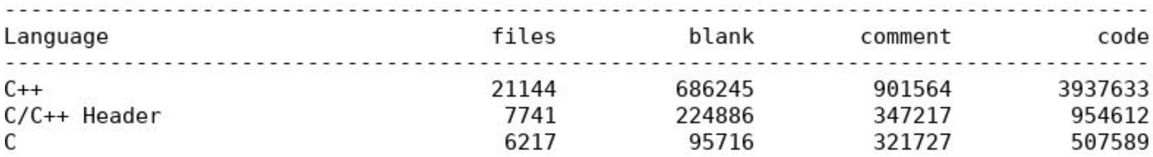
Clang + LLVM

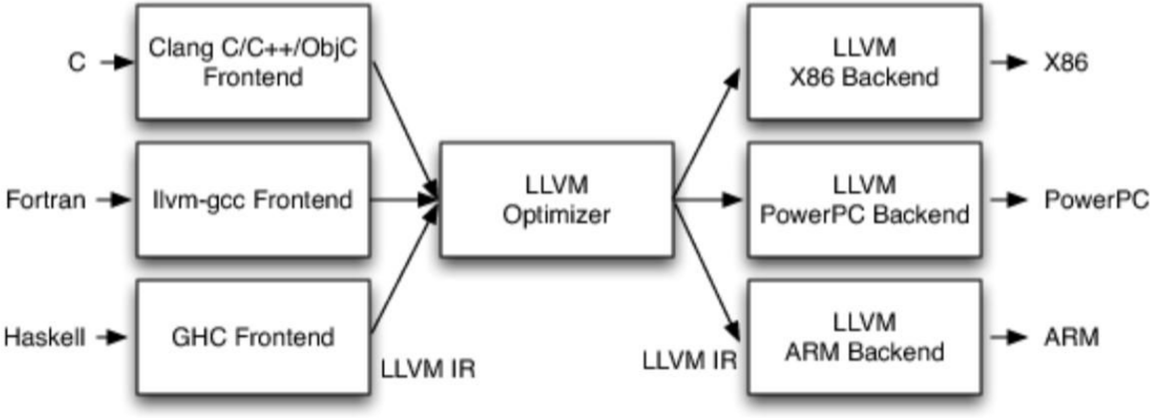
Danila Kutenin, Google

- LLVM = Low Level Virtual Machine
- Began as a research project at the University of Illinois in 2004 https://llvm.org/pubs/2004-01-30-CGO-LLVM.html
- Prank got out of control
- Open source! https://github.com/llvm/llvm-project
- Compilers + linkers + standard libraries + tooling + runtime sanitizers + libc + kernel(?) + magic
- Frontends: ActionScript, Ada, C#, C/C++/Objective-C, Common Lisp, D, Delphi, Fortran, Haskell, Kotlin, Lua, Python, R, Ruby, Rust, Scala, Swift and REALLY MORE
- Backends: ARM, MIPS, PowerPC, x86, x86-64, RISC-V (new, clang-9)
- Works on Linux, FreeBSD, Windows!!!
- Written in C++. 5mln+ lines of actual code
- 180+ contributors, 750+ commits weekly!
- Main contributors: Google, Apple, Intel, IBM, **community**
    - Users: 100% at Google, 99% at Yandex, 100% at Apple



```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                          21144         686245         901564        3937633
C/C++ Header                  7741         224886         347217         954612
C                             6217          95716         321727         507589
```

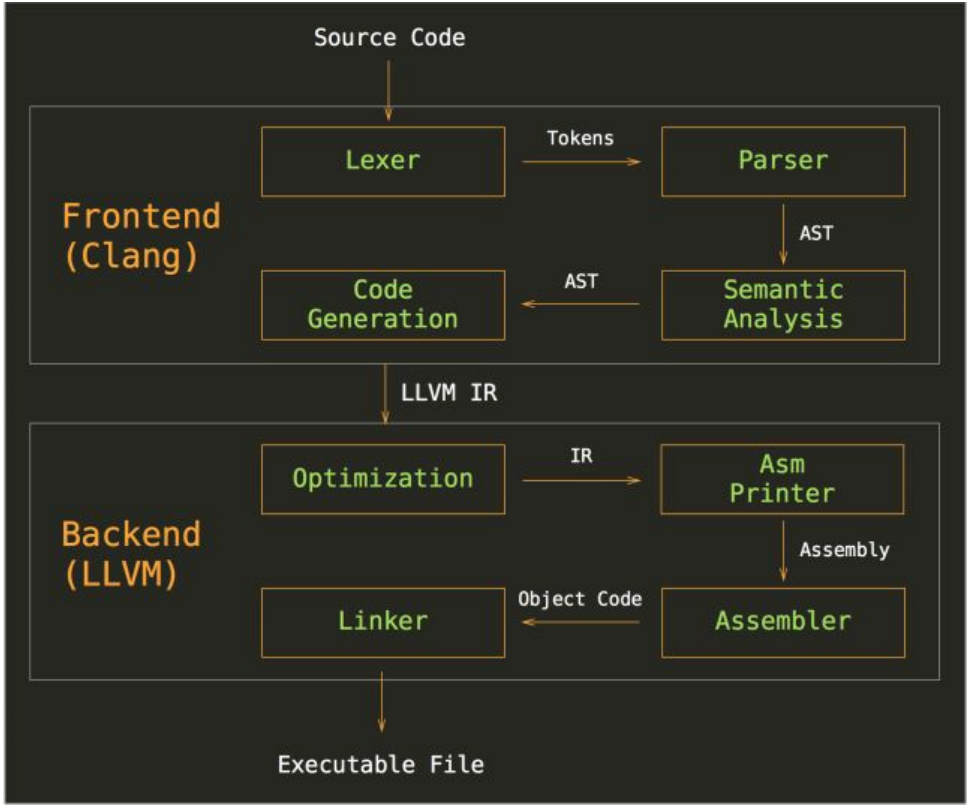
Three phase design



- Frontend: Lexer (Tokenizer), Parser, AST -> IR
- Optimizer: various passes on IR
- Backend: platform code generation

IR -- intermediate representation

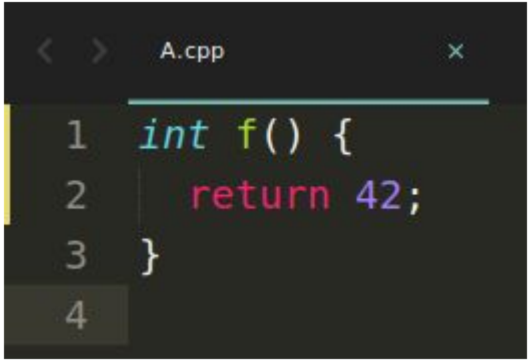
Three phase design



Frontend

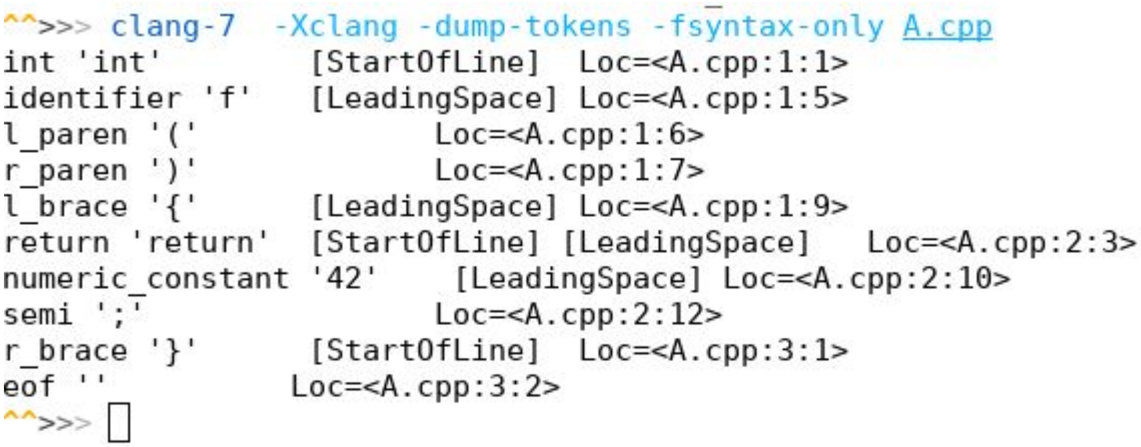
- Token parsing
- AST (key idea of this lecture)
- Semantic analysis
- Challenges:
  - Better error codes. It is really complex in C++, really! Fun links
  - Good AST representation to understand what and why
  - Good initial code generation for IR
  - Hard semantics compliance

Frontend



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

```

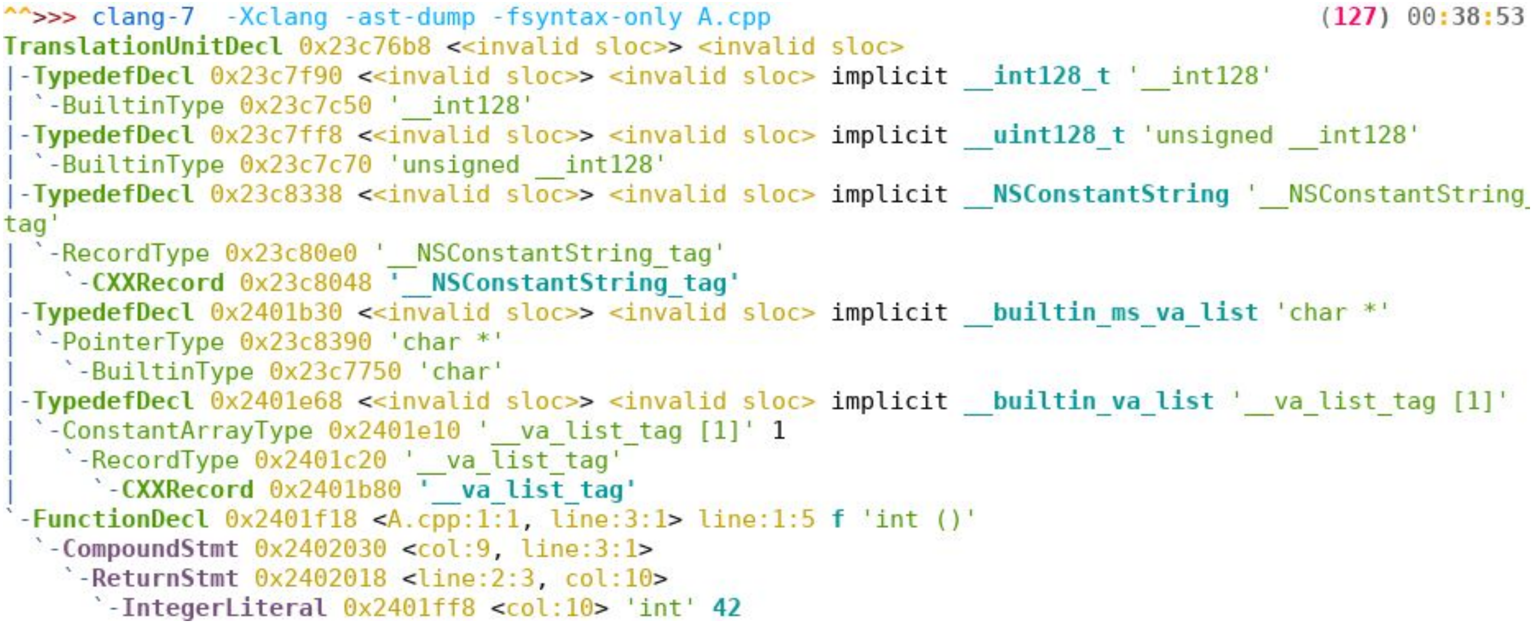
clang++ -Xclang -dump-tokens -fsyntax-only



```
^^>>> clang-7  -Xclang -dump-tokens -fsyntax-only A.cpp
int 'int'        [StartOfLine]  Loc=<A.cpp:1:1>
identifier 'f'   [LeadingSpace] Loc=<A.cpp:1:5>
l_paren '('             Loc=<A.cpp:1:6>
r_paren ')'             Loc=<A.cpp:1:7>
l_brace '{'      [LeadingSpace] Loc=<A.cpp:1:9>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<A.cpp:2:3>
numeric_constant '42'    [LeadingSpace] Loc=<A.cpp:2:10>
semi ';'                Loc=<A.cpp:2:12>
r_brace '}'      [StartOfLine]  Loc=<A.cpp:3:1>
eof ''          Loc=<A.cpp:3:2>
^^>>> 
```

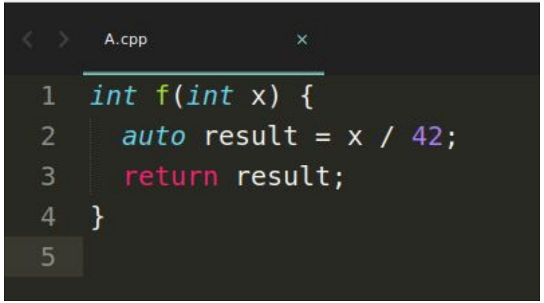
Abstract Syntax Tree

clang++ -Xclang -ast-dump -fsyntax-only



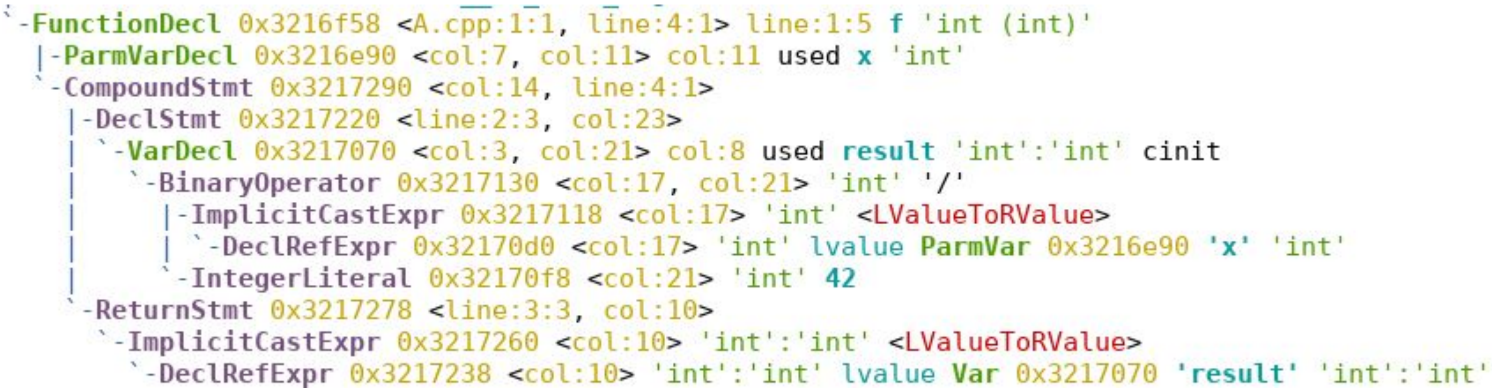
```
^^>>> clang-7  -Xclang -ast-dump -fsyntax-only A.cpp                                      (127) 00:38:53
TranslationUnitDecl 0x23c76b8 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x23c7f90 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x23c7c50 '__int128'
|-TypedefDecl 0x23c7ff8 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x23c7c70 'unsigned __int128'
|-TypedefDecl 0x23c8338 <<invalid sloc>> <invalid sloc> implicit __NSConstantString '__NSConstantString_
tag'
| `-RecordType 0x23c80e0 '__NSConstantString_tag'
|   `-CXXRecord 0x23c8048 '__NSConstantString_tag'
|-TypedefDecl 0x2401b30 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x23c8390 'char *'
|   `-BuiltinType 0x23c7750 'char'
|-TypedefDecl 0x2401e68 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list '__va_list_tag [1]'
| `-ConstantArrayType 0x2401e10 '__va_list_tag [1]' 1
|   `-RecordType 0x2401c20 '__va_list_tag'
|     `-CXXRecord 0x2401b80 '__va_list_tag'
`-FunctionDecl 0x2401f18 <A.cpp:1:1, line:3:1> line:1:5 f 'int ()'
  `-CompoundStmt 0x2402030 <col:9, line:3:1>
    `-ReturnStmt 0x2402018 <line:2:3, col:10>
      `-IntegerLiteral 0x2401ff8 <col:10> 'int' 42
```

Abstract Syntax Tree



```
int f(int x) {
  auto result = x / 42;
  return result;
}
```

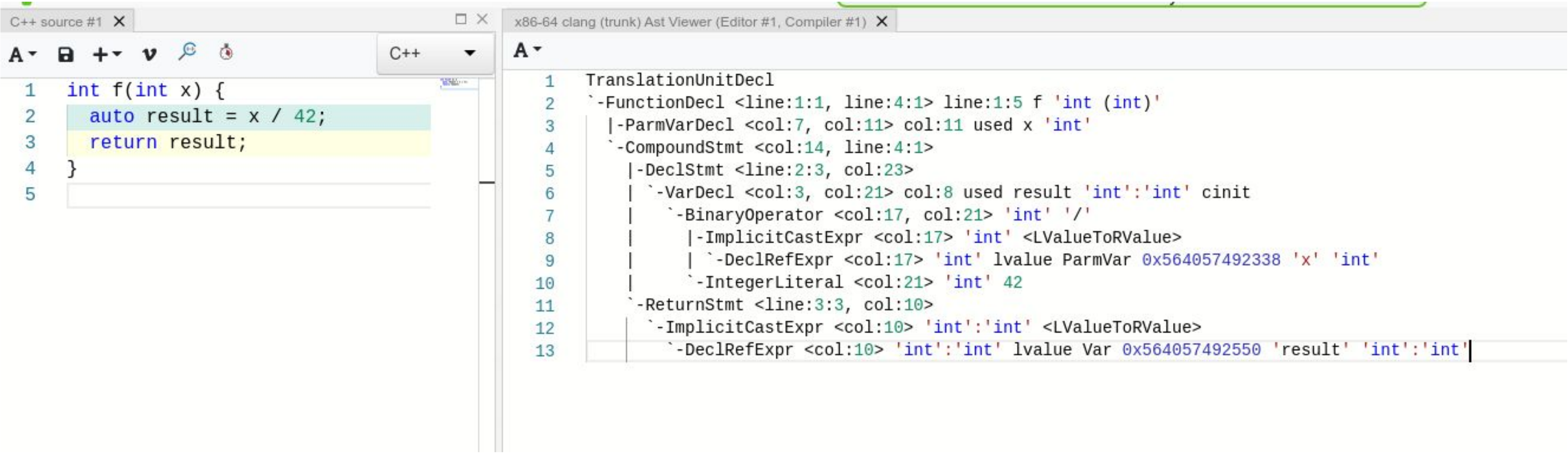
clang++ -Xclang -ast-dump -fsyntax-only



```
`-FunctionDecl 0x3216f58 <A.cpp:1:1, line:4:1> line:1:5 f 'int (int)'
  |-ParmVarDecl 0x3216e90 <col:7, col:11> col:11 used x 'int'
  `-CompoundStmt 0x3217290 <col:14, line:4:1>
    |-DeclStmt 0x3217220 <line:2:3, col:23>
    | `-VarDecl 0x3217070 <col:3, col:21> col:8 used result 'int':'int' cinit
    |   `-BinaryOperator 0x3217130 <col:17, col:21> 'int' '/'
    |     |-ImplicitCastExpr 0x3217118 <col:17> 'int' <LValueToRValue>
    |     | `-DeclRefExpr 0x32170d0 <col:17> 'int' lvalue ParmVar 0x3216e90 'x' 'int'
    |     `-IntegerLiteral 0x32170f8 <col:21> 'int' 42
    `-ReturnStmt 0x3217278 <line:3:3, col:10>
      `-ImplicitCastExpr 0x3217260 <col:10> 'int':'int' <LValueToRValue>
        `-DeclRefExpr 0x3217238 <col:10> 'int':'int' lvalue Var 0x3217070 'result' 'int':'int'
```

Abstract Syntax Tree
Do it in web!

https://gcc.godbolt.org/z/pvvJAf



```cpp
int f(int x) {
  auto result = x / 42;
  return result;
}

```

```
TranslationUnitDecl
`-FunctionDecl <line:1:1, line:4:1> line:1:5 f 'int (int)'
  |-ParmVarDecl <col:7, col:11> col:11 used x 'int'
  `-CompoundStmt <col:14, line:4:1>
    |-DeclStmt <line:2:3, col:23>
    | `-VarDecl <col:3, col:21> col:8 used result 'int':'int' cinit
    |   `-BinaryOperator <col:17, col:21> 'int' '/'
    |     |-ImplicitCastExpr <col:17> 'int' <LValueToRValue>
    |     | `-DeclRefExpr <col:17> 'int' lvalue ParmVar 0x564057492338 'x' 'int'
    |     `-IntegerLiteral <col:21> 'int' 42
    `-ReturnStmt <line:3:3, col:10>
      `-ImplicitCastExpr <col:10> 'int':'int' <LValueToRValue>
        `-DeclRefExpr <col:10> 'int':'int' lvalue Var 0x564057492550 'result' 'int':'int'
```

Godbolt https://gcc.godbolt.org
Used for checks and fast progress

- Web compiler explorer
- Many insight features
- Hell of the compiler versions

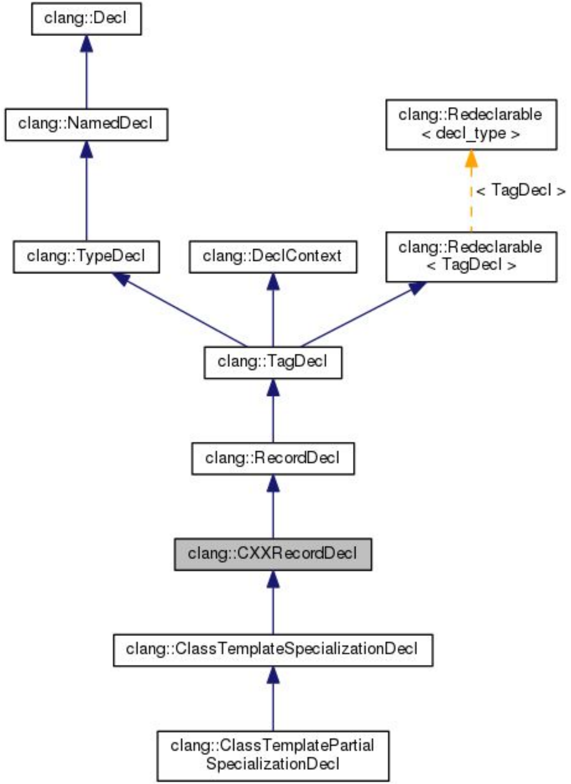
RecursiveASTVisitor

- traverse the AST (i.e. go to each node);
- at a given node, walk up the class hierarchy, starting from the node's CXXRecordDecl, until the top-most class (e.g. Stmt, Decl, or Type) is reached.
- given a (node, class) combination, where 'class' is some base class of the dynamic type of 'node', call a user overridable function to actually visit the node.

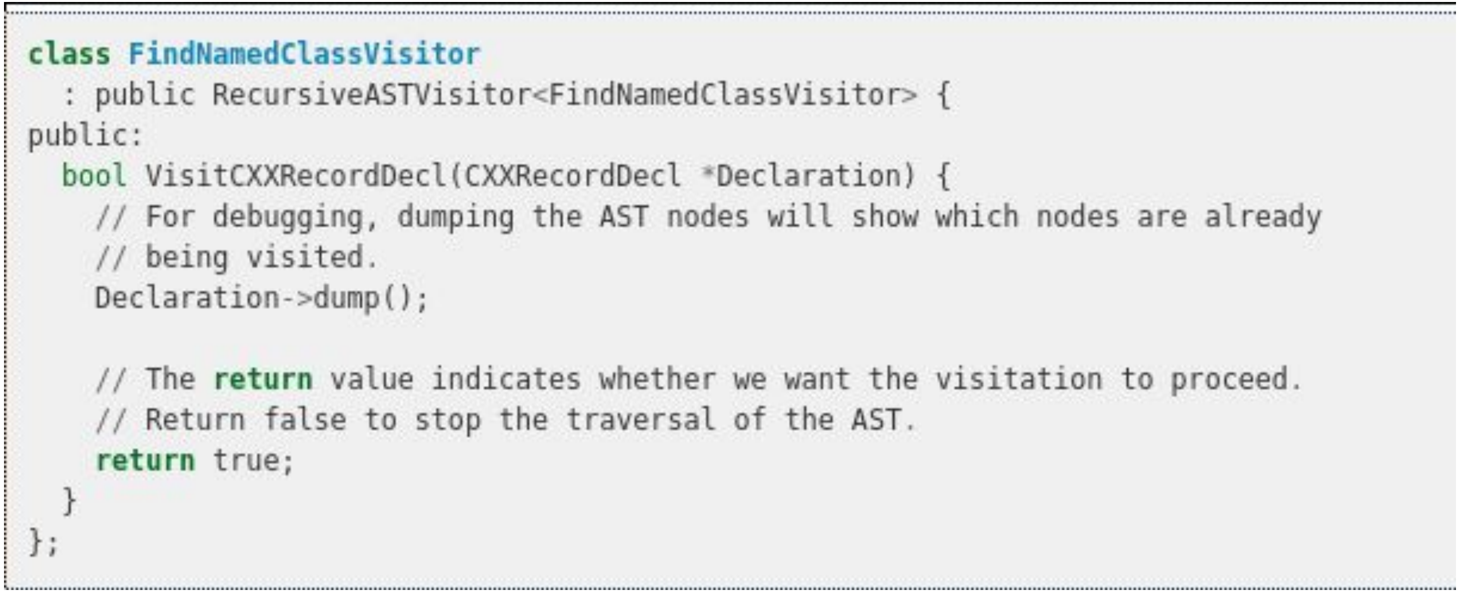
https://clang.llvm.org/docs/RAVFrontendAction.html

RecursiveASTVisitor

Inheritance diagram for clang::CXXRecordDecl:


clang::Decl
clang::NamedDecl
clang::Redeclarable < decl_type >
< TagDecl >
clang::TypeDecl
clang::DeclContext
clang::Redeclarable < TagDecl >
clang::TagDecl
clang::RecordDecl
clang::CXXRecordDecl
clang::ClassTemplateSpecializationDecl
clang::ClassTemplatePartial SpecializationDecl

RecursiveASTVisitor

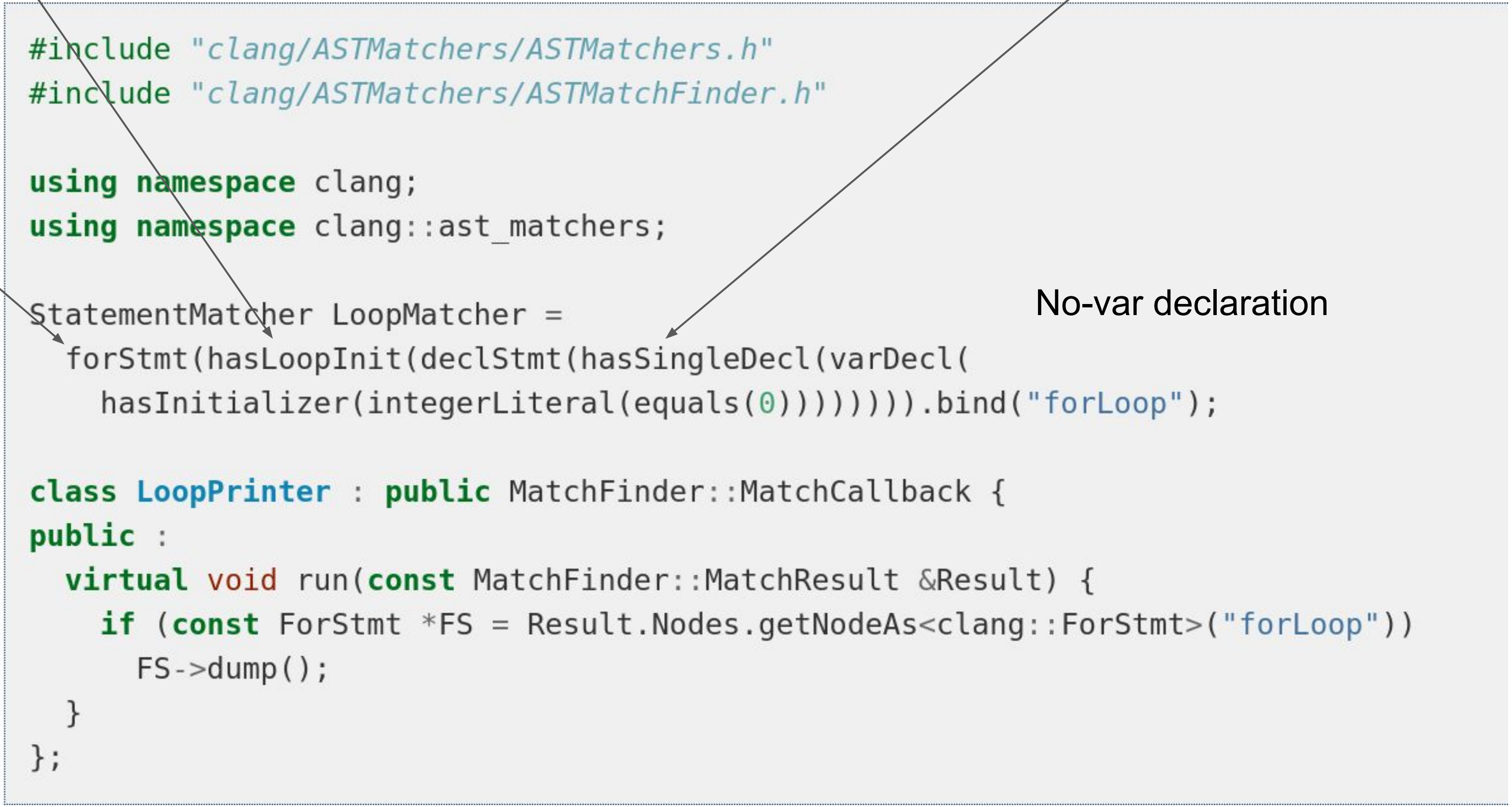


```
class FindNamedClassVisitor
  : public RecursiveASTVisitor<FindNamedClassVisitor> {
public:
  bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
    // For debugging, dumping the AST nodes will show which nodes are already
    // being visited.
    Declaration->dump();

    // The return value indicates whether we want the visitation to proceed.
    // Return false to stop the traversal of the AST.
    return true;
  }
};
```

AST Matchers

Loop initialization

There is one and only one
declaration with zero int initialization



```
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"

using namespace clang;
using namespace clang::ast_matchers;

StatementMatcher LoopMatcher =
  forStmt(hasLoopInit(declStmt(hasSingleDecl(varDecl(
    hasInitializer(integerLiteral(equals(0)))))))).bind("forLoop");

class LoopPrinter : public MatchFinder::MatchCallback {
public :
  virtual void run(const MatchFinder::MatchResult &Result) {
    if (const ForStmt *FS = Result.Nodes.getNodeAs<clang::ForStmt>("forLoop"))
      FS->dump();
  }
};
```

For loop

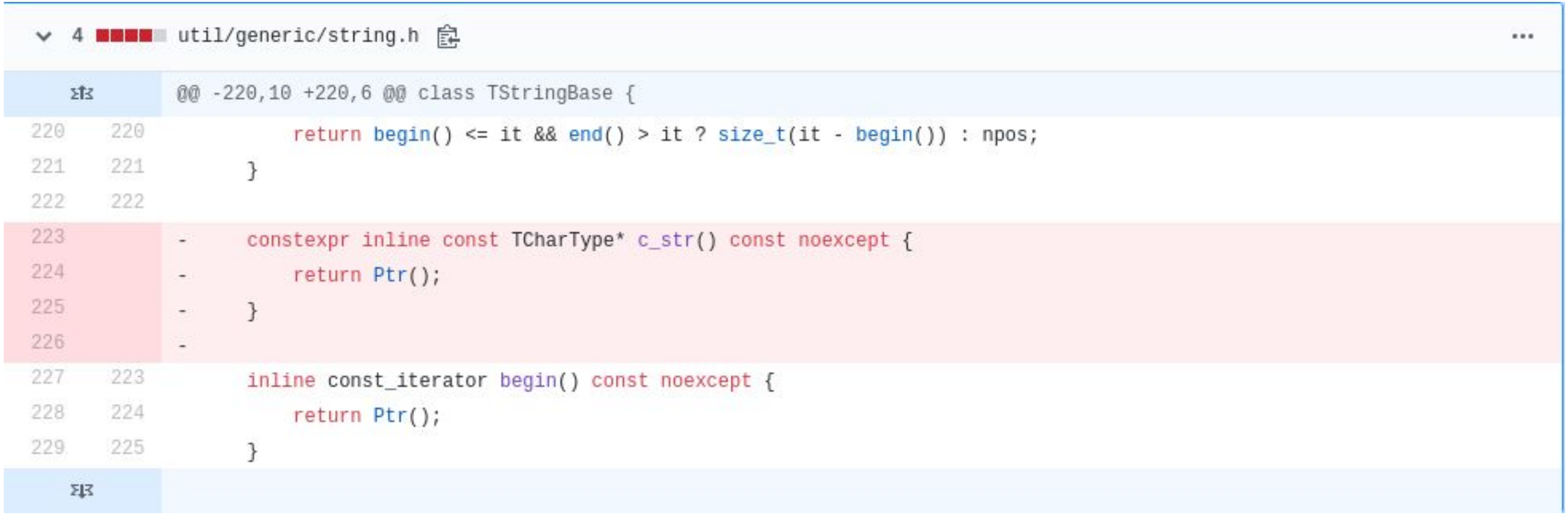
No-var declaration

AST Matchers

- https://clang.llvm.org/docs/LibASTMatchersReference.html -- full matchers reference. Check AST, iterate until working
- https://clang.llvm.org/docs/LibASTMatchersTutorial.html -- good example how to write matchers
- Look at clang-tidy, there are hundreds of matchers there, use them, learn, iterate
- Google, Yandex, Apple are using these techniques to provide better quality code a lot

AST Matchers

TStringBuf had c_str() method

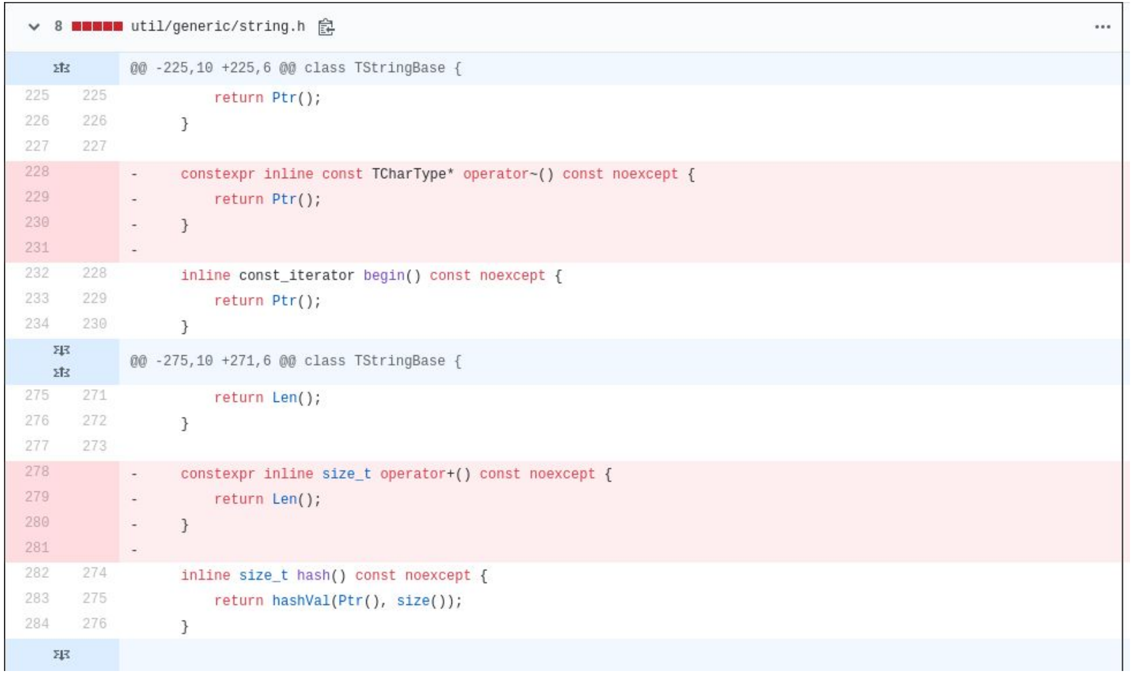


```
4 util/generic/string.h

@@ -220,10 +220,6 @@ class TStringBase {
          return begin() <= it && end() > it ? size_t(it - begin()) : npos;
      }

-     constexpr inline const TCharType* c_str() const noexcept {
-         return Ptr();
-     }
-
      inline const_iterator begin() const noexcept {
          return Ptr();
      }
```

Was only possible to find it with AST Matchers.
grep does not help.

AST Matchers

TString had unary operator~() and operator+() methods for .data() and .size() from **1997**



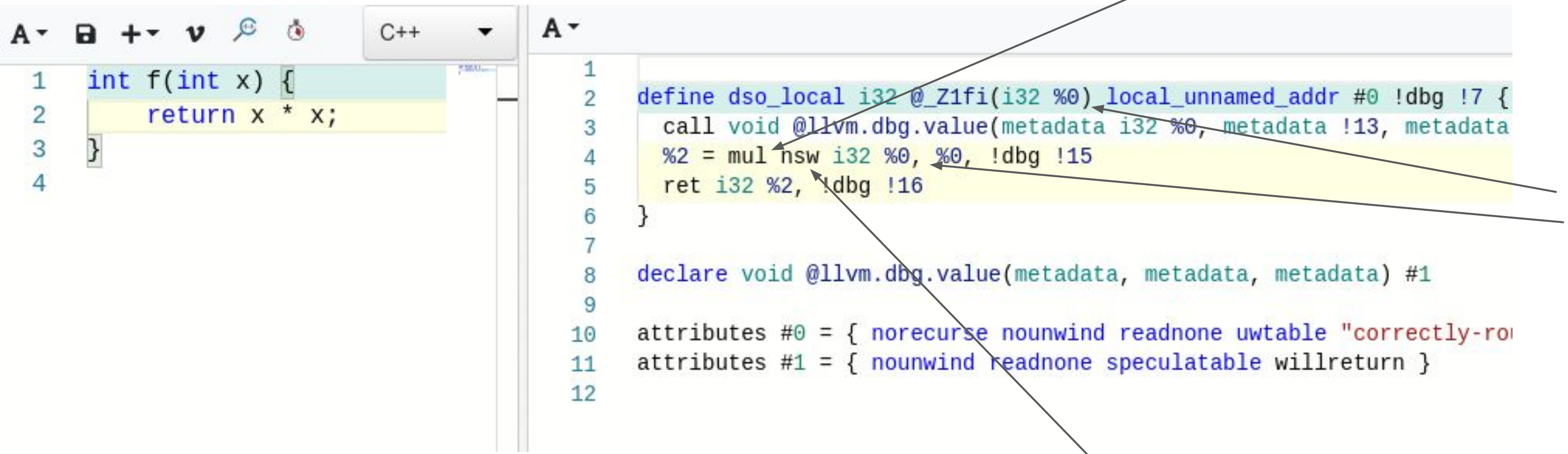
8 util/generic/string.h

```
@@ -225,10 +225,6 @@ class TStringBase {
            return Ptr();
        }

-       constexpr inline const TCharType* operator~() const noexcept {
-           return Ptr();
-       }
-
        inline const_iterator begin() const noexcept {
            return Ptr();
        }
@@ -275,10 +271,6 @@ class TStringBase {
            return Len();
        }

-       constexpr inline size_t operator+() const noexcept {
-           return Len();
-       }
-
        inline size_t hash() const noexcept {
            return hashVal(Ptr(), size());
        }
```

Was only possible to find it with AST Matchers. grep does not help AT ALL. 100000+ usages were removed. 2 days.

LLVM IR. Middle-End

- C++ is fast because compilers are doing a great job
- LLVM has its own assembly. Abstract and well recognized among many.
- https://llvm.org/docs/LangRef.html, 1700 pages.



```cpp
int f(int x) {
    return x * x;
}
```

```llvm

define dso_local i32 @_Z1fi(i32 %0) local_unnamed_addr #0 !dbg !7 {
  call void @llvm.dbg.value(metadata i32 %0, metadata !13, metadata
  %2 = mul nsw i32 %0, %0, !dbg !15
  ret i32 %2, !dbg !16
}

declare void @llvm.dbg.value(metadata, metadata, metadata) #1

attributes #0 = { norecurse nounwind readnone uwtable "correctly-ro
attributes #1 = { nounwind readnone speculatable willreturn }
```

Multiplication

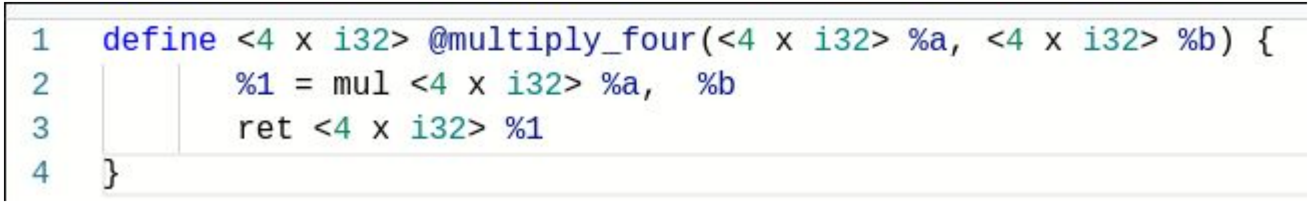
Vars

No signed wrap (otherwise UB in C++)

https://gcc.godbolt.org/z/CDHbKi

LLVM IR. Middle-End

- Types. Integers any width. i1, i31, i32, i64, i128. Vectors like <4 x i32> (SIMD). Structs: either an integer or float, either consists of structs
- Static Single Assignment



```
define <4 x i32> @multiply_four(<4 x i32> %a, <4 x i32> %b) {
        %1 = mul <4 x i32> %a,  %b
        ret <4 x i32> %1
}
```

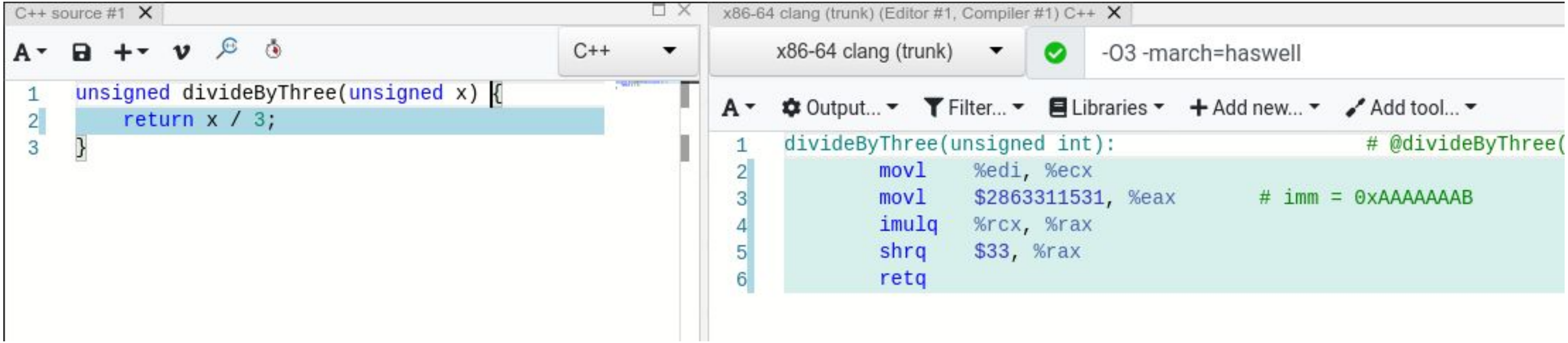
LLVM IR. Middle-End

- LLVM passes https://llvm.org/docs/Passes.html
- Transformation to equivalent IR but probably faster
  - Constant folding https://godbolt.org/z/pc4wQf
  - Constant propagation. https://godbolt.org/z/aGagaE
  - Common subexpression elimination. https://godbolt.org/z/fBFeQj
  - Dead code removal. https://godbolt.org/z/fDTwiK
  - Peephole optimizations. https://gcc.godbolt.org/z/aWTT9U
  - Vectorizers, unrollings, SIMD accelerations, inliner, tail calls, etc.
- ThinLTO and LTO. Stores IR and uses the cross target optimization passes
  - Devirtualization for the only one implementation classes

LLVM IR. Middle-End. Tuning. Use with caution

- **-O3,** -Ofast not recommended
- **-flto=thin**. +10%, Chromium, Google|Yandex. Compilation time tradeoff.
- **-mllvm -inline-threshold=1000.** Compilation time tradeoff
- x86-64:
  - 128 bit x86-64 options: **-mssse3, -msse4.1, -msse4.2, -mcx16**, **-maes**, **-mpclmul** (Galoi field extension) https://software.intel.com/sites/landingpage/IntrinsicsGuide/
  - 256 bit x86-64. -mavx, -mavx2, -mfma, -mxop
  - Bit manipulation. -mbmi2, **-mpopcnt**
  - **-mprefer-vector-width=128,** clang is still bad with 256 bit unless proven
- PowerPC:
  - **-maltivec, -mvsx**
  - LLVM has Intel intrinsics port to PowerPC equivalents
- ARM:
  - **-march=armv8.2-a+fp16+dotprod+simd**
  - Yandex has Intel intrinsics port to ARM by lecturer :-)

LLVM IR. Middle-End



C++

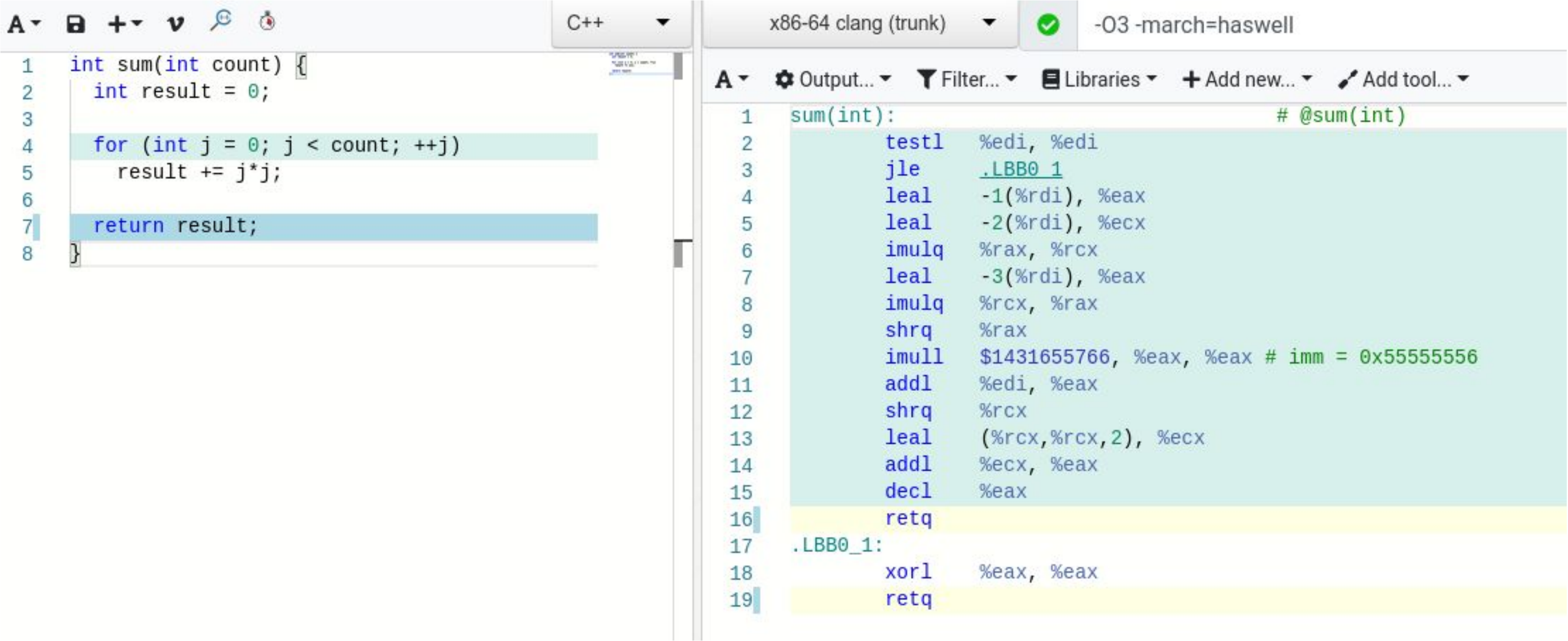
```cpp
unsigned divideByThree(unsigned x) {
    return x / 3;
}
```

x86-64 clang (trunk)

-O3 -march=haswell

```asm
divideByThree(unsigned int):                            # @divideByThree(
        movl    %edi, %ecx
        movl    $2863311531, %eax         # imm = 0xAAAAAAAB
        imulq   %rcx, %rax
        shrq    $33, %rax
        retq
```

LLVM IR. Middle-End



C++

x86-64 clang (trunk)

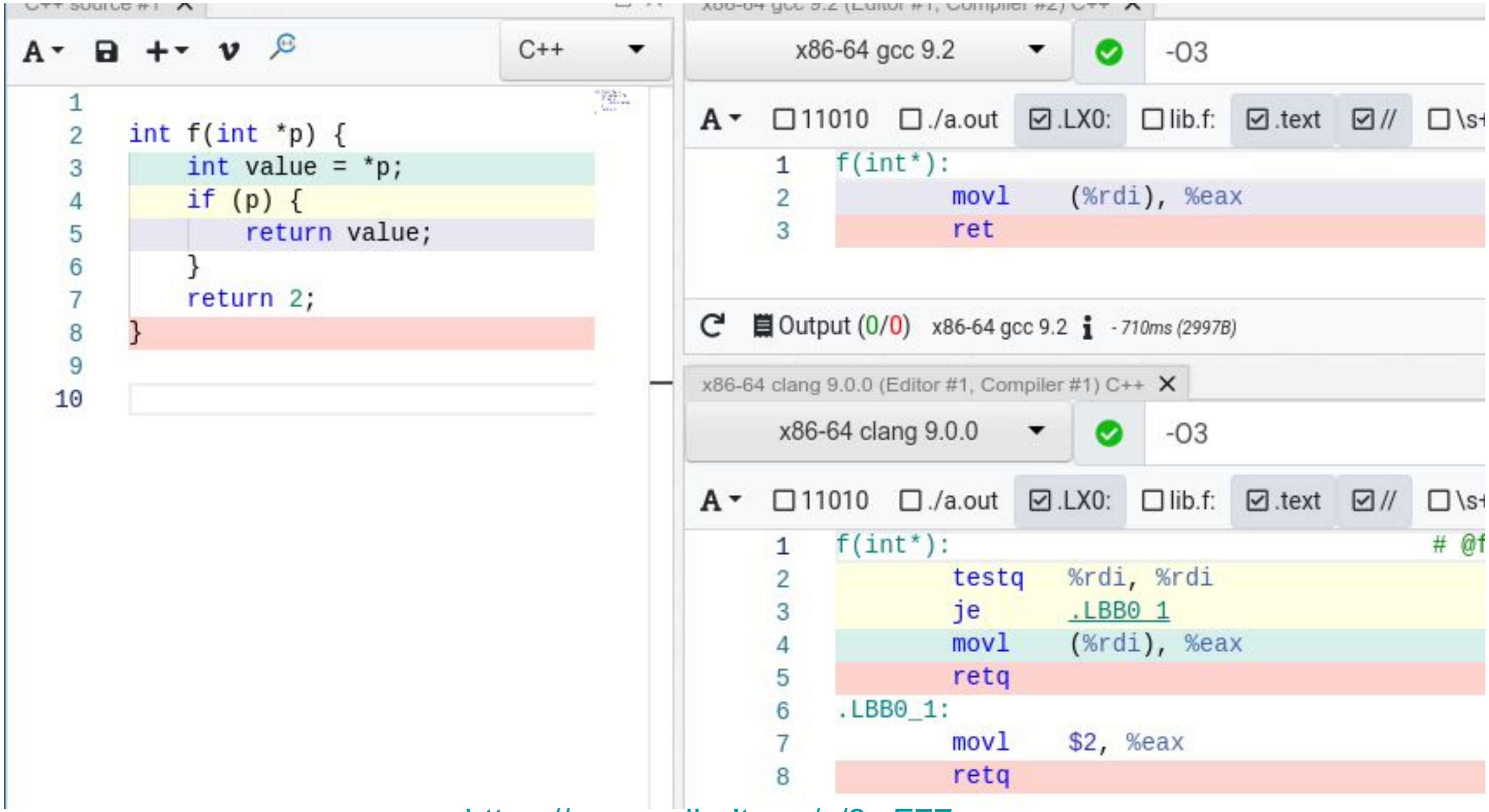
-O3 -march=haswell

```
int sum(int count) {
  int result = 0;

  for (int j = 0; j < count; ++j)
    result += j*j;

  return result;
}
```

```
sum(int):                               # @sum(int)
        testl   %edi, %edi
        jle     .LBB0_1
        leal    -1(%rdi), %eax
        leal    -2(%rdi), %ecx
        imulq   %rax, %rcx
        leal    -3(%rdi), %eax
        imulq   %rcx, %rax
        shrq    %rax
        imull   $1431655766, %eax, %eax # imm = 0x55555556
        addl    %edi, %eax
        shrq    %rcx
        leal    (%rcx,%rcx,2), %ecx
        addl    %ecx, %eax
        decl    %eax
        retq
.LBB0_1:
        xorl    %eax, %eax
        retq
```

https://kristerw.blogspot.com/2019/04/how-llvm-optimizes-geometric-sums.html
https://bohr.wlu.ca/ezima/papers/ISSAC94_p242-bachmann.pdf

LLVM IR. Middle-End


C++ source #1
A ▾
+ ▾
C++ ▾
x86-64 gcc 9.2 (Editor #1, Compiler #2) C++
A ▾ □ 11010 □ ./a.out ☑ .LX0: □ lib.f: ☑ .text ☑ // □ \s
Output (0/0) x86-64 gcc 9.2 - 710ms (2997B)
x86-64 clang 9.0.0 (Editor #1, Compiler #1) C++
A ▾ □ 11010 □ ./a.out ☑ .LX0: □ lib.f: ☑ .text ☑ // □ \s

```cpp
int f(int *p) {
    int value = *p;
    if (p) {
        return value;
    }
    return 2;
}
```

x86-64 gcc 9.2 -O3

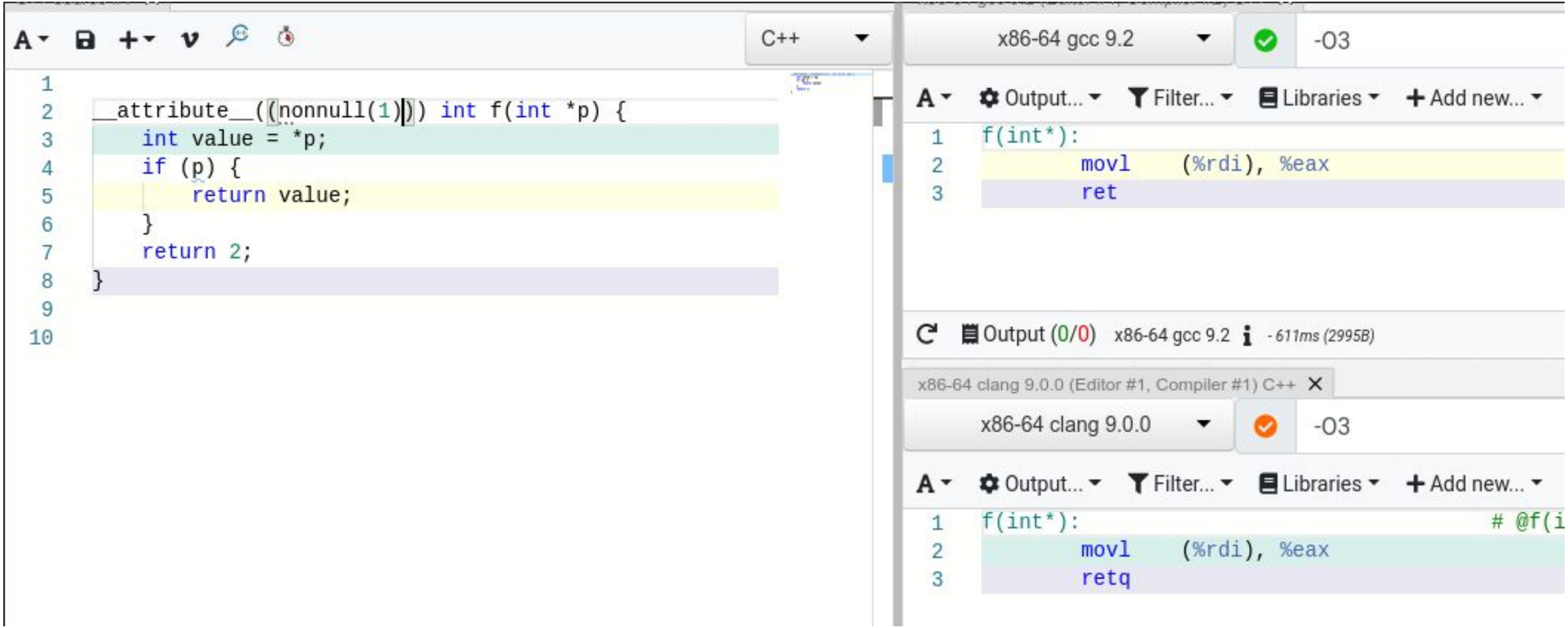
```
f(int*):
        movl    (%rdi), %eax
        ret
```

x86-64 clang 9.0.0 -O3

```
f(int*):                                # @f
        testq   %rdi, %rdi
        je      .LBB0_1
        movl    (%rdi), %eax
        retq
.LBB0_1:
        movl    $2, %eax
        retq
```

https://gcc.godbolt.org/z/9wF77m

LLVM IR. Middle-End



```cpp

__attribute__((nonnull(1))) int f(int *p) {
    int value = *p;
    if (p) {
        return value;
    }
    return 2;
}


```

x86-64 gcc 9.2, -O3:

```asm
f(int*):
        movl    (%rdi), %eax
        ret
```

x86-64 clang 9.0.0, -O3:

```asm
f(int*):                                # @f(i
        movl    (%rdi), %eax
        retq
```

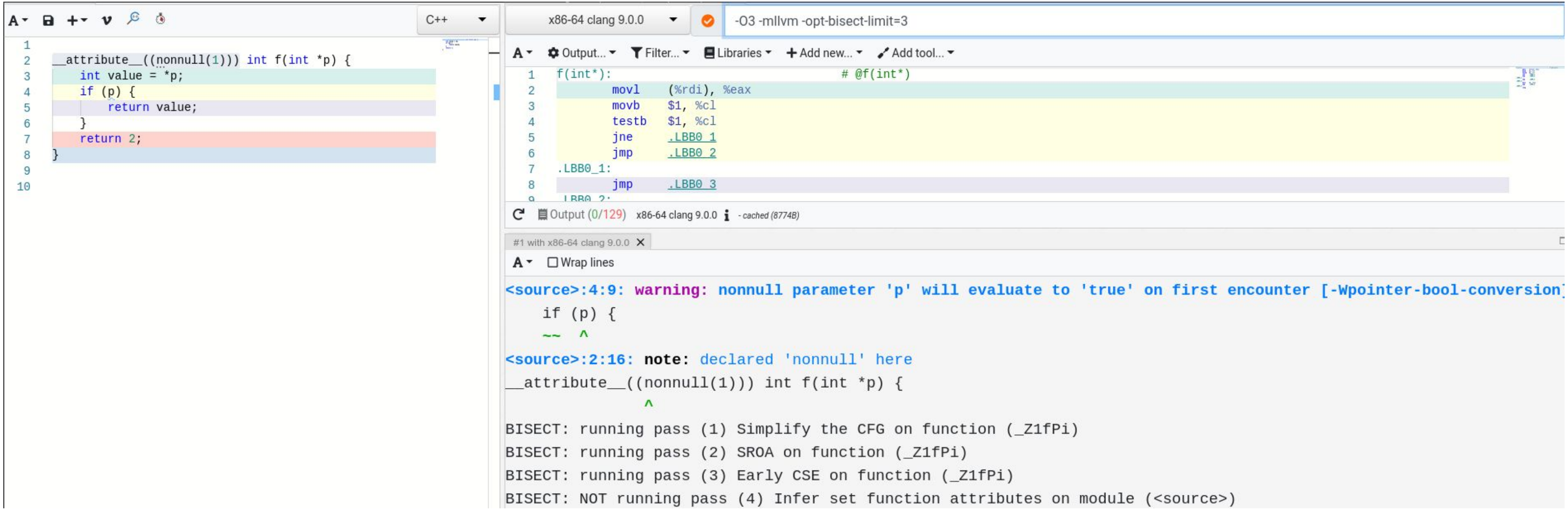
https://gcc.godbolt.org/z/VFfjpQ

LLVM IR. Middle-End

-mllvm -opt-bisect-limit=num

num was equal to 3

LLVM IR. Middle-End



x86-64 clang 9.0.0

-O3 -mllvm -opt-bisect-limit=3

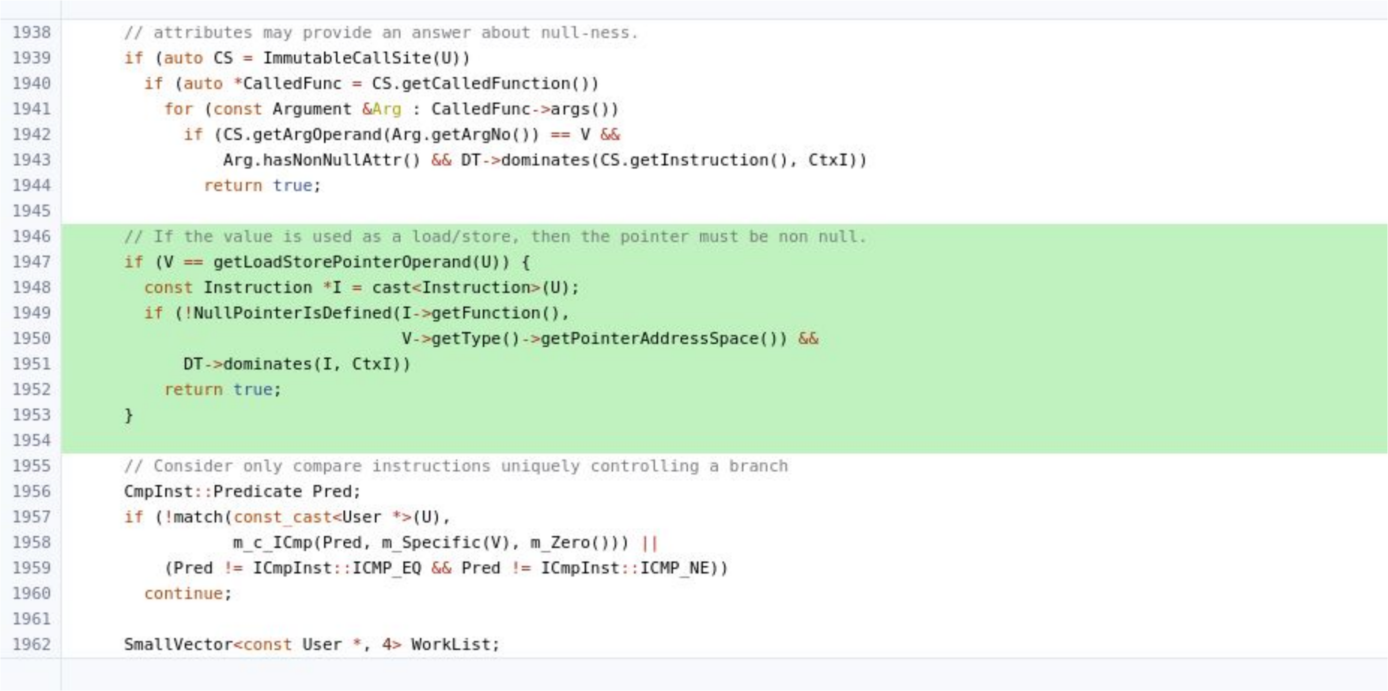
```
__attribute__((nonnull(1))) int f(int *p) {
    int value = *p;
    if (p) {
        return value;
    }
    return 2;
}
```

```
f(int*):                                # @f(int*)
        movl    (%rdi), %eax
        movb    $1, %cl
        testb   $1, %cl
        jne     .LBB0_1
        jmp     .LBB0_2
.LBB0_1:
        jmp     .LBB0_3
.LBB0_2:
```

```
<source>:4:9: warning: nonnull parameter 'p' will evaluate to 'true' on first encounter [-Wpointer-bool-conversion]
    if (p) {
    ~~  ^
<source>:2:16: note: declared 'nonnull' here
__attribute__((nonnull(1))) int f(int *p) {
               ^
BISECT: running pass (1) Simplify the CFG on function (_Z1fPi)
BISECT: running pass (2) SROA on function (_Z1fPi)
BISECT: running pass (3) Early CSE on function (_Z1fPi)
BISECT: NOT running pass (4) Infer set function attributes on module (<source>)
```

https://gcc.godbolt.org/z/RbKFWW

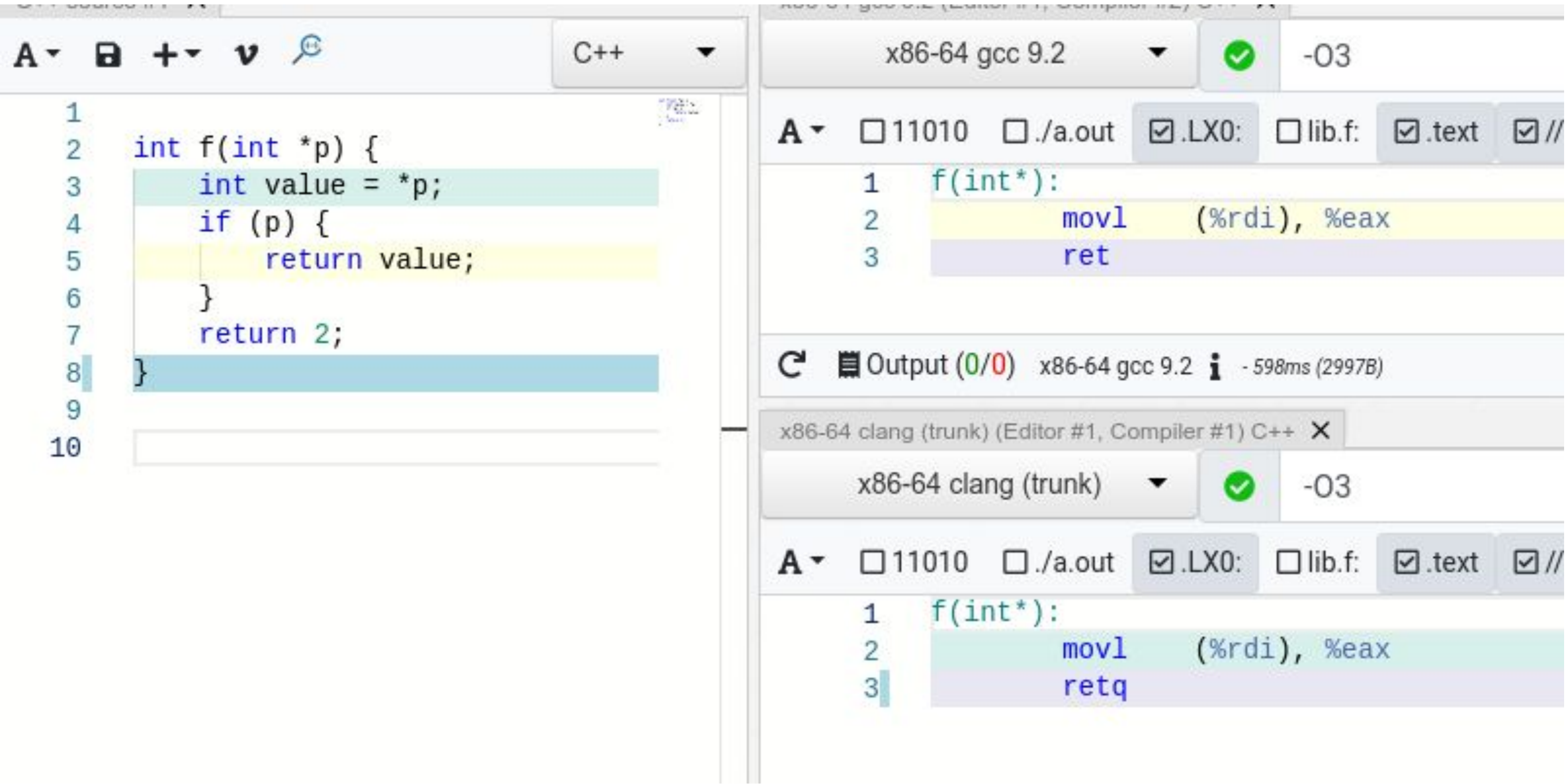
LLVM IR. Middle-End



```
// attributes may provide an answer about null-ness.
if (auto CS = ImmutableCallSite(U))
  if (auto *CalledFunc = CS.getCalledFunction())
    for (const Argument &Arg : CalledFunc->args())
      if (CS.getArgOperand(Arg.getArgNo()) == V &&
          Arg.hasNonNullAttr() && DT->dominates(CS.getInstruction(), CtxI))
        return true;

// If the value is used as a load/store, then the pointer must be non null.
if (V == getLoadStorePointerOperand(U)) {
  const Instruction *I = cast<Instruction>(U);
  if (!NullPointerIsDefined(I->getFunction(),
                            V->getType()->getPointerAddressSpace()) &&
      DT->dominates(I, CtxI))
    return true;
}

// Consider only compare instructions uniquely controlling a branch
CmpInst::Predicate Pred;
if (!match(const_cast<User *>(U),
           m_c_ICmp(Pred, m_Specific(V), m_Zero())) ||
    (Pred != ICmpInst::ICMP_EQ && Pred != ICmpInst::ICMP_NE))
  continue;

SmallVector<const User *, 4> WorkList;
```

https://reviews.llvm.org/D71177

LLVM IR. Middle-End



C++

```cpp

int f(int *p) {
    int value = *p;
    if (p) {
        return value;
    }
    return 2;
}


```

x86-64 gcc 9.2 -O3

☐ 11010 ☐ ./a.out ☑ .LX0: ☐ lib.f: ☑ .text ☑ //

```
f(int*):
        movl    (%rdi), %eax
        ret
```

Output (0/0) x86-64 gcc 9.2 - 598ms (2997B)

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

x86-64 clang (trunk) -O3

☐ 11010 ☐ ./a.out ☑ .LX0: ☐ lib.f: ☑ .text ☑ //

```
f(int*):
        movl    (%rdi), %eax
        retq
```

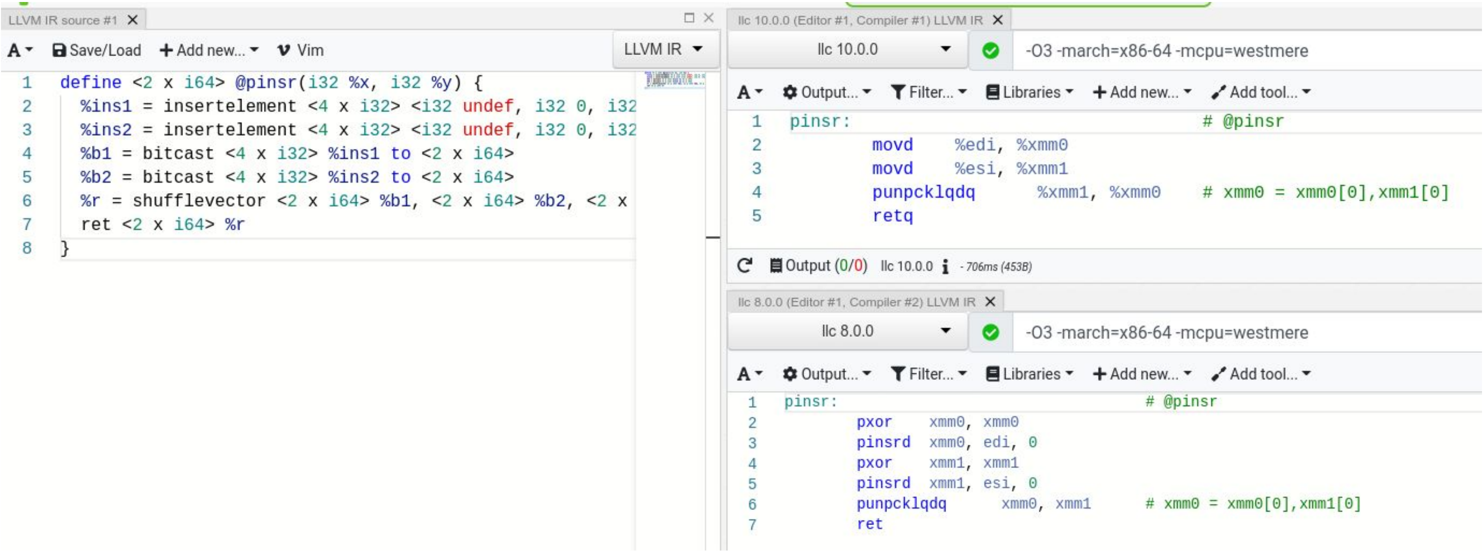
Clang 10!

LLVM IR. Middle-End

https://telegra.ph/Kak-propatchit-LLVM-za-odin-den-s-vidimym-performansom-12-14 (in Russian)

Backend

- Lots of processor semantics heuristics
- https://llvm.org/docs/CommandGuide/llvm-mca.html -- overall trying to optimize the CPU clock execution.



```
define <2 x i64> @pinsr(i32 %x, i32 %y) {
  %ins1 = insertelement <4 x i32> <i32 undef, i32 0, i32
  %ins2 = insertelement <4 x i32> <i32 undef, i32 0, i32
  %b1 = bitcast <4 x i32> %ins1 to <2 x i64>
  %b2 = bitcast <4 x i32> %ins2 to <2 x i64>
  %r = shufflevector <2 x i64> %b1, <2 x i64> %b2, <2 x
  ret <2 x i64> %r
}
```

llc 10.0.0 -O3 -march=x86-64 -mcpu=westmere

```
pinsr:                                  # @pinsr
        movd    %edi, %xmm0
        movd    %esi, %xmm1
        punpcklqdq      %xmm1, %xmm0    # xmm0 = xmm0[0],xmm1[0]
        retq
```

Output (0/0) llc 10.0.0 - 706ms (453B)

llc 8.0.0 -O3 -march=x86-64 -mcpu=westmere

```
pinsr:                                  # @pinsr
        pxor    xmm0, xmm0
        pinsrd  xmm0, edi, 0
        pxor    xmm1, xmm1
        pinsrd  xmm1, esi, 0
        punpcklqdq      xmm0, xmm1      # xmm0 = xmm0[0],xmm1[0]
        ret
```

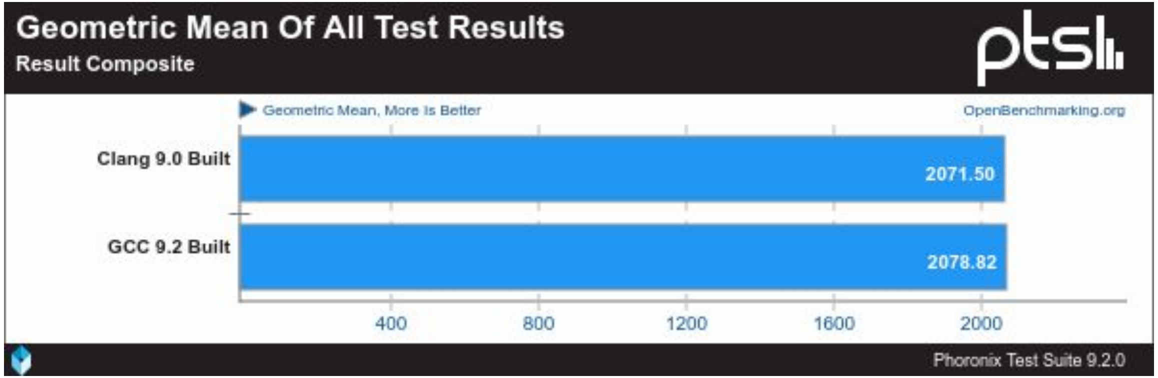
https://reviews.llvm.org/D60852

Tools

- ASAN (all non allocated region access, double free), HWASAN (hardware assistant)
- TSAN (data races)
- MSAN (use of uninitialized memory)
- UBSAN (undefined behavior, bad casts, math overflow, etc)
- Kernel versions of sanitizers
- clang-tidy -- linter with matchers
- IWYU (include what you use)
- lldb -- like gdb but for LLVM

Future

- Linux under clang by default. Already can be built and shows 0.3% loss to GCC. Because of tooling, clang found thousands of bugs.



Future

- Libc (in progress)
- Its own kernel(?)
- Beat GCC performance in majority of test cases
- Better error handling
- More languages. Flang was added in llvm-10
- IR machine learning techniques. NIPS paper

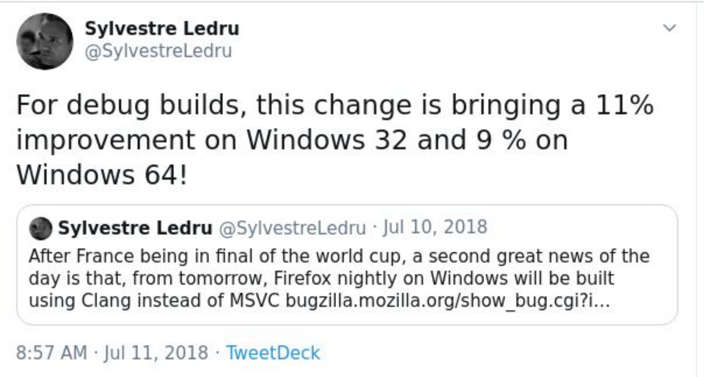
GCC vs Clang

-5% up to 5%, with tuning I saw always better results

https://www.phoronix.com/scan.php?page=news_item&px=GCC-LLVM-Clang-Icelake-Tests

Future

- Windows MSVC competition. Game industry and browsers start using clang

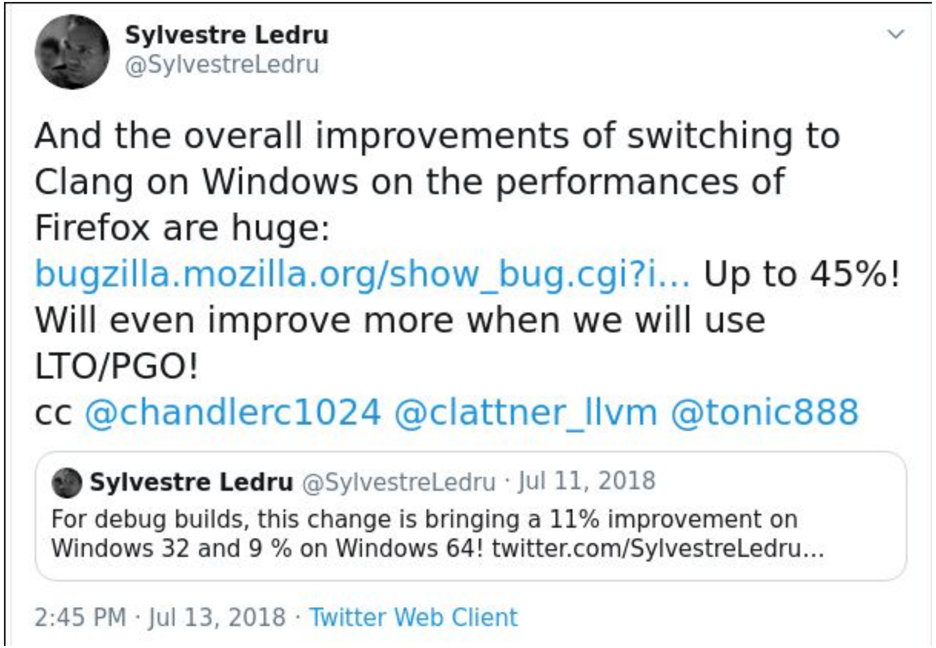


Sylvestre Ledru
@SylvestreLedru

For debug builds, this change is bringing a 11% improvement on Windows 32 and 9 % on Windows 64!

Sylvestre Ledru @SylvestreLedru · Jul 10, 2018
After France being in final of the world cup, a second great news of the day is that, from tomorrow, Firefox nightly on Windows will be built using Clang instead of MSVC bugzilla.mozilla.org/show_bug.cgi?i...

8:57 AM · Jul 11, 2018 · TweetDeck



Sylvestre Ledru
@SylvestreLedru

And the overall improvements of switching to Clang on Windows on the performances of Firefox are huge: bugzilla.mozilla.org/show_bug.cgi?i... Up to 45%! Will even improve more when we will use LTO/PGO!
cc @chandlerc1024 @clattner_llvm @tonic888

Sylvestre Ledru @SylvestreLedru · Jul 11, 2018
For debug builds, this change is bringing a 11% improvement on Windows 32 and 9 % on Windows 64! twitter.com/SylvestreLedru...

2:45 PM · Jul 13, 2018 · Twitter Web Client

https://twitter.com/SylvestreLedru/status/1017751994788917249

Impact

Contribute: https://llvm.org/OpenProjects.html