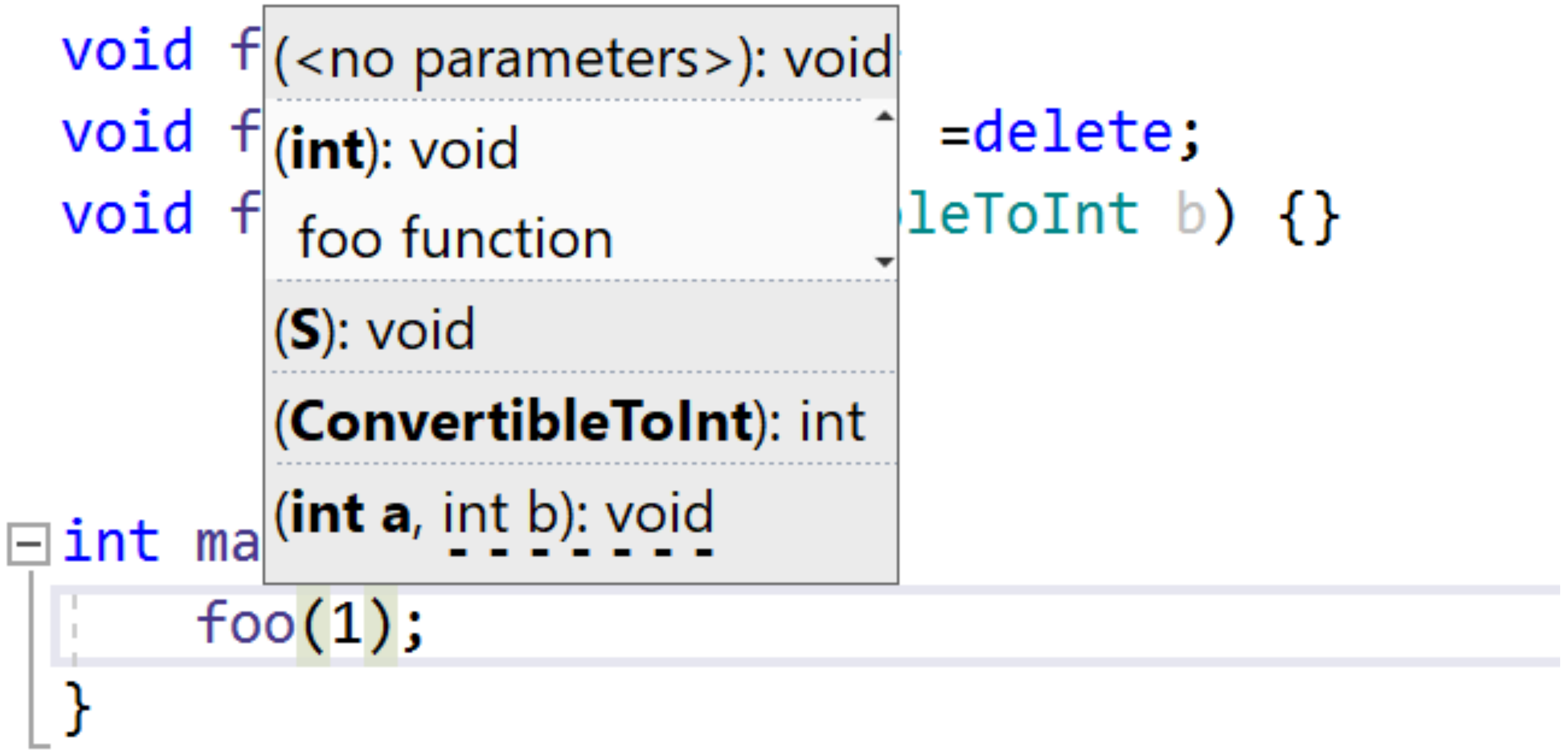
Show candidates set – parameter info

1. One-by-one or all together
2. 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
void f                             =delete;
void f                             leToInt b) {}

int ma
    foo(1);
}
```

(<no parameters>): void

(**int**): void

foo function

(**S**): void

(**ConvertibleToInt**): int

(**int a**, int b): void

The power of tools: Overloads debug

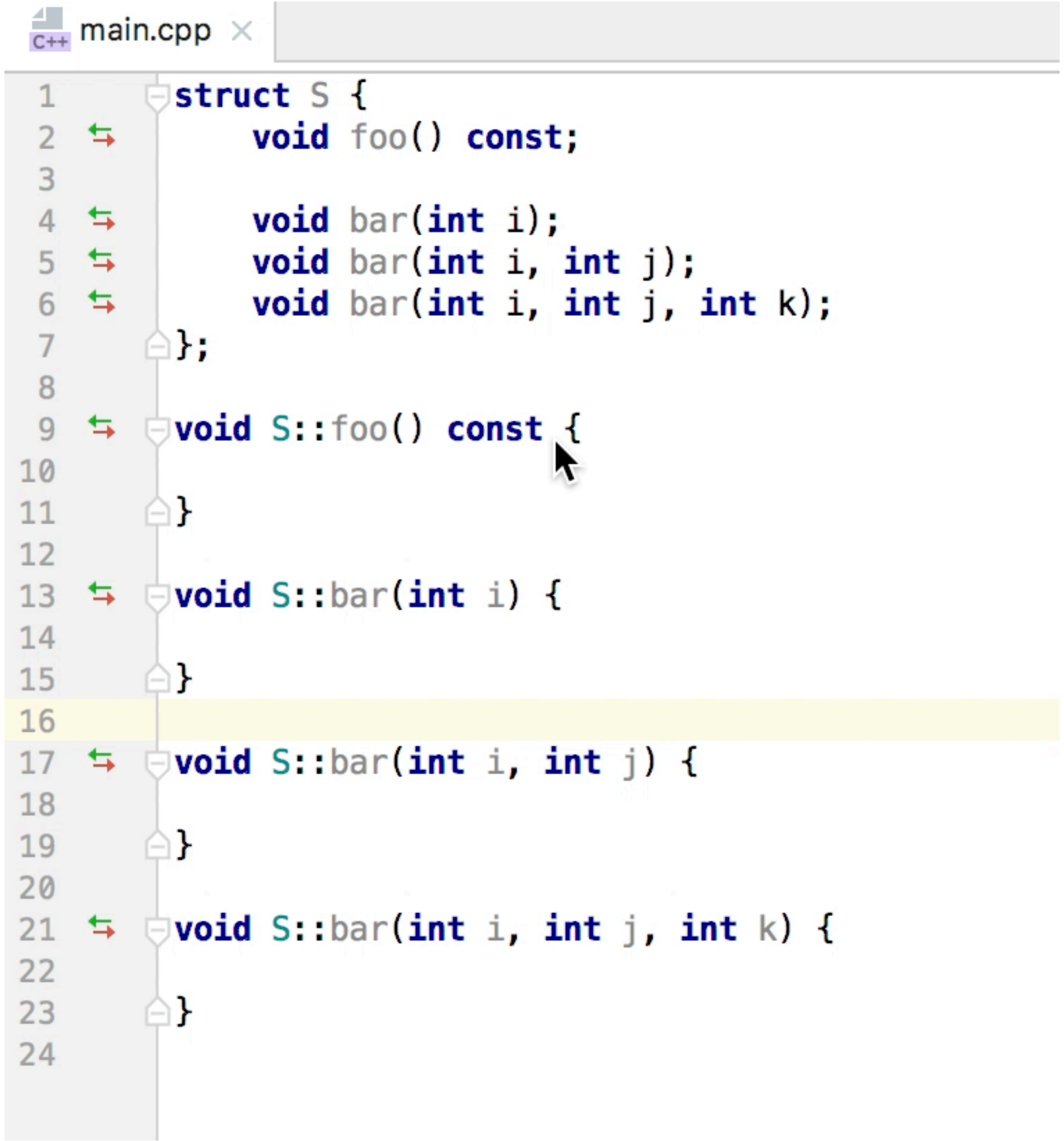
1. Show candidates set
2. Show explanations

?

The power of tools: Overloads debug

—

1. Show candidates set
2. Show explanations
3. Navigate to similar functions/ operators



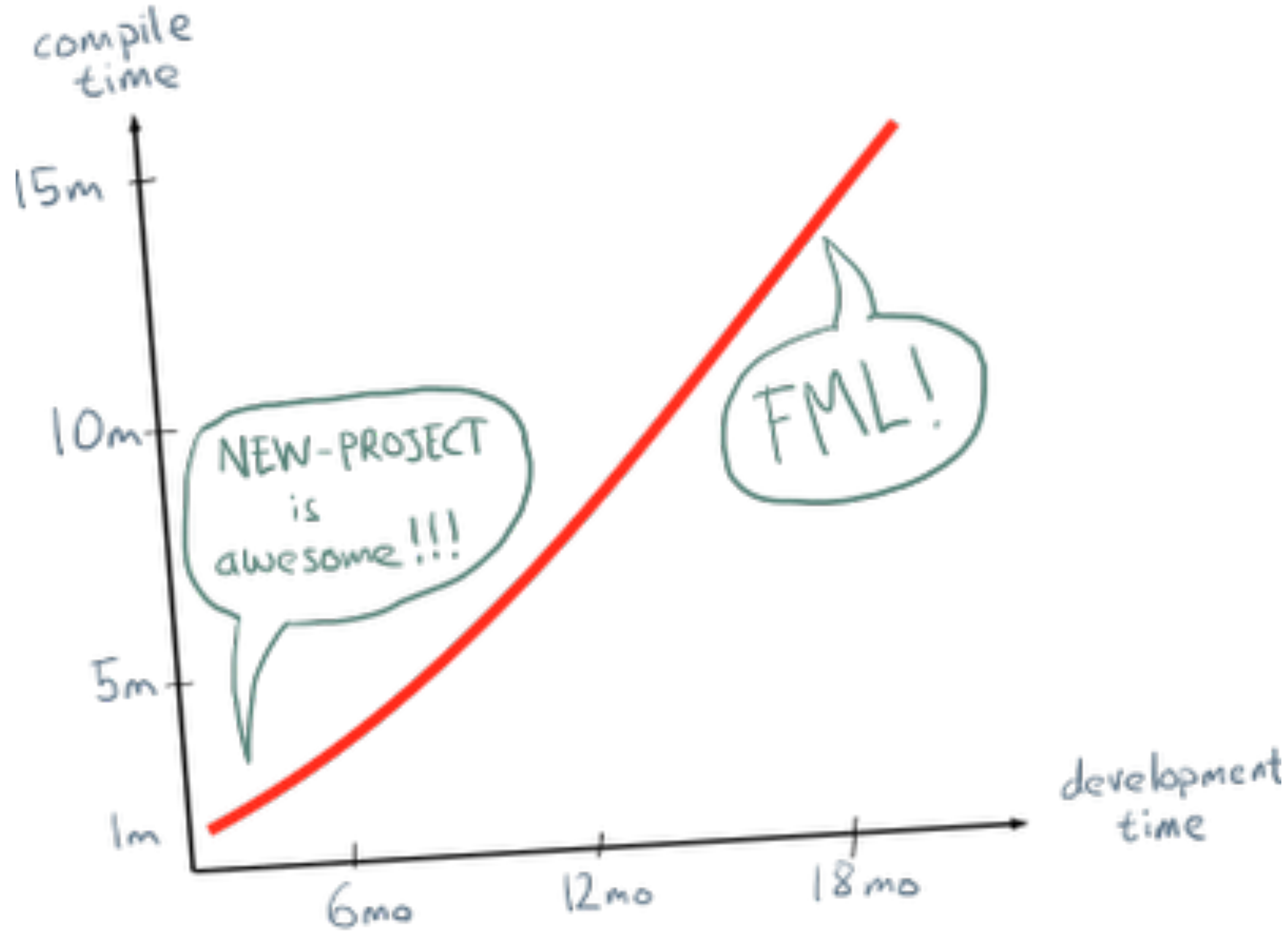
```cpp
struct S {
    void foo() const;

    void bar(int i);
    void bar(int i, int j);
    void bar(int i, int j, int k);
};

void S::foo() const {

}

void S::bar(int i) {

}

void S::bar(int i, int j) {

}

void S::bar(int i, int j, int k) {

}

```

The power of tools:
Includes profiler

—

“Once an #include has been added, it stays” (http://bitsquid.blogspot.co.uk/2011/10/caring-by-sharing-header-hero.html)

Blowup factor = total lines / total lines parsed



The power of tools:
Includes profiler

—

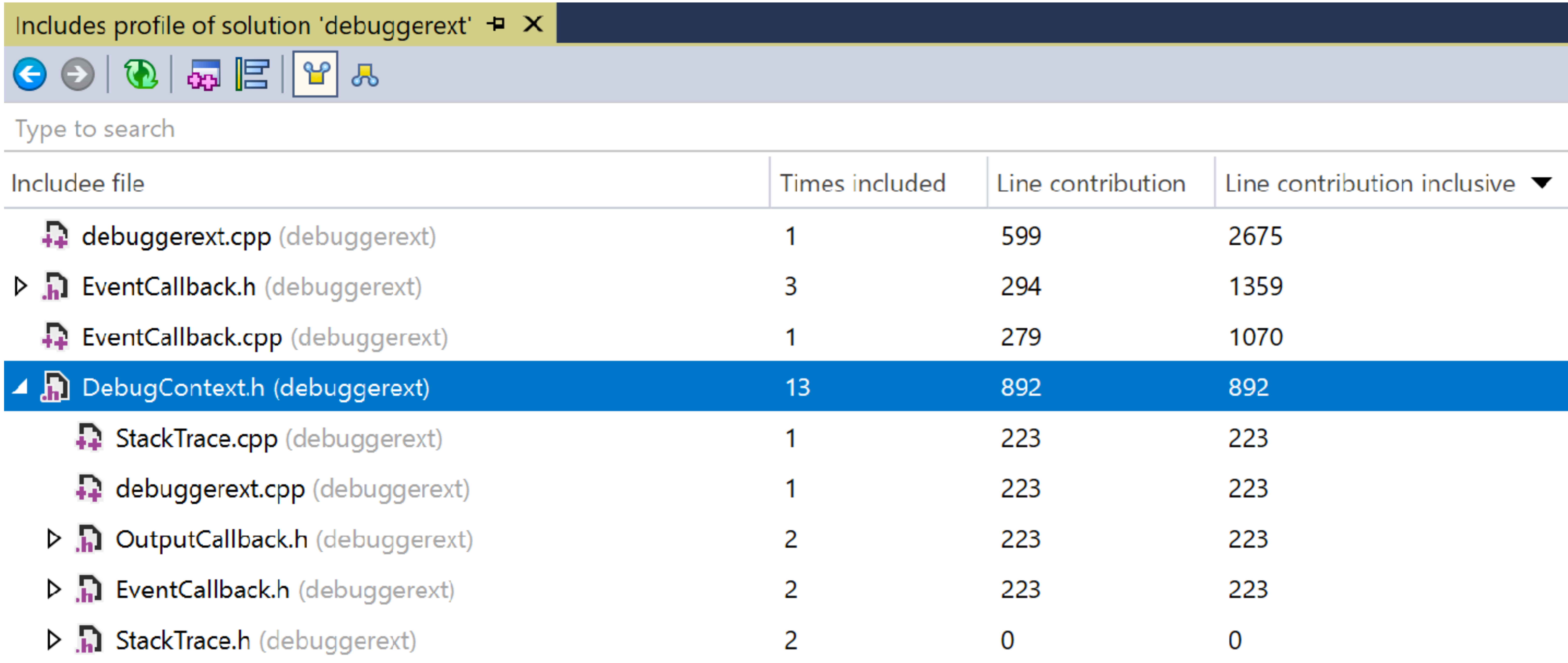
Header heros:

- PCH

The power of tools: Includes profiler

Header heros:

- PCH
- Profilers



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 profiler

—

Header heros:

- PCH
- Profilers
- Optimizers
  - Unused include check
  - Include what you use (and don't include what you don't use)
  - Includator

References

- Bjarne Stroustrup, Writing Good C++14
  - [CppCon 2015] https://www.youtube.com/watch?v=1OEu9C51K2A
- Herbert G. Mayer, ECE 103 Engineering Programming Chapter 7 Compiling C Programs
  - http://slideplayer.com/slide/9665389/
- Herb Sutter, Meta - Thoughts on Generative C++
  - [CppCon 2017] https://www.youtube.com/watch?v=4AfRAVcThyA
- Niklas, bitsquid blog, Caring by Sharing: Header Hero
  - http://bitsquid.blogspot.co.uk/2011/10/caring-by-sharing-header-hero.html

Thank you for your attention

—

Questions?