Use the Source

Dietmar Kühl
API Technology London
Bloomberg LP

Copyright

Overview

- what is clang?
- which interfaces are provided by clang?
- prototype doing some checks
- what else can and should be done

Clang

- open-source C, C++ compiler for llvm
- actively developed by Apple and Google
  - ... plus a growing open-source community
- implemented using C++
  - ... although with some restrictions

Clang Source

- http://clang.llvm.org
- tool living in the llvm source tree
- current version is 3.2

Clang Goals

- fast compiles and low memory use
- expressive diagnostics
- modular library based architecture
- support for diverse clients

Clang Interfaces

- lexer and preprocessor
- abstract syntax tree (AST)
- semantic information (Sema)
- control flow graph (CFG)

Creating a Plug-In

- basic entry point to live within clang
- requires creation of a shared object
- basic steps are
  - create a symbol to create the plug-in
  - override the suitable virtual functions

Basic Plug-In

```
class Plug : public PluginASTAction {
    ASTConsumer*
    CreateASTConsumer(CompilerInstance& ci,
                      llvm::StringRef file);

    bool ParseArgs(CompilerInstance const& ci,
                   std::vector<std::string> const& av){
        return true; // no arguments for now
    }
};
FrontendPluginRegistry::Add<Plug> r("nop", "");
```

Basic Consumer

```
typedef CompilerInstance CI;
struct Consumer: ASTConsumer {
    Consumer(CI& ci, std::string name) {}
};
clang::ASTConsumer*
Plugin::CreateASTConsumer(CI& ci,
                          llvm::StringRef file) {
    return new Consumer(ci, file);
}
```

Building a Plug-In

- requires a debug build of clang
- macros:
  -D__STDC_LIMIT_MACROS
  -D__STDC_CONSTANT_MACROS
- flags: -fno-exceptions -fno-rtti
- linker: -Wl,-undefined,dynamic_lookup

Running the Plug-In

- clang -cc1 -load nop.dylib \
  -plugin nop -plugin-arg-nop argument \
  file.cpp

- some compiler flags are readily supported, e.g., -I<dir> and -D<macro>

Report #define

- demonstrate preprocessor events
- introduce source locations
- show how to produce diagnostics

Preprocessor

- events for preprocessor actions:
  - changed the active file (start/end of include)
  - defined, undefined, expanded macro
  - #if, #ifdef, #ifndef, #elif, #else, #endif
- tracked separately from the source

PPCallbacks

```
struct PP: PPCallbacks
{
    PP(CompilerInstance& c): c_(c) {}
    void MacroDefined(Token const& token,
                      MacroInfo const* MI);
    CompilerInstance& c_;
};
```

Register PPCallbacks

```
ASTConsumer*
Plug::CreateASTConsumer(
                    CompilerInstance& c,
                    llvm::StringRef) {
    c.getPreprocessor().addPPCallbacks(
                                  new PP(c));
    return new Consumer();
}
```

Report a Macro

```
typedef DiagnosticsEngine DE;
void PP::MacroDefined(Token const& t,
                      MacroInfo const* MI) {
  SourceLocation w(MI->getDefinitionLoc());
  DE &de(this->c_.getDiagnostics());
  int id = de.getCustomDiagID(DE::Warning,
                              "macro-def: '%0'");
  DiagnosticBuilder(de.Report(w, id))
    << t.getIdentifierInfo()->getName();
}
```

Example Output

```
In file included from tst.cpp:1:
./tst.h:1:9:warning: macro-def: 'example'
#define example(x) !x
        ^
```

Diagnostics

- created from a registry to for translations
- it is possible to use custom diagnostics
- flexible format string
  - positional arguments for many types
  - support for enumerated values, plurals, etc.

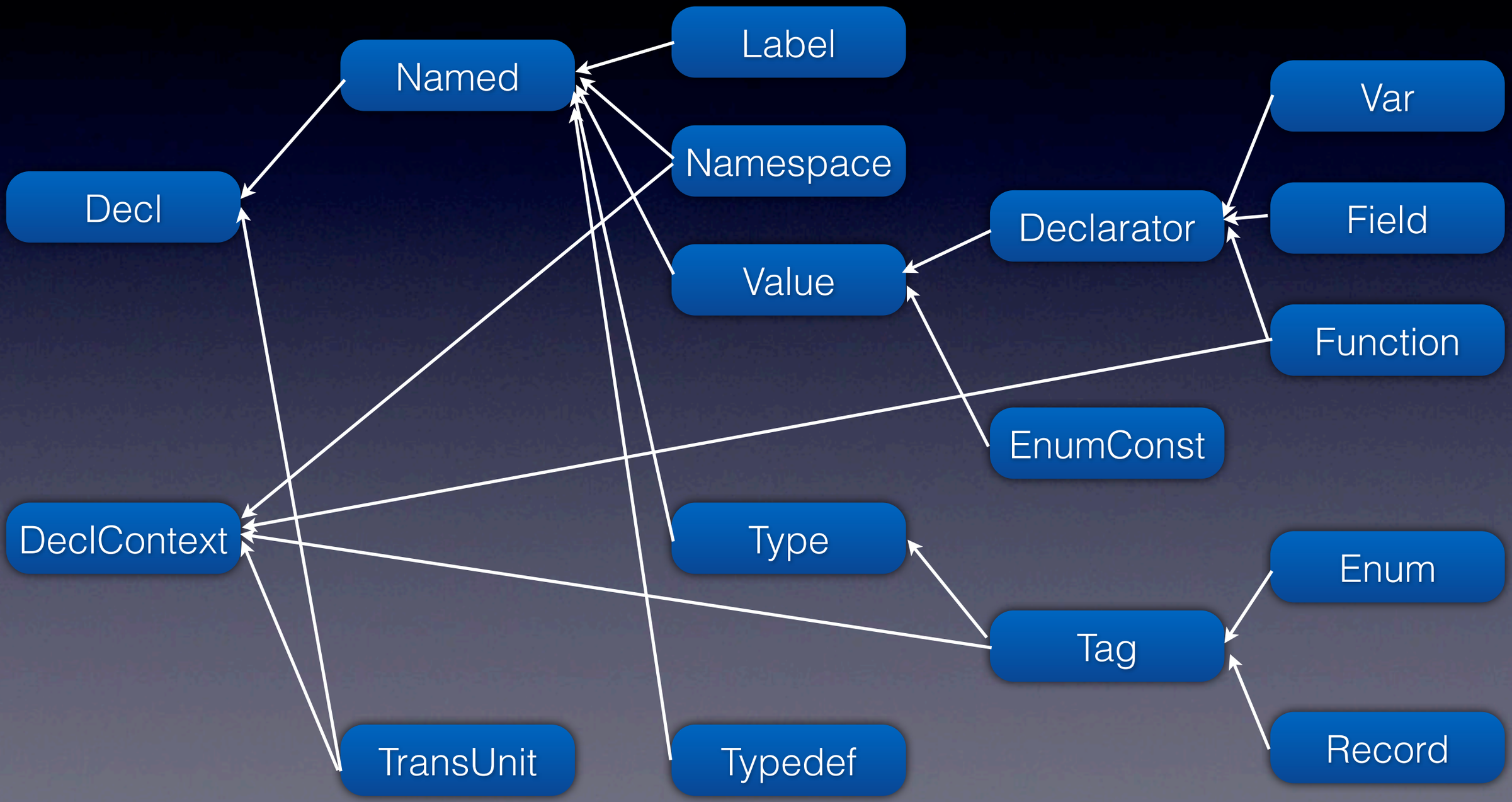
Abstract Syntax Tree

- represents all C++ constructs
- constructs can be navigated
- visitors can be built via includes for nodes
- hierarchies for declaration, expressions, statements, and types
- all entities specify their source location

Declaration

- top-level of the AST
  - contexts containing groups of declarations
- declarations reference prior declarations
- declarations may refer to
  - expressions e.g. initializers
  - statements e.g. for function definitions

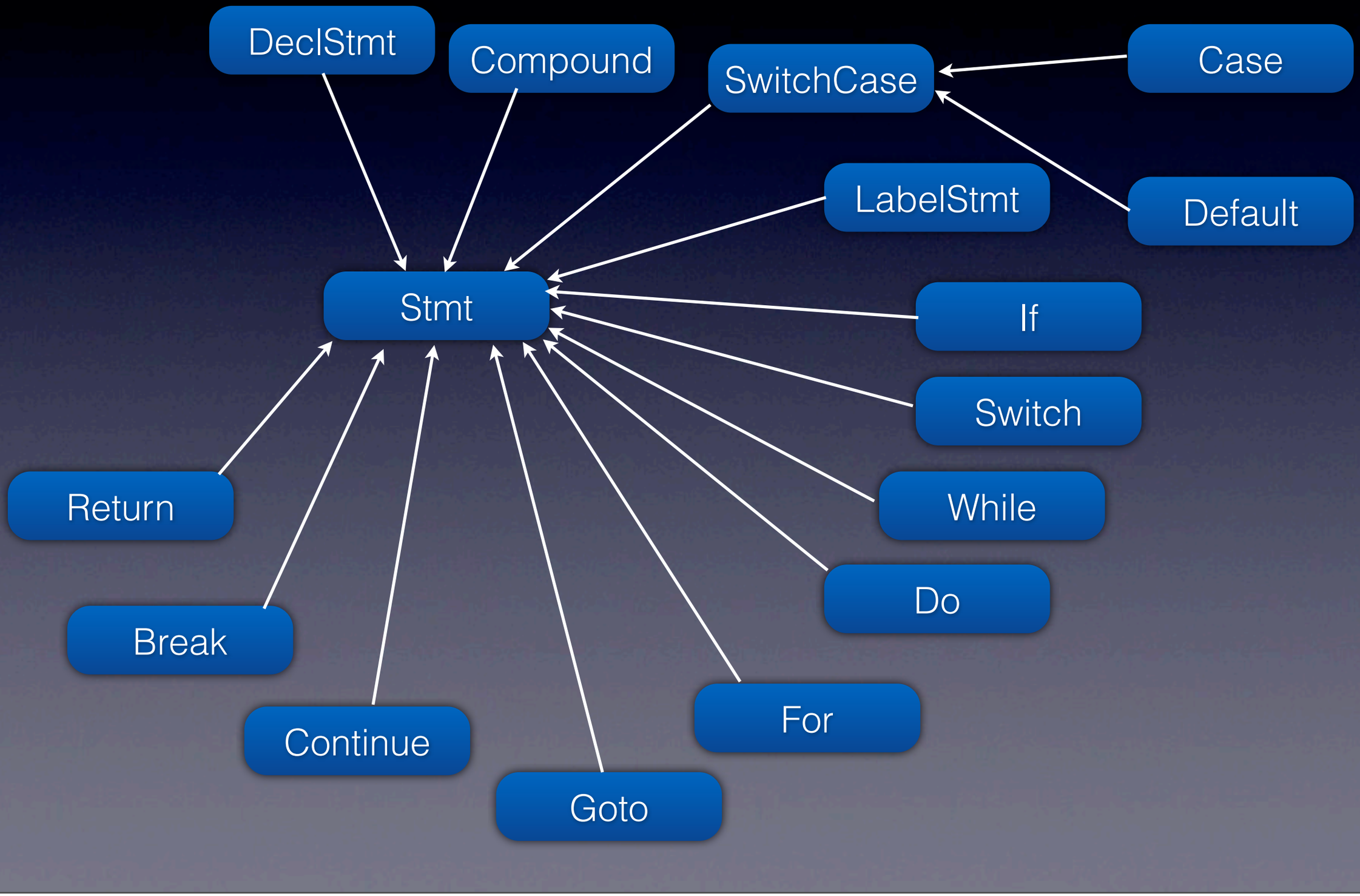
Declarations (part)



Statements

- represent the bodies of functions
- fine grained hierarchy of language constructs
- statements use various expressions

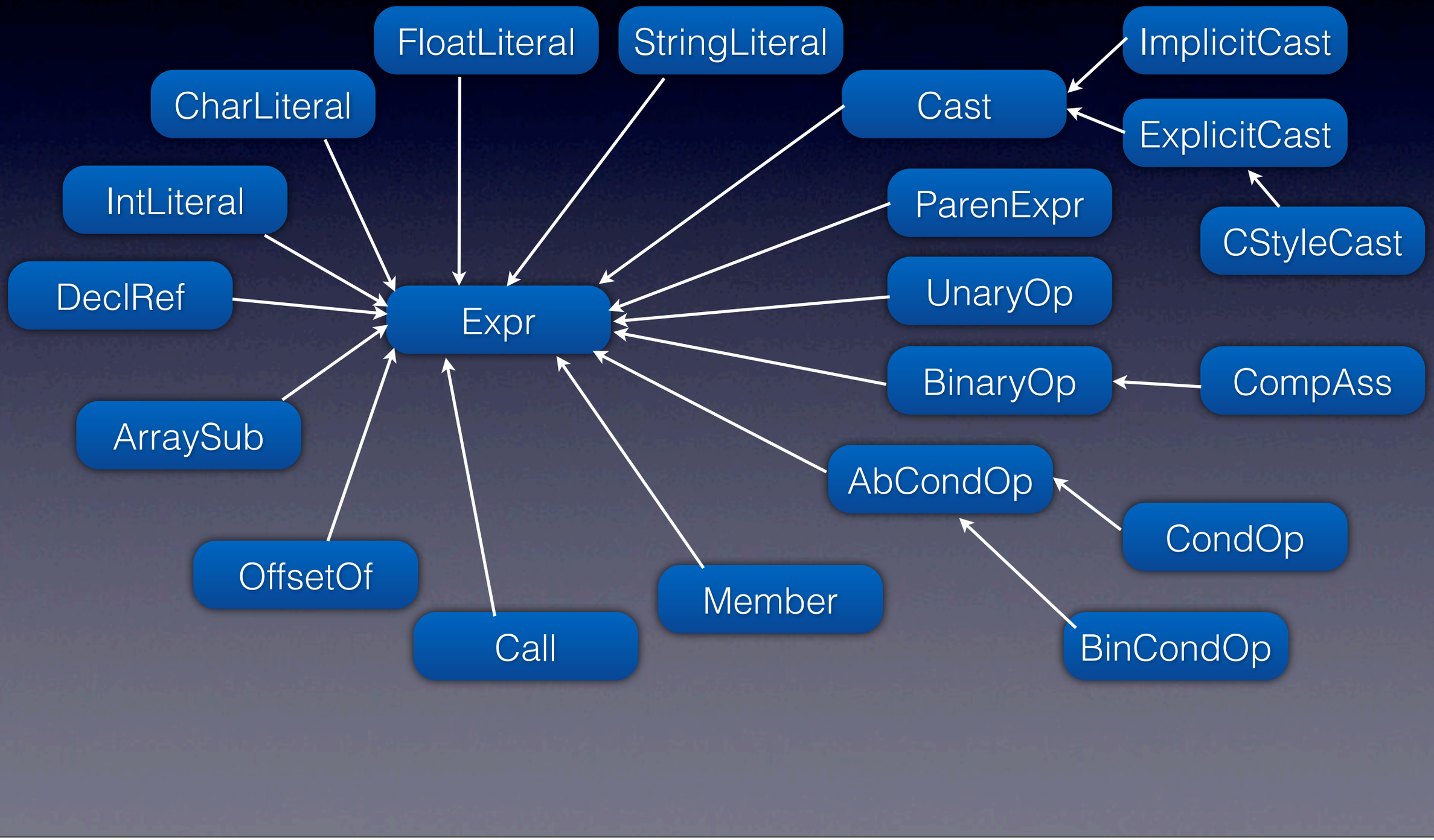
Statements (part)



Expression

- represent the actually evaluated contexts
- again use a fine grained hierarchy
- include some auxiliary wrappers, e.g. for conversions
- refers to all the types involved in expressions

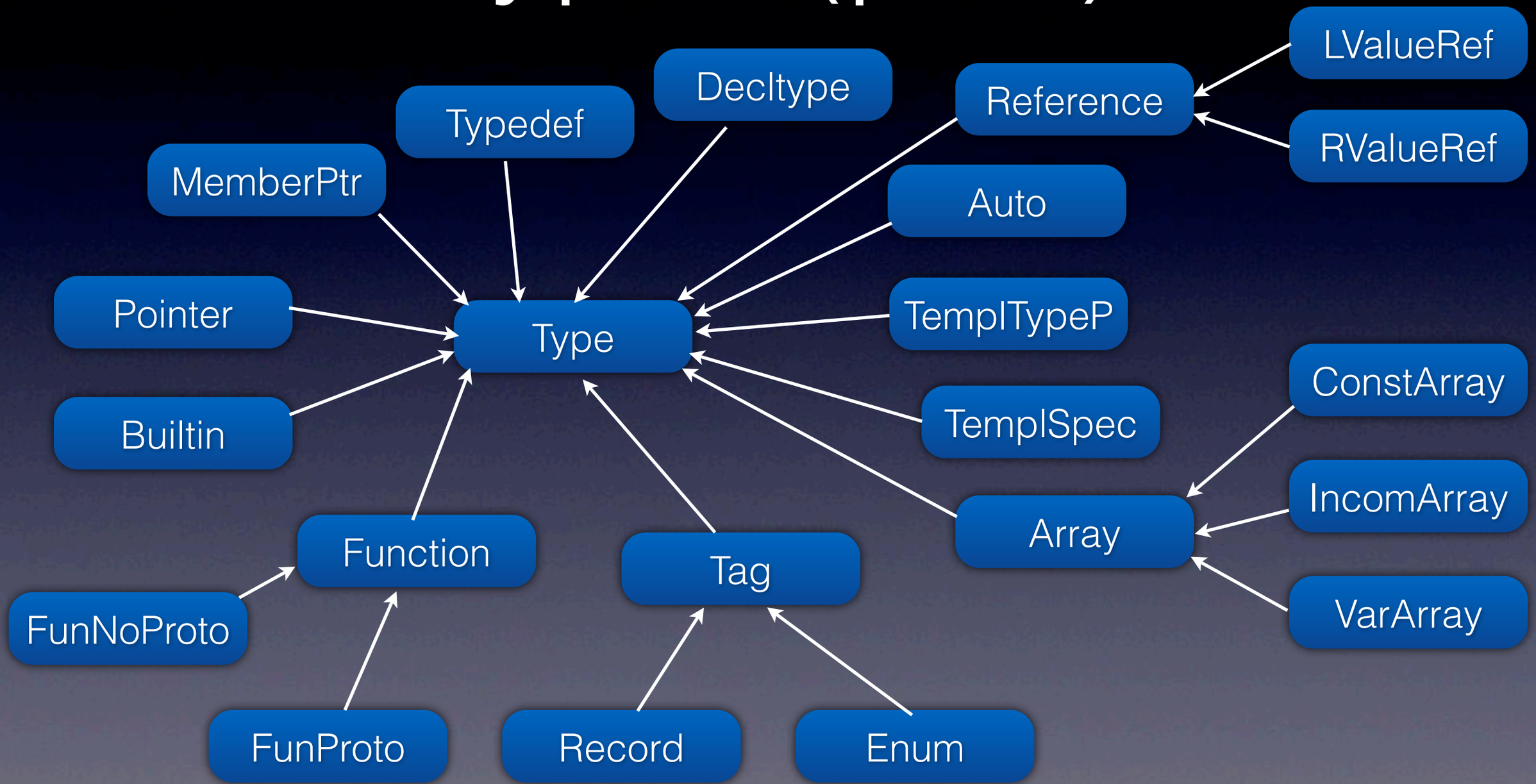
Expressions (part)



Types

- types encode details of their typedef
  - different types for different typedefs
    - allows reporting names used by users
    - canonical type is the underlying actual type
- qualifiers are not part of the types but are part of flyweight objects actually referenced

Types (part)


LValueRef
Decltype
Reference
Typedef
RValueRef
MemberPtr
Auto
Pointer
TemplTypeP
Type
ConstArray
Builtin
TemplSpec
IncomArray
Array
Function
Tag
FunNoProto
VarArray
FunProto
Record
Enum

QualType

- reference the actual type
- represents the different forms of types
  - const, volatile, restrict qualifiers
  - are different for typedef to retain these names!
    - to compare types use the *canonical* object

Example: Dodgy Casts

- for 32 bit systems int and long have same size
- for 64 bit system they do not
- objective: detect casts likely to be wrong:
  - int* p = new int();
  - (long*)p
  - reinterpret_cast<long*>(p)

AST Entry Point

```
void doDecl(CI* c, Decl* d) {
    Visitor(c).TraverseDecl(d);
}
```

```
bool Con::HandleTopLevelDecl(DeclGroupRef DG) {
    std::for_each(DG.begin(), DG.end(),
        std::bind1st(std::ptr_fun(&doDecl), &this->c_));
    return true;
}
```

Visiting the AST

```
struct Visitor: RecursiveASTVisitor<Visitor> {
    CI* c_;
    Visitor(CI* c): c_(c) {}

    bool VisitCStyleCastExpr(CStyleCastExpr* e);
    bool VisitCXXReinterpretCastExpr(
                            CXXReinterpretCastExpr* e);
    bool checkCast(CastExpr const* e);
};
```

Check Expression

```
bool Visitor::checkCast(CastExpr const* e) {
    if (e->getCastKind() == CK_BitCast
        && e->getType()->isPointerType()
        && e->getSubExpr()->getType()->isPointerType()
        && getCanon(e)->isIntegerType()
        && getCanon(e) != getCanon(e->getSubExpr()))
    {
        ...
    }
}
```

Extract Interesting Type

```
QualType getCanon(Expr const* e)
{
    return e->getType()
        ->getPointeeType()
        .getCanonicalType()
        .getUnqualifiedType();
}
```

Check for the Types

```
QualType to = getCanon(e);
QualType from = getCanon(e->getSubExpr());
typedef BuiltinType BT;
BT const* bt0 = dyn_cast<BT>(to.getTypePtr());
BT const* bt1 = dyn_cast<BT>(from.getTypePtr());
if (bt0 && bt1
    && ((bt0->getKind() == BuiltinType::Long
           && bt1->getKind() == BuiltinType::Int)
         || (bt0->getKind() == BuiltinType::ULong
             && bt1->getKind() == BuiltinType::UInt))) {
```

The Report

```
SourceLocation w(e->getExprLoc());
DE &de(this->c_->getDiagnostics());
int id = de.getCustomDiagID(DE::Warning,
      "dodgy cast from %0 to %1");
DiagnosticBuilder(de.Report(w, id))
   << e->getSubExpr()->getType().getAsString()
   << e->getType().getAsString();
```

Test Example

```
template <typename T> void use(T) {}
int main() {
    typedef int  I;
    int i0 = 17;
    typedef long L;
    use(reinterpret_cast<I const*>(&i0));
    use(reinterpret_cast<L const*>(&i0));
    use(reinterpret_cast<long*>(&i0));
}
```

Test Output

```
tst.cpp:6:9:warning: dodgy cast from int * to const L *
    use(reinterpret_cast<L const*>(&i0));
            ^
tst.cpp:7:9:warning: dodgy cast from int * to long *
    use(reinterpret_cast<long*>(&i0));
            ^
2 warnings generated.
```

Sema

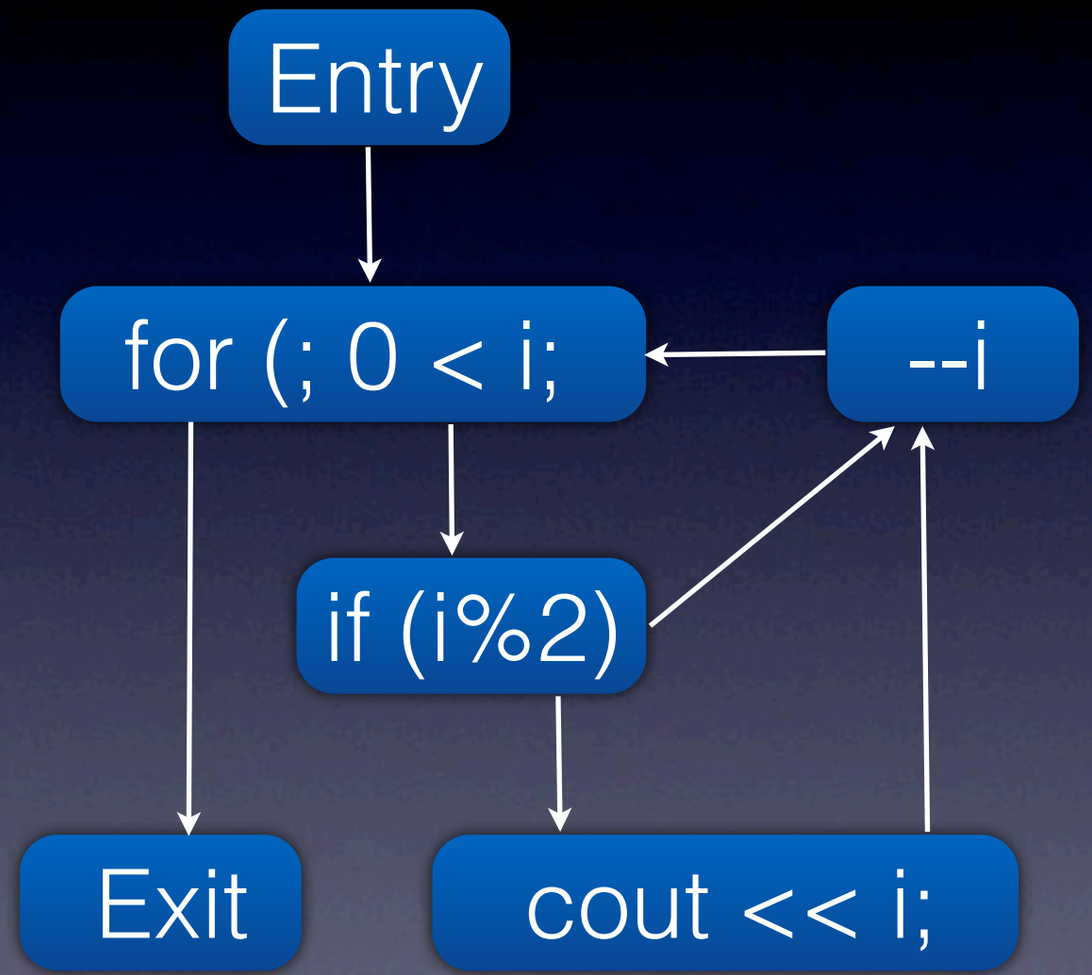
- semantic information beyond the AST
- somewhat of the kitchen sink
- provides lots of functions for C++ rules

Control Flow Graph

- can be obtained for any statement
- gives a graph modelling control flow
  - basic block level (unconditional sequences)
  - consists of CFGBlocks referencing Stmts

CFG Example

```
void f(int i) {
    for (; 0 < i; --i) {
        if (i % 2) {
            cout << i;
        }
    }
}
```



Cyclomatic Complexity

- measures the number of conditionals:
  - computes a graph property
  - #edges - #nodes + 2 * #components
- cyclomatic complexity correlated to bugs?

Visitor for CFG

```
struct Visitor: RecursiveASTVisitor<Visitor> {
    DE & de_; int id_;
    Visitor(CI* c):
        de_(c->getDiagnostics()),
        id_(this->de_.getCustomDiagID(DE::Warning,
                "cyclomatic complexity: %0")) {}

    bool VisitFunctionDecl(FunctionDecl* d);
};
```

Get CFG Build

```
bool Visitor::VisitFunctionDecl(FunctionDecl* d) {
    if (d->doesThisDeclarationHaveABody()) {
        std::auto_ptr<CFG> cfg(
            CFG::buildCFG(d,
                          d->getBody(),
                          &d->getASTContext(),
                          CFG::BuildOptions()));
...
```

Traverse the CFG

```
// cfg->dump(LangOptions(), false);
int edges(0);
for (CFG::iterator it(cfg->begin()),
          end(cfg->end()); it != end; ++it) {
    edges += (*it)->succ_size();
}
...
```

Print the CFG Result

```
if (10 < edges - cfg->size() + 2) {
    SourceLocation w(d->getLocStart());
    DiagnosticBuilder(
        this->de_.Report(w, this->id_))
            << (edges - cfg->size() + 2);
}
```

Clang Plug-ins

- all this is available to clang plug-ins
- available when C++ source analysis needed
- name completion
- refactoring tools
- analysis of source for various uses
- documentation frameworks

Static Analysis

- checking for coding guidelines
- detect common errors
  - existing tools check for generic errors
  - company specific errors are not detected
- compute code metrics of various kinds

Prototype Analyser

- check for certain coding guidelines
- detect certain specific errors
- detect code use which may be questionable
- should be publicly available soon...

Guideline Checks

- use of anonymous namespace in header
- component header not included first
- global type or function not declared in header
- same arguments names in redeclarations
- member defined in class definition

Code Generation

- basis to generate type-information
- output default implementations
- input/output operators
- comparison operators
- swap() functions

C++2011 to C++2003

- on some platforms C++2011 isn't supported
- the compiler can't be changed
- C++2011 has nice features
- ⇒ convert some C++2011 into C++2003