Clang-tidy Code Dojo

Piotr Padlewski i Marek Sokołowski

Warsaw C++ Users Group 07.06.2016

IIIT

Drużyna w składzie

- Krystyna Gajczyk
- Stanisław Barzowski
- Piotr Padlewski
- Jakub Staroń
- Marek Sokołowski

IIIT

Zajmujemy się rozwijaniem nowych checków do clang-tidy.

W przyszłości cięższą analizą statyczną?

Clang-tidy

Clang-tidy to coś w rodzaju lintera opartego na clangu. Jednak spośród innych linterów dostępnych na rynku wyróżnia go:

- banalnie proste pisanie nowych checków,
- transformowanie kodu.

Clang-tidy-3.8 posiada ~80 checków.

AST - Abstract Syntax Tree

- AST składa się z różnych nodów.
- Każdy node to klasa. Hierarchia nodów taka, jak hierarchia klas.

Główne nody (gdzie ‘:’ oznacza dziedziczenie):

- `Stmt`
- `CompoundStmt : Stmt` - blok {}
- `Expr : Stmt` - expression
- `Decl` - deklaracja
- `CXXRecordDecl : (... : ( NamedDecl : Decl))`
- `Type`
- `QualType` - typ z constami i volatilami

Casty i dump AST

W LLVM wszędzie używa się wskaźników bez obawy o zwalnianie pamięci.

Dzięki temu można bez problemu efektywnie castować w górę hierarchii używając:

- `llvm::dyn_cast<Typ>` - zwraca `null`, jeśli się nie uda.
- `llvm::dyn_cast<Typ>` - assercja, jeśli się nie uda.

```
auto *E = llvm::dyn_cast<Expr>(MyStmt);
```

Dumpowanie AST:

```
clang-check file.cpp -ast-dump [-ast-dump-filter=filter]
[-p CD] [-- flagi kompilacji]
```

AST - dokumentacja

http://clang.llvm.org/docs/IntroductionToTheClangAST.html

http://clang.llvm.org/doxygen/classes.html - Doxygen

ASTMatchers

- ASTMatchers to minibiblioteka do pisania zapytań na AST w podobny sposób, jak w SQL.
- Matchery to klasy, które mają metodę `matches() -> bool`.
- Zaczynają się z małej litery, aby móc odróżnić je od klas AST.
- Można (i należy) je przypisywać do zmiennych, np.:

```
auto matcher = cxxRecordDecl(hasName("nazwa_klasy"));
```

- Szablony dbają o to, żeby typy matcherów się zgadzały (inaczej: compilation error).

Node Matchers

- `decl, enumDecl, functionDecl, cxxConstructorDecl, ...`
- `stmt, ifStmt, forStmt, breakStmt, nullStmt, ...`
- `expr, castExpr, callExpr, cxxThisExpr, ...`
- `type, functionType, pointerType, ...`

Matchują się do odpowiednich nodów w AST. Odpowiadają klasom w AST, na przykład `CallExpr` -> `callExpr()`, `CXXConstructorDecl` -> `cxxConstructorDecl()`.

Można wołać na nich `.bind("nazwa")`.

Narrowing Matchers

- `allOf(1, 2, …), anyOf(1, 2, …)`
- `anything(), unless(1)`
- `[NamedDecl] hasName(1), hasAnyName(1, 2, …), ...`
- `[Type] booleanType(), equalsNode(Other), ...`

Matchery ograniczające zbiór wyników do tych spełniających odpowiednie własności.

```
> m namedDecl(anyOf(hasName("__i"), isPrivate()))
```

AST Traversal Matchers

- `eachOf(1, 2, …), forEach(...)`
- `has(...), hasParent(...), hasDescendant(...), hasAncestor(...)`
- `[BinaryOperator] hasLHS(...), hasRHS(...), ...`
- `[CXXConstructExpr] hasDeclaration(...), ...`
- `[Expr] hasType(...), ...`

Ograniczają wyniki na podstawie ułożenia w drzewie AST.

```
> m binaryOperator(anyOf(hasLHS(hasType(isPublic())),
                         hasRHS(callExpr())))
```

Własny matcher

Jeśli nie ma matchera, który udostępnia szukaną funkcjonalność, ale da się ją łatwo wyciągnąć z API AST, to możemy napisać własny matcher, używając makra `AST_MATCHER` lub `AST_MATCHER_P`, np:

```
AST_MATCHER(FieldDecl, isOneBitBitField) {
  return Node.isBitField() &&
      Node.getBitWidthValue(Finder->getASTContext()) == 1;
}
```

```
> m fieldDecl(isOneBitBitField())
```

Dokumentacja ASTMatchers

http://clang.llvm.org/docs/LibASTMatchersReference.html

http://clang.llvm.org/docs/LibASTMatchers.html

http://clang.llvm.org/doxygen/classes.html

+ unit testy
+ inne checki

Clang-query - tworzenie matcherów

Clang-query to prosty shell do pisania i uruchamiania matcherów.

- Skompilowane z libedit podpowiada składnię.
- Można nazywać matchery za pomocą składni “let nazwa matcher”.
- Są bardzo małe rozbieżności z clang-tidy.

```
clang-query file.cpp [-p DC][-- flagi]
```

Pomocny skrypt

```
clang-tidy/add_new_check.py
```

- Tworzy check w *clang-tidy/module/CheckName{.h|.cpp}*.
- Dodaje odpowiednie opcje do CMake.
- Tworzy *test/clang-tidy/module-check-name.cpp*.
- Tworzy *docs/clang-tidy/checks/module-check-name.rst*.

Należy tylko samemu dodać notkę w *ReleaseNotes.rst*.

Zadanie: modernize-use-emplace

Napisz matcher, który szuka wywołania `push_back` na `std::vector`, które należałoby zmienić na wywołania `emplace_back`.

```
vector<Obj> v1;
vector<pair<int, int>> v2;
vector<optional<string>> v3;


v1.push_back(Obj(42));   // v1.emplace_back(42);
v2.push_back(std::make_pair(42, 42));
// v2.emplace_back(42, 42);
v3.push_back("abc");   // v3.emplace_back("abc");
```

Diagnostyki w Clang-Tidy

```
DiagnosticBuilder diag(Location, Description, [Level])
```

- Wyrzuca diagnostykę (najczęściej warning) w podanym miejscu kodu i o danym opisie; za pomocą operator<< możemy podpowiadać fixy i przekazywać parametry:
  - `FixItHint::CreateInsertion(Loc, Code)`
  - `FixItHint::CreateRemoval(Range)`
  - `FixItHint::CreateReplacement(Range, Code)`

```
SourceLocation LocStart = /* ... */, LocEnd = /* ... */;
diag(LocStart, "potentially unintended semicolon")
    << FixItHint::CreateRemoval(SourceRange(LocStart, LocEnd));
```

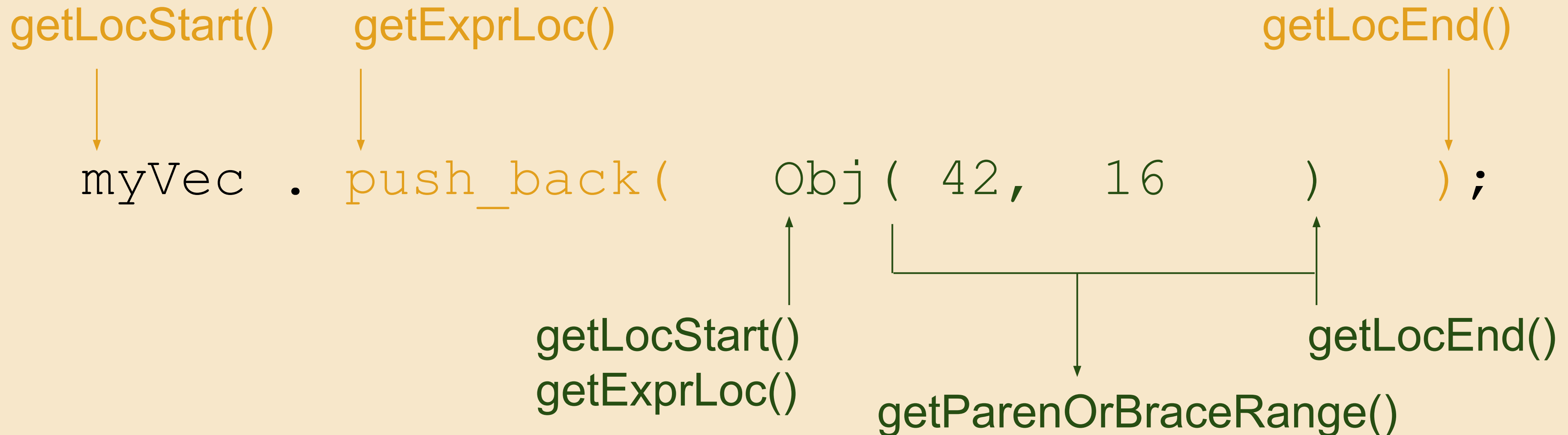
Gdzie jestem w kodzie?

- `SourceLocation:`
  - wskazuje miejsce w kodzie
- `SourceRange:`
  - [początkowy znak, końcowy znak)
  - [początek pierwszego tokena, początek końcowego tokena)
- `CharSourceRange:`
  - [początkowy znak, końcowy znak)

Gdzie jestem w kodzie?

push_back(...): CXXMemberCallExpr

Obj(...): CXXConstructExpr



Co chcemy zrobić?

```
myVec . push_back(    Obj( 42,   16     )    );
```

emplace_back(

usunąć

Przydatne linki

Nagranie z tego wykładu wkrótce

http://bbannier.github.io/blog/2015/05/02/Writing-a-basic-clang-static-analysis-check.html

https://youtu.be/1S2A0VWGOws

https://youtu.be/nzCLcfH3pb0

https://youtu.be/VqCkCDFLSsc