Demo 1:

1) Dump the AST (same thing, either way)
    a. clang -c -Xclang -ast-dump init.cpp
    b. clang -cc1 -ast-dump init.cpp
    c. Notice a few things ..
        i. Built-in types – va_list, NSConstantString, __int128
        ii. Notice the function definition is represented in AST
        iii. We'll come back to this later.
2) Show list of clang-tidy checks enabled by default
    a. clang-tidy --list-checks
3) Show list of all available clang-tidy checks
    a. clang-tidy --list-checks –checks=*
    b. clang-tidy --list-checks --checks=* | grep change (notice that a checker with "change" is not present).
4) Display init.cpp, see local variables are not initialized.
5) Run checker to check for uninitialized variables, then fix the results …
    a. clang-tidy --checks=-*,cppcoreguidelines-init-variables init.cpp --
    b. clang-tidy --checks=-*,cppcoreguidelines-init-variables init.cpp -- --fix
6) The "Fix" did a few things …
    a. Initialized all uninitialized local variables.
    b. Added "math.h" for the NAN define.
7) You can do this too!!!!

Demo 2: Create the boiler plate for the checker we'll write.

1) cd to ~/llvm-project-add-example/clang-tools-extra/clang-tidy
    a. Clean up: git reset --hard; git clean -f
2) ./add_new_check.py misc change-malloc
3) See files created, modified …

```
Updating ./misc/CMakeLists.txt...
Creating ./misc/ChangeMallocCheck.h...
Creating ./misc/ChangeMallocCheck.cpp...
Updating ./misc/MiscTidyModule.cpp...
Updating ../docs/ReleaseNotes.rst...
Creating ../test/clang-tidy/checkers/misc-change-malloc.cpp...
Creating ../docs/clang-tidy/checks/misc-change-malloc.rst...
Updating ../docs/clang-tidy/checks/list.rst...
Done. Now it's your turn!
```

4) Notice new files, changed files.
5) Look at new header file, new implement, and test. We'll ignore the documentation files for now.
6) Build, notice the new checker showing up in the list?
    a. clang-tidy --list-checks --checks=* | grep change

Demo 3: Clang-query

Demo3 : demo3.c source code. AST dump.

```
#include <stdlib.h>
#include <memory.h>
int foo(void) {
    void *ptr = malloc(4);

    free(ptr);
    return 0;
}
int fee(int i) {
    return i*2;
}
int gee(int i) {
    return i/2;
}
int anError(int i) {
    return i/0;
}
```

```
    1)  Load this source into clang-query
            a.  clang-query  malloc.c –
    2)  display all functionDecl()’s
                m functionDecl()
            a.  210 matches for me, wow!! Too much.
    3)  When working with real examples in clang-query, it’s best to do one of two things.
            a.  Reduce the code example under exploration, or …
            b.  Use a narrowing search.
    4)  Try …
                m functionDecl(isExpansionInMainFile())
            a.  Better, down to 4 matches in malloc.c
            b.
    5)  Display all variable declarations …
                m varDecl(isExpansionInMainFile())
            a.  4 matches – size_t, void *, one not displayed – hmmm? What to do?
    6)  Set output for “detailed-ast”
            a.  set output detailed-ast
                m varDecl(isExpansionInMainFile())
            b.  Now we can see the detailed AST. We can use this to help us with “narrowing”
                matchers.
    7)  Narrow function decls…
                m
                functionDecl(isExpansionInMainFile(),anyOf(hasName("malloc"),hasName("free"),hasNa
                me("foo")))
    8)  Match just “fee”
                m functionDecl(isExpansionInMainFile(),hasName("fee"))
    9)  Regex search for any function decl starting with “f”
                m functionDecl(isExpansionInMainFile(),matchesName("f"))
    10) Search for binary operators
            a.  m binaryOperator(isExpansionInMainFile())
    11) Search for operator “/”
                m binaryOperator(isExpansionInMainFile(),hasOperatorName("/"))
    12) Search for operator “*”
                m binaryOperator(isExpansionInMainFile(),hasOperatorName("*"))
    13) Search for multiplication by 0
                m
                binaryOperator(isExpansionInMainFile(),hasOperatorName("*"),hasRHS(integerLiteral(
                equals(0))))
    14) Search for division by 0
                m
                binaryOperator(isExpansionInMainFile(),hasOperatorName("/"),hasRHS(integerLiteral(
                equals(0))))
```

15) ok, enough playing 😊 . From these examples, you can see how to use the libMatchers help and clang-query to explore AST matches. Let's keep going, find a good match to implement in code. We're interested in finding callsites to "malloc", perhaps excluding one callsite (we'll come to this). This will also apply to other functions we might also replace, like "free"
16) Find all Call expressions
    a. `m callExpr()`
17) Narrow to all call expressions that have function declarations. Callee is what we call a "traversal matcher". These specify the relationship to other nodes that are reachable from the current node.

```
        m callExpr(callee()) - oops, an error. Callee must have an argument.
        m callExpr(callee(functionDecl()))
```

18) Narrow to name "malloc"

```
        m callExpr(callee(functionDecl(hasName("malloc"))))
```

19) Similar for "free"

```
        m callExpr(callee(functionDecl(hasName("free"))))
```

Step 1: Change…

- Just changes "malloc" to "acme_zalloc".
- You've already heard there's more bugs in the chip, but you're told to stay focused on this for now 😊

ChangeMallocCheck.cpp:

```
void ChangeMallocCheck::registerMatchers(MatchFinder *Finder) {
  Finder->addMatcher(callExpr(callee(functionDecl(hasName("malloc")))).bind("malloc"),this);
}

void ChangeMallocCheck::check(const MatchFinder::MatchResult &Result) {
  const CallExpr *callExpr = Result.Nodes.getNodeAs<CallExpr>("malloc");
  if (callExpr) {
    auto start = callExpr->getBeginLoc();
    auto Diag = diag(start, "use acme_zalloc() instead of malloc()")
      << FixItHint::CreateReplacement(SourceRange(start, start.getLocWithOffset(strlen("malloc")-
1)), "acme_zalloc");
  }
}
```

No changes needed to ChangeMallocCheck.h

Build, try it.

Step 2 – Replace free, extend malloc

```
void ChangeMallocCheck::registerMatchers(MatchFinder *Finder) {
  Finder->addMatcher(callExpr(callee(functionDecl(hasName("malloc")))).bind("malloc"),this);
  Finder->addMatcher(callExpr(callee(functionDecl(hasName("free")))).bind("free"),this);
}

void ChangeMallocCheck::check(const MatchFinder::MatchResult &Result) {
  SmallString<64> NewArgument;
  const CallExpr *callExpr = Result.Nodes.getNodeAs<CallExpr>("malloc");
  if (callExpr) {
    auto start = callExpr->getBeginLoc();
    auto Diag = diag(start, "use acme_zalloc() instead of malloc()")
      << FixItHint::CreateReplacement(SourceRange(start, start.getLocWithOffset(strlen("malloc")-
1)), "acme_zalloc");
    NewArgument = Twine(", ZERO_INITIALIZE").str();
    const auto InsertNewArgument = FixItHint::CreateInsertion(callExpr->getEndLoc(), 
NewArgument);
    Diag << InsertNewArgument;
  }
  callExpr = Result.Nodes.getNodeAs<CallExpr>("free");
  if (callExpr) {
    auto start = callExpr->getBeginLoc();
    auto Diag = diag(start, "use acme_free() instead of free()")
      << FixItHint::CreateReplacement(SourceRange(start, start.getLocWithOffset(strlen("free")-
1)), "acme_free");
    Diag << FixItHint::CreateInsertion(callExpr->getArg(0)->getBeginLoc(), "(void **)&");
  }
}
```