How to Write a Checker in 24 Hours

Clang Static Analyzer

Anna Zaks and Jordan Rose
Apple Inc.

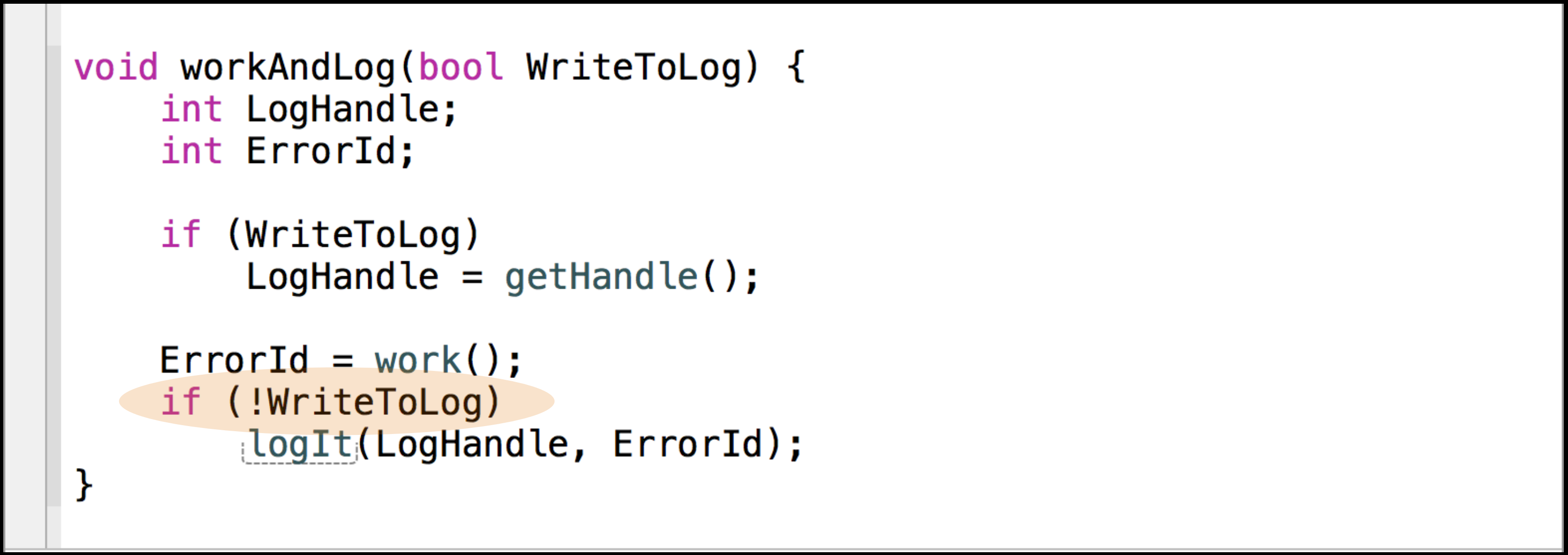
What is this talk about?

- The Clang Static Analyzer is a bug finding tool
- It can be extended with custom “checkers”
- We’ll teach you how to write your own checker

Warnings are great!

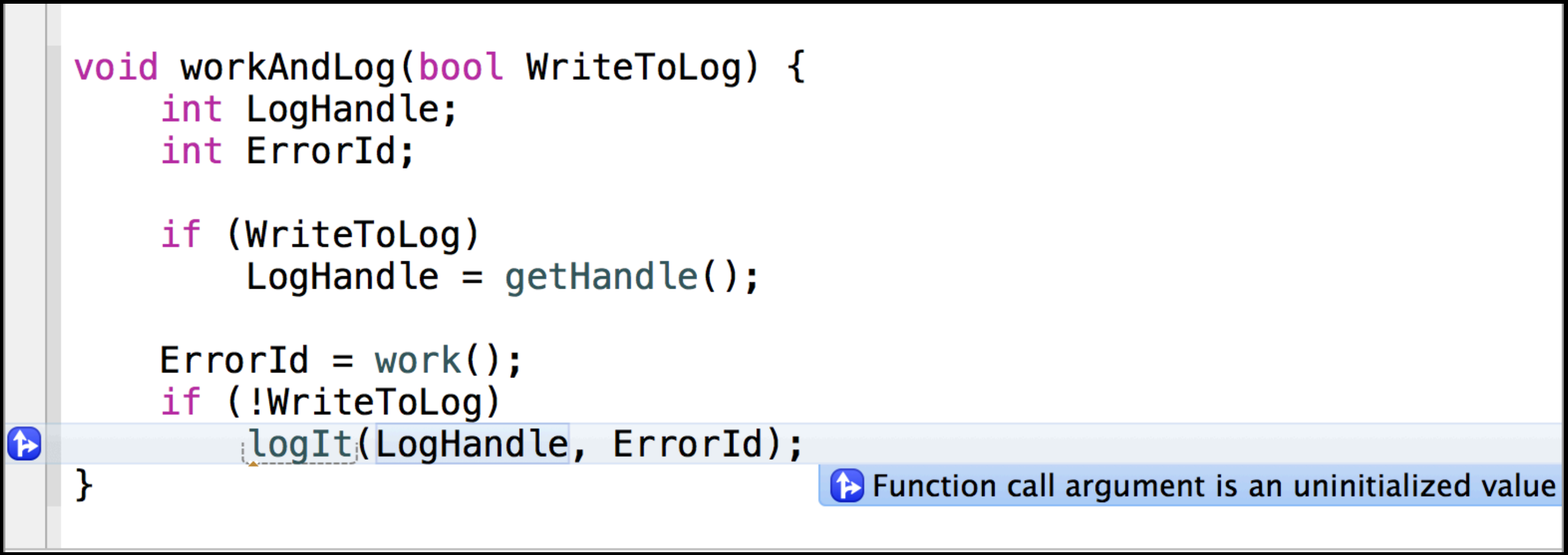
- Provide free and fast code review
- Catch errors early in the development cycle

Compiler Warnings are Limited



```
void workAndLog(bool WriteToLog) {
    int LogHandle;
    int ErrorId;

    if (WriteToLog)
        LogHandle = getHandle();

    ErrorId = work();
    if (!WriteToLog)
        logIt(LogHandle, ErrorId);
}
```

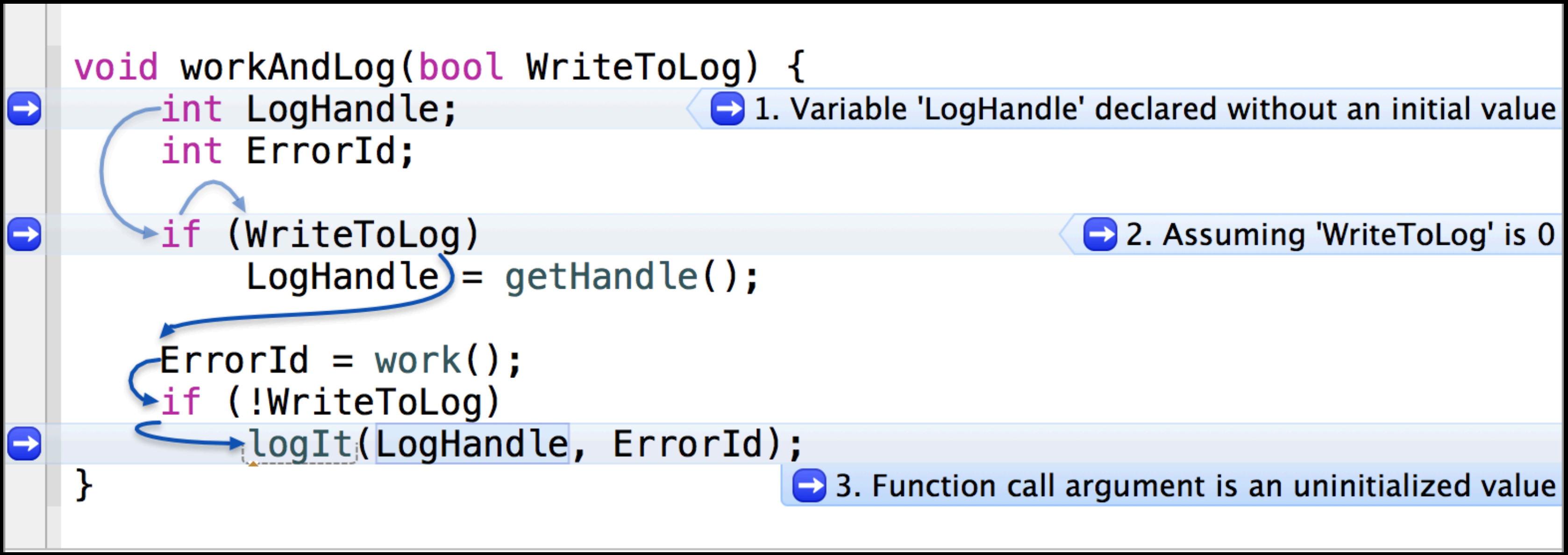
Static Analyzer to the Rescue



```
void workAndLog(bool WriteToLog) {
    int LogHandle;
    int ErrorId;

    if (WriteToLog)
        LogHandle = getHandle();

    ErrorId = work();
    if (!WriteToLog)
        logIt(LogHandle, ErrorId);
}
```

Function call argument is an uninitialized value

Static Analyzer to the Rescue



```
void workAndLog(bool WriteToLog) {
    int LogHandle;
    int ErrorId;

    if (WriteToLog)
        LogHandle = getHandle();

    ErrorId = work();
    if (!WriteToLog)
        logIt(LogHandle, ErrorId);
}
```

1. Variable 'LogHandle' declared without an initial value

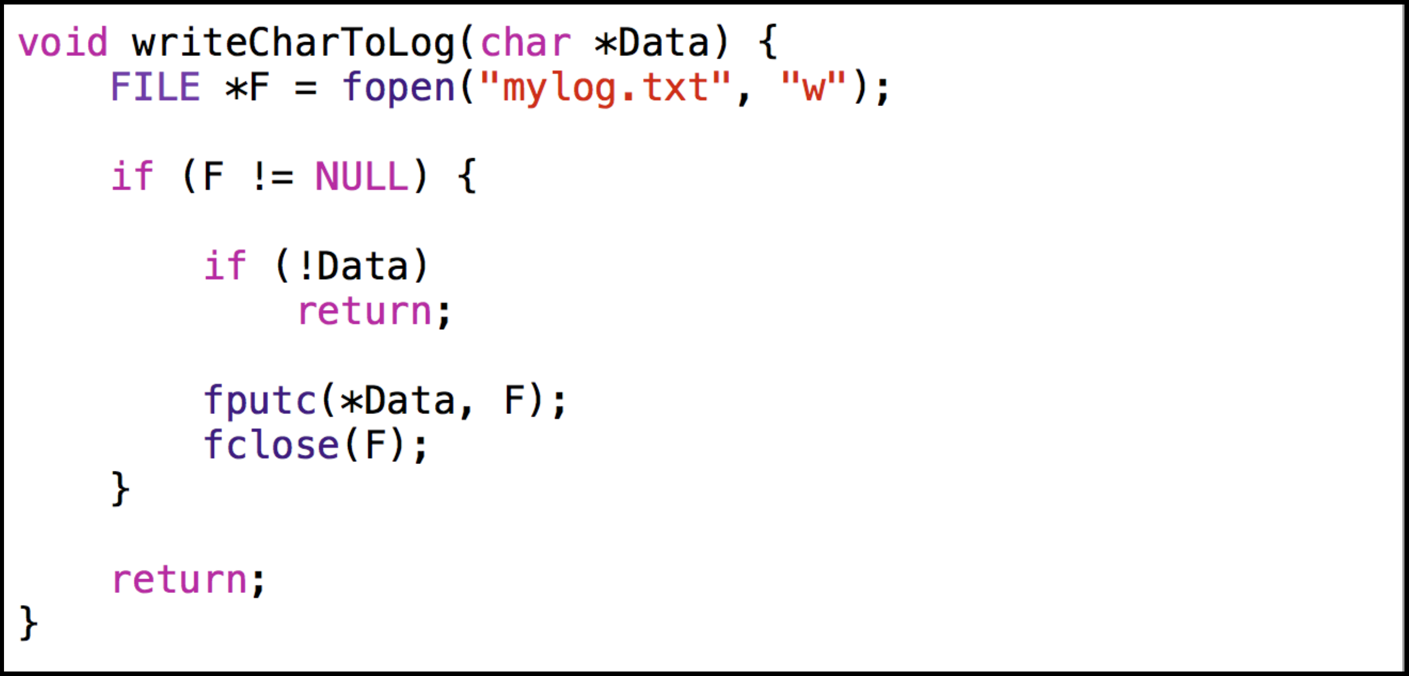
2. Assuming 'WriteToLog' is 0

3. Function call argument is an uninitialized value

Why Static Analysis?

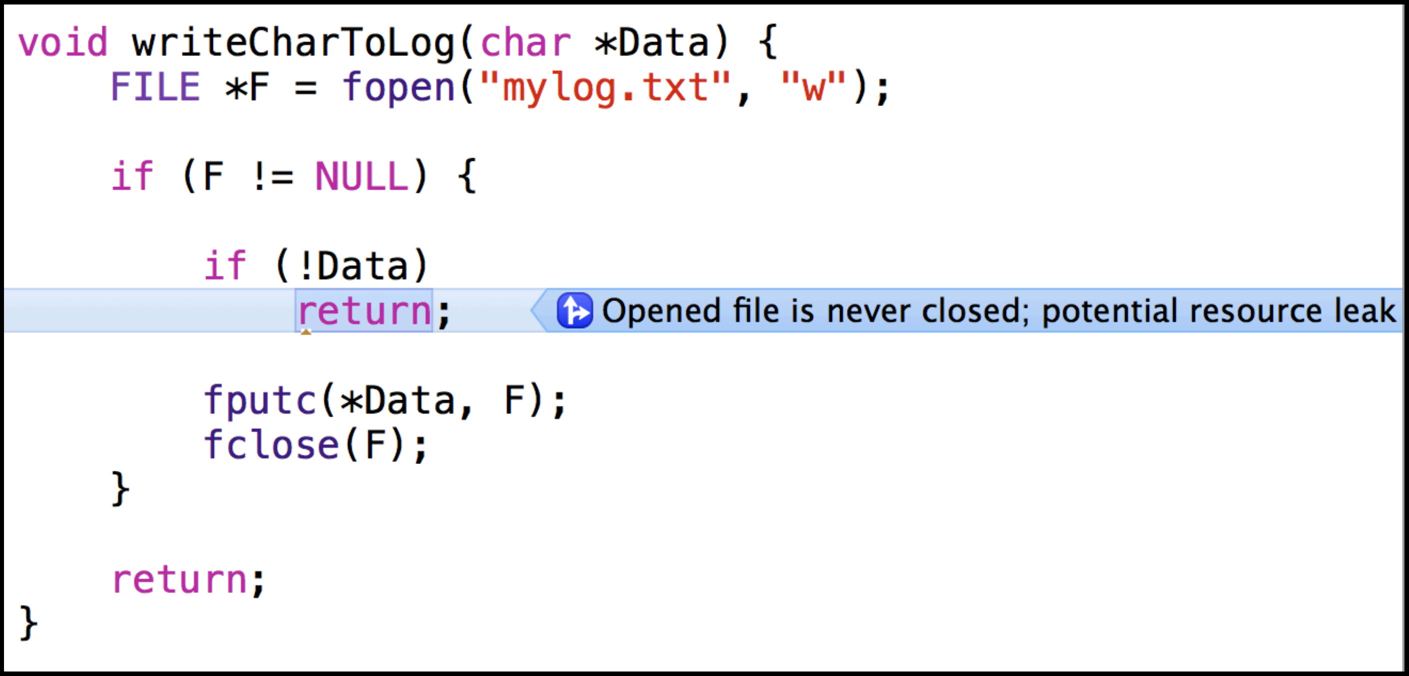
- Explores each path through the program
  - Path-sensitive, context-sensitive analysis
  - Algorithm is exponential (but bounded)
- Produces very precise results
- Able to find more bugs
  - use-after-free
  - resource leaks
  - ...

Check that a File is Closed on *each* Path



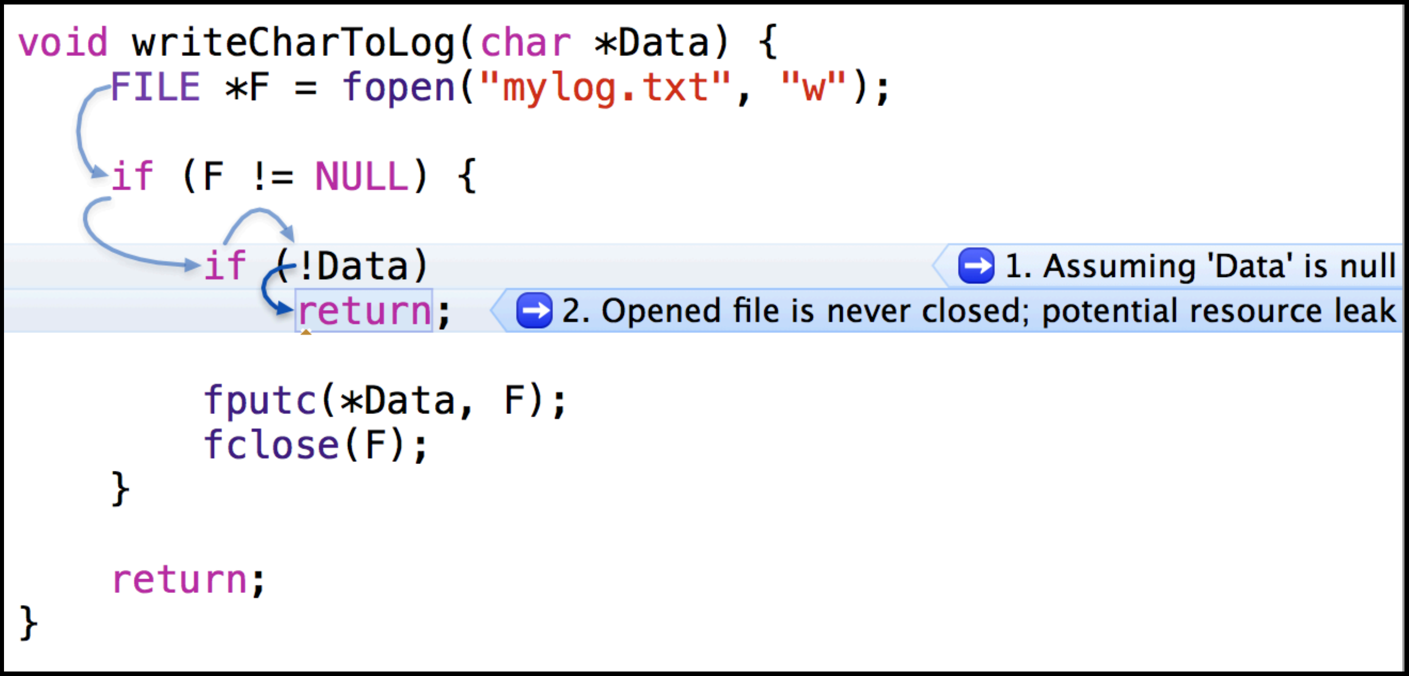
```
void writeCharToLog(char *Data) {
    FILE *F = fopen("mylog.txt", "w");

    if (F != NULL) {

        if (!Data)
            return;

        fputc(*Data, F);
        fclose(F);
    }

    return;
}
```

Check that a File is Closed on *each* Path



```
void writeCharToLog(char *Data) {
    FILE *F = fopen("mylog.txt", "w");

    if (F != NULL) {

        if (!Data)
            return;     Opened file is never closed; potential resource leak

        fputc(*Data, F);
        fclose(F);
    }

    return;
}
```

Check that a File is Closed on *each* Path

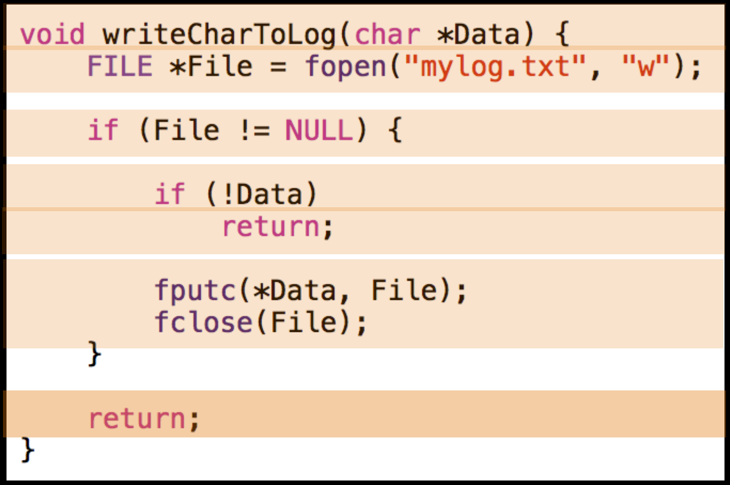


```
void writeCharToLog(char *Data) {
    FILE *F = fopen("mylog.txt", "w");

    if (F != NULL) {

        if (!Data)                          1. Assuming 'Data' is null
            return;     2. Opened file is never closed; potential resource leak

        fputc(*Data, F);
        fclose(F);
    }

    return;
}
```

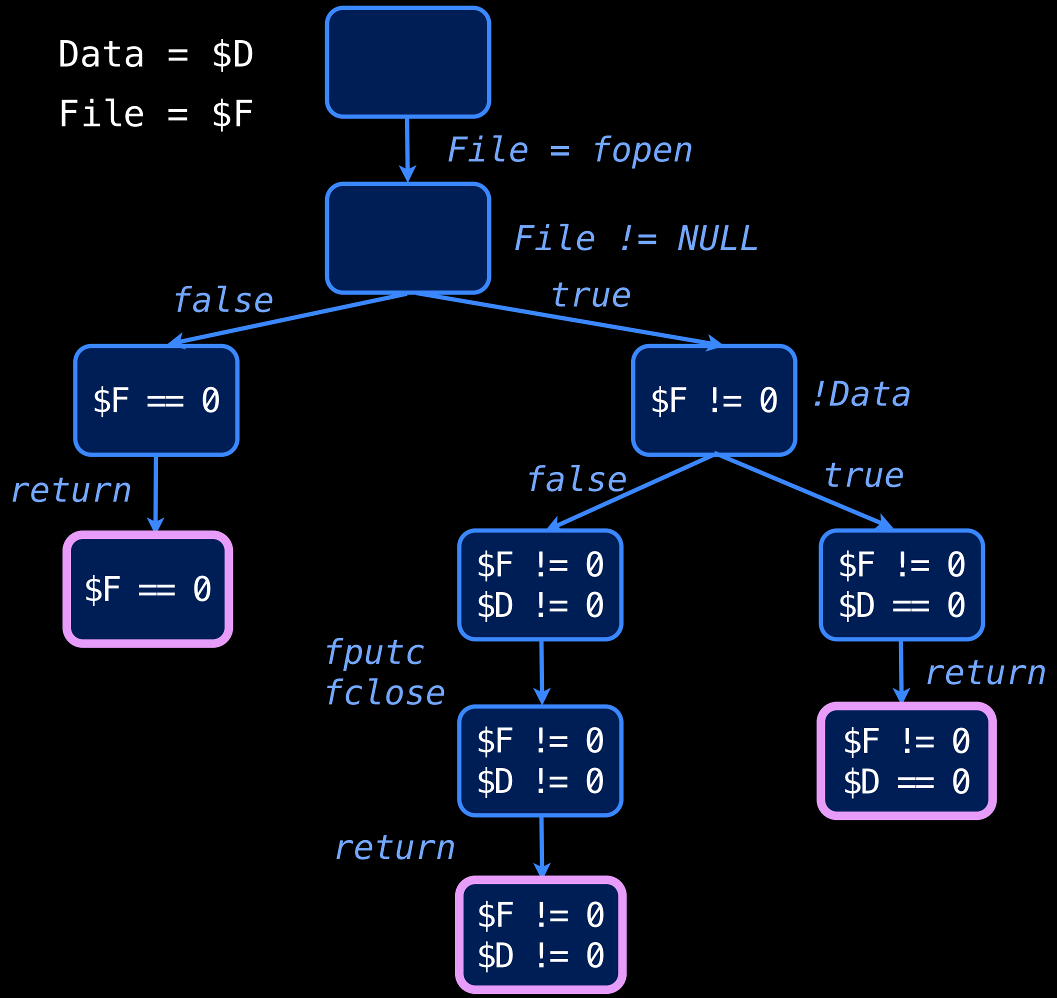
Symbolic Execution

- Performs a path-sensitive walk of Clang's CFG
- Similar to program execution but
  - Explores every possible path through the program
  - Uses symbolic values
- Collects the constraints on symbolic values along each path
- Uses constraints to determine feasibility of paths

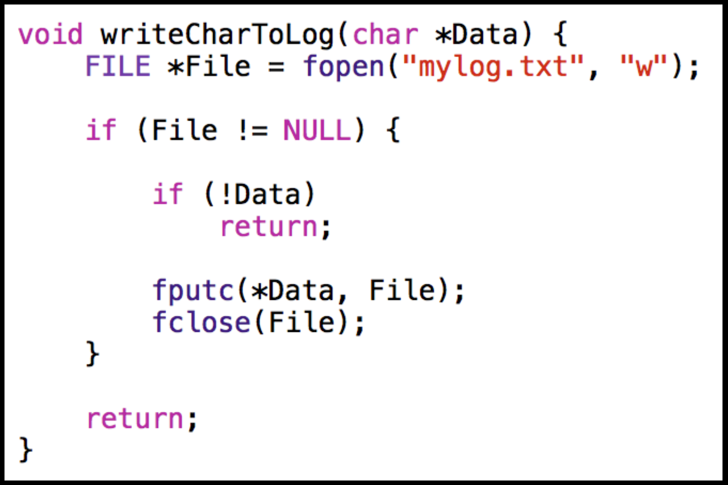
Builds a Graph of Reachable Program States



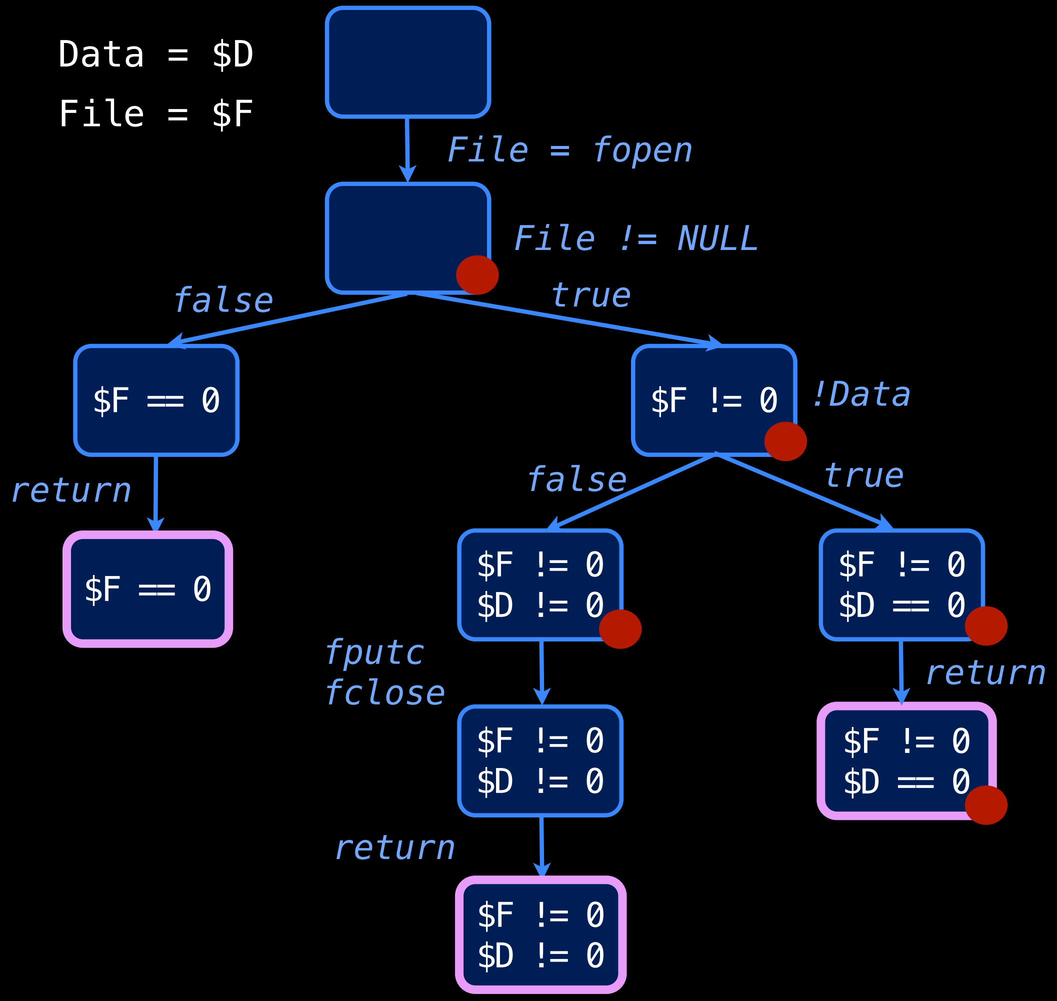
```
void writeCharToLog(char *Data) {
    FILE *File = fopen("mylog.txt", "w");

    if (File != NULL) {

        if (!Data)
            return;

        fputc(*Data, File);
        fclose(File);
    }

    return;
}
```



Builds a Graph of Reachable Program States



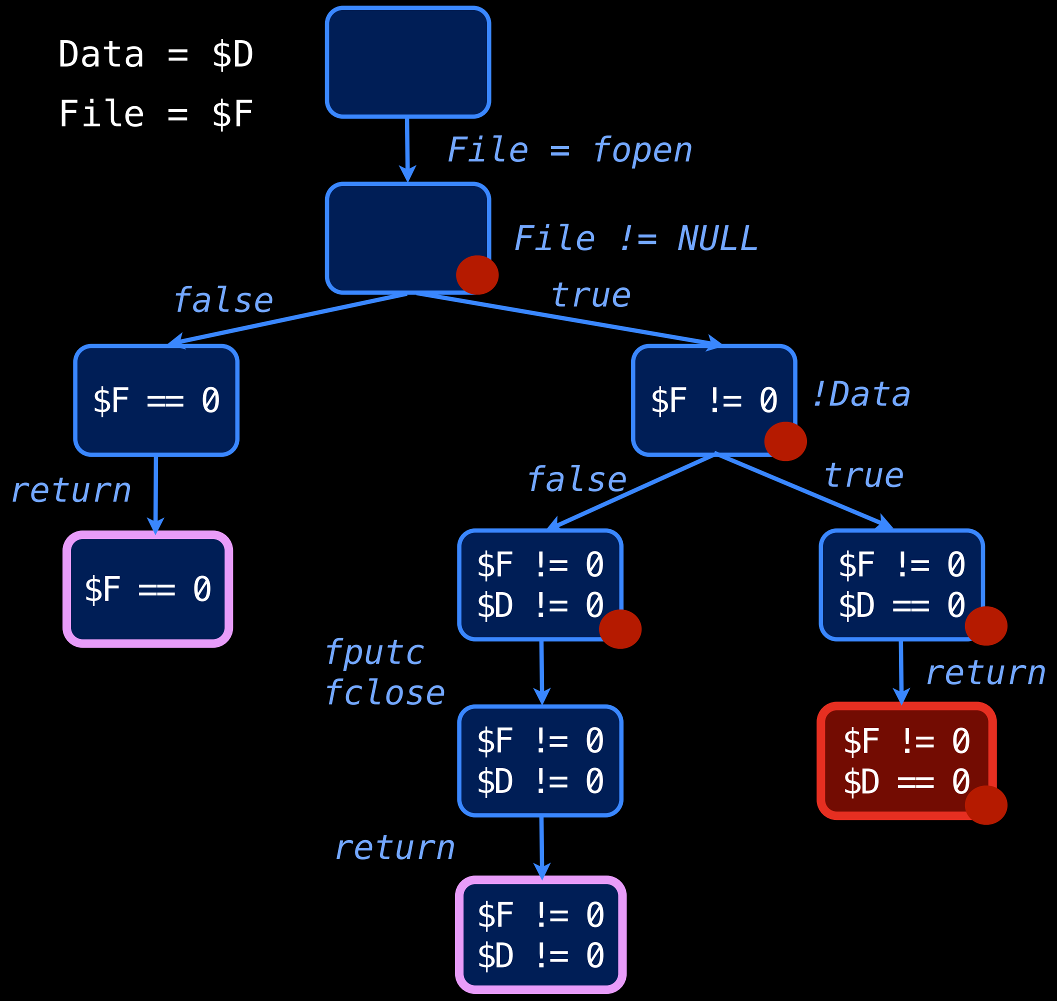
```
void writeCharToLog(char *Data) {
    FILE *File = fopen("mylog.txt", "w");

    if (File != NULL) {

        if (!Data)
            return;

        fputc(*Data, File);
        fclose(File);
    }

    return;
}
```



● Denotes that the file is open

Finding a Bug ~ Graph Reachability

```c
void writeCharToLog(char *Data) {
    FILE *File = fopen("mylog.txt", "w");

    if (File != NULL) {

        if (!Data)
            return;

        fputc(*Data, File);
        fclose(File);
    }

    return;
}
```



● Denotes that the file is open

What's in a Node?

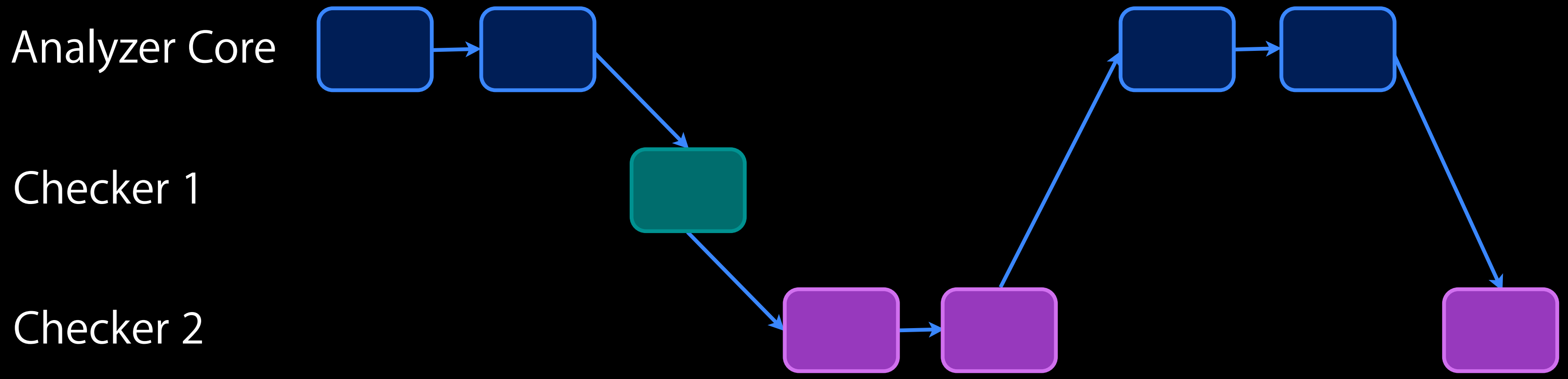
Program Point | Program State

- Execution location
  - pre-statement
  - post-statement
  - entering a call
  - ...
- Stack frame

- Environment: `Expr` -> values
- Store: memory location -> values
- Constraints on symbolic values
- Generic Data Map (GDM)

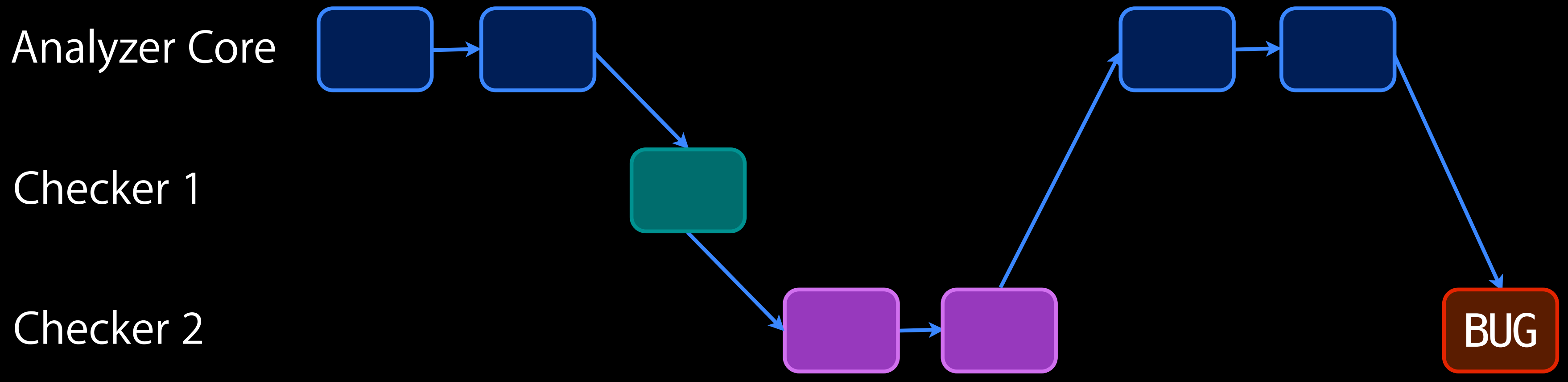
Extending with Checkers

- Checkers participate in the graph construction



Extending with Checkers

- Checkers participate in the graph construction
- Checkers can stop path exploration by creating sink nodes



Checkers are Visitors

```
checkPreStmt (const ReturnStmt *S, CheckerContext &C) const
```

Before return statement is processed

```
checkPostCall (const CallEvent &Call, CheckerContext &C) const
```

After a call has been processed

```
checkBind (SVal L, SVal R, const Stmt *S, CheckerContext &C) const
```

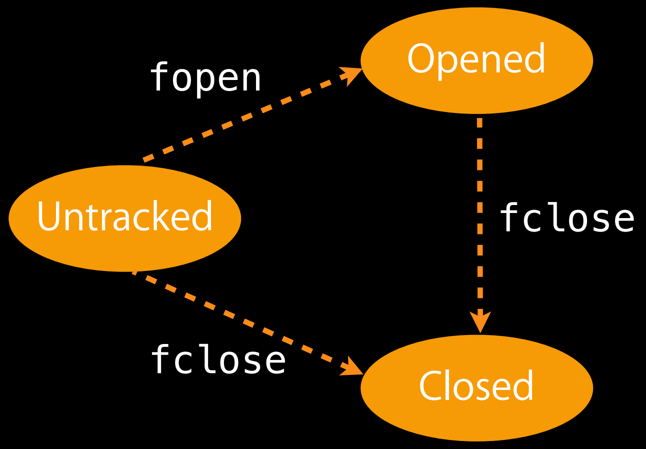
On binding of a value to a location as a result of processing the statement

See the checker writer page for more details:
http://clang-analyzer.llvm.org/checker_dev_manual.html

Let’s Write a Unix Stream API Checker

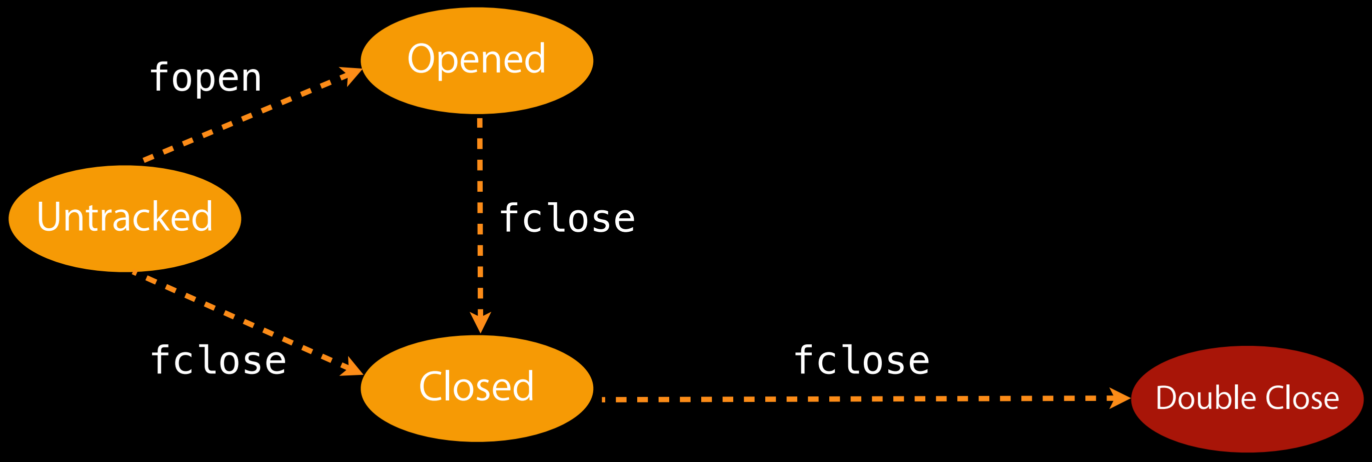
Stream State Transitions

- File handle state changes are driven by the API calls



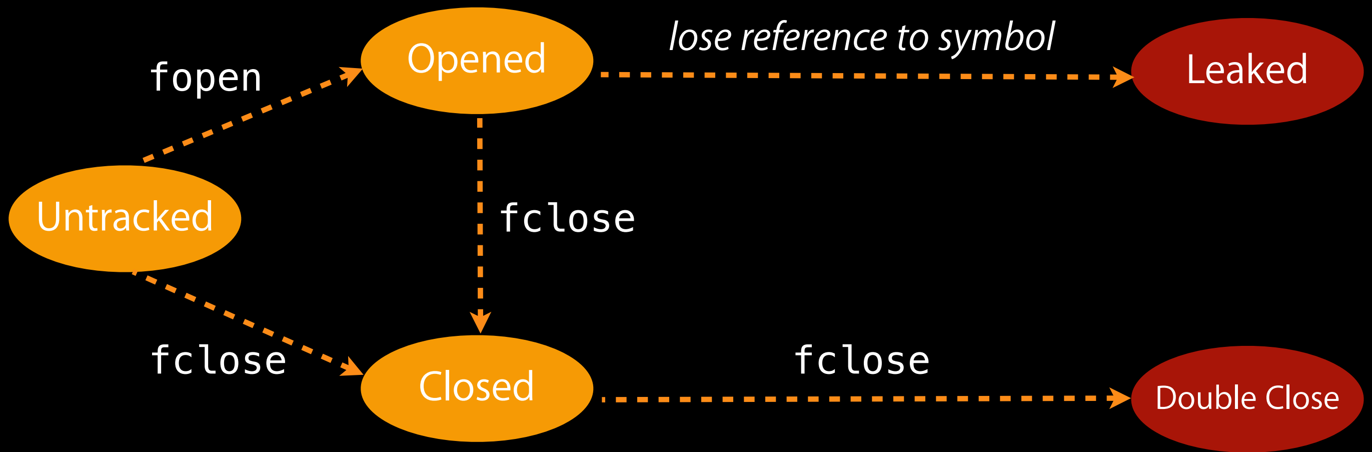
Stream State Transitions

- File handle state changes are driven by the API calls
- Error States:
  - If a file has been closed, it should not be accessed again.



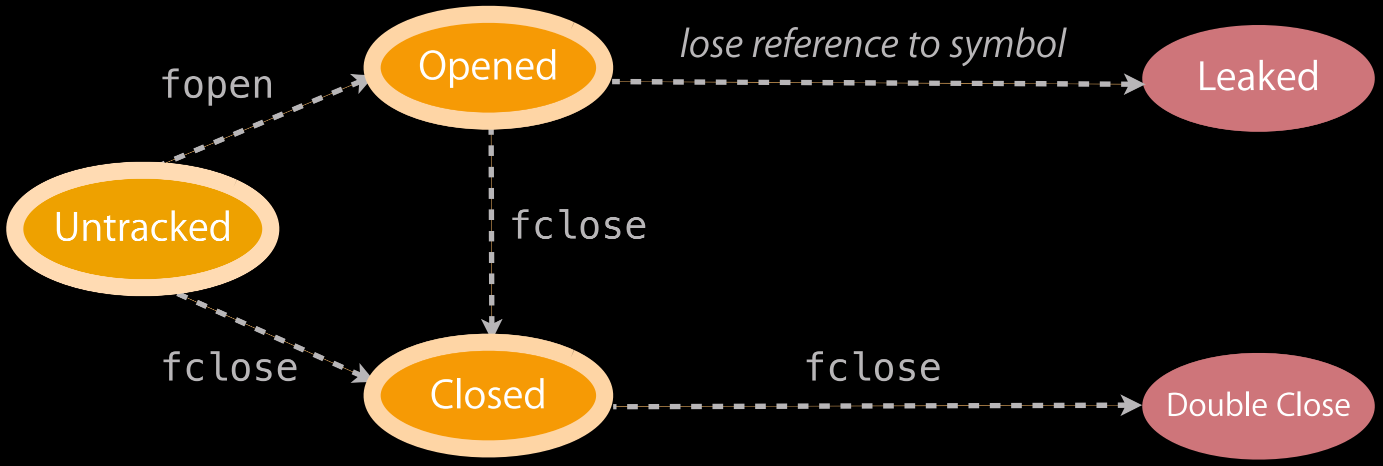
Stream State Transitions

- File handle state changes are driven by the API calls
- Error States:
  - If a file has been closed, it should not be accessed again.
  - If a file was opened with `fopen`, it must be closed with `fclose`



Stream Checker Recipe

- Define the state of a file descriptor
- Add state transition corresponding to `fopen`
- Add transitions driven by `fclose`
- Report error on double close
- Report error on leak



Defining the State of a File Descriptor

```
struct StreamState {
  enum Kind { Opened, Closed } K;

  StreamState(Kind InK) : K(InK) { }
};
```

Defining the State of a File Descriptor

```
struct StreamState {
  enum Kind { Opened, Closed } K;

  StreamState(Kind InK) : K(InK) { }

  bool operator==(const StreamState &X) const {
    return K == X.K;
  }
  void Profile(llvm::FoldingSetNodeID &ID) const {
    ID.AddInteger(K);
  }





};
```

Defining the State of a File Descriptor

```
struct StreamState {
  enum Kind { Opened, Closed } K;

  StreamState(Kind InK) : K(InK) { }

  bool operator==(const StreamState &X) const {
    return K == X.K;
  }
  void Profile(llvm::FoldingSetNodeID &ID) const {
    ID.AddInteger(K);
  }

  bool isOpened() const { return K == Opened; }
  bool isClosed() const { return K == Closed; }

  static StreamState getOpened() { return StreamState(Opened); }
  static StreamState getClosed() { return StreamState(Closed); }
};
```

Checker State is Part of `ProgramState`

- We need to store a map from file handles to `StreamState`

Checker State is Part of `ProgramState`

- We need to store a map from file handles to `StreamState`
- When we change states, we'll need to update the map

```
State = State->set<StreamMap>(FileDesc, StreamState::getOpened());
```

Checker State is Part of `ProgramState`

- We need to store a map from file handles to `StreamState`
- When we change states, we'll need to update the map

```
State = State->set<StreamMap>(FileDesc, StreamState::getOpened());
```

- Later, we can retrieve the state from the map

```
const StreamState *SS = State->get<StreamMap>(FileDesc);
```

Checker State is Part of `ProgramState`

- We need to store a map from file handles to `StreamState`
- When we change states, we'll need to update the map

```
State = State->set<StreamMap>(FileDesc, StreamState::getOpened());
```

- Later, we can retrieve the state from the map

```
const StreamState *SS = State->get<StreamMap>(FileDesc);
```

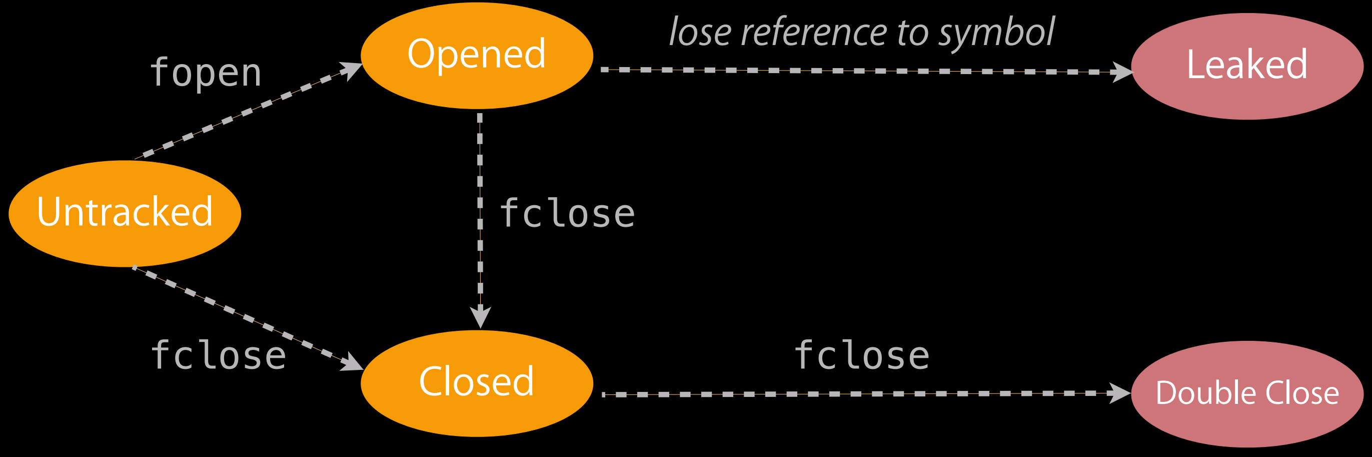
- The map itself must be registered in advance

```
/// Register a map from tracked stream symbols to their state.
REGISTER_MAP_WITH_PROGRAMSTATE(StreamMap, SymbolRef, StreamState)
```

Stream Checker Recipe

✓ Define the state of a file descriptor

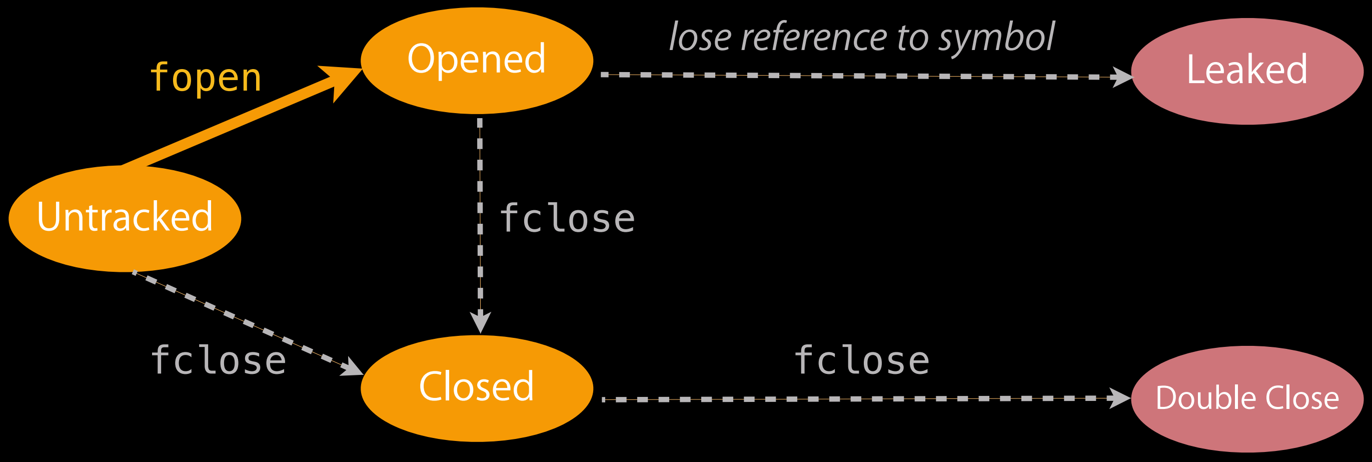
- Add state transition corresponding to `fopen`
- Add transitions driven by `fclose`
- Report error on double close
- Report error on leak



Stream Checker Recipe

✓ Define the state of a file descriptor

- Add state transition corresponding to `fopen`
- Add transitions driven by `fclose`
- Report error on double close
- Report error on leak



Register for fopen

```
class SimpleStreamChecker : public Checker<check::PostCall> {

public:
  /// Process fopen.
  void checkPostCall(const CallEvent &Call, CheckerContext &C) const;
};
```

- Visitor callbacks are implemented using templates
- A `PostCall` visit means the checker is called after processing a call
- Checkers are stateless! State belongs in `ProgramState`

Process fopen

```
void SimpleStreamChecker::checkPostCall(const CallEvent &Call,
                                        CheckerContext &C) const {
  if (!Call.isGlobalCFunction("fopen"))
    return;









}
```

Process `fopen`

```
void SimpleStreamChecker::checkPostCall(const CallEvent &Call,
                                        CheckerContext &C) const {
  if (!Call.isGlobalCFunction("fopen"))
    return;

  // Get the symb                                   file handle.
  SymbolRef FileD                                  Symbol();
  if (!FileDesc)
    return;






}
```

`Call.getReturnValue()`

- SVals are **symbolic execution values**
  - Transient, like values in a real program!

Process `fopen`

```
void SimpleStreamChecker::checkPostCall(const CallEvent &Call,
                                        CheckerContext &C) const {
  if (!Call.isGlobalCFunction("fopen"))
    return;

  // Get the symbolic value corresponding to the file handle.
  SymbolRef FileDesc = Call.getReturnValue().getAsSymbol();
  if (!FileDesc)
    return;





}
```

- SVals are **symbolic execution values**
  - Transient, like values in a real program!
- Symbols are a **persistent** representation of opaque values

Process fopen

```
void SimpleStreamChecker::checkPostCall(const CallEvent &Call,
                                        CheckerContext &C) const {
  if (!Call.isGlobalCFunction("fopen"))
    return;

  // Get the symbolic value corresponding to the file handle.
  SymbolRef FileDesc = Call.getReturnValue().getAsSymbol();
  if (!FileDesc)
    return;

  // Generate the next transition (an edge in the exploded graph).
  ProgramStateRef State = C.getState();
  State = State->set<StreamMap>(FileDesc, StreamState::getOpened());
  C.addTransition(State);
}
```

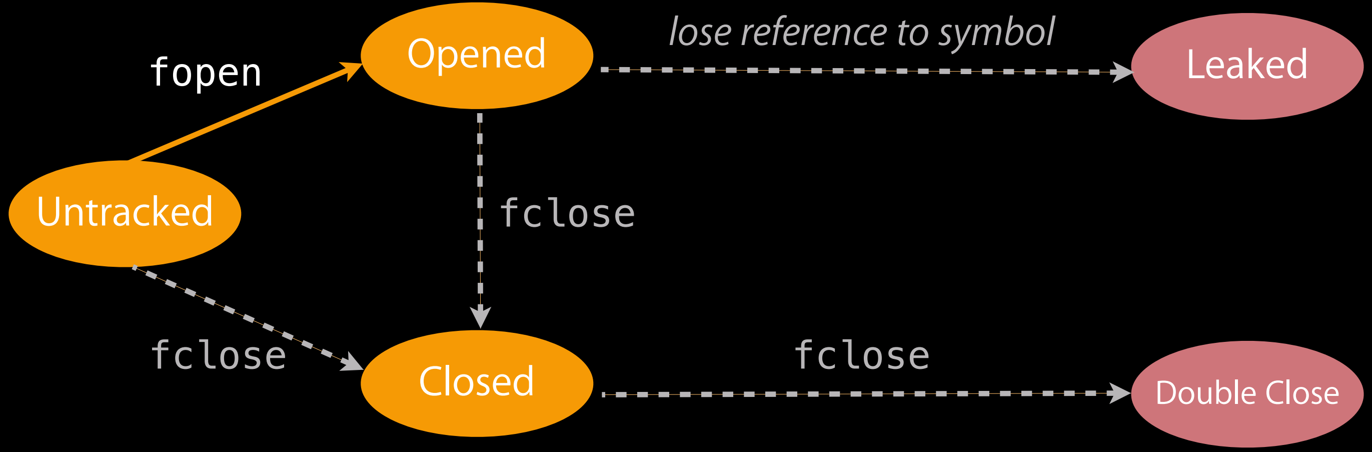
- SVals are **symbolic execution values**
  - Transient, like values in a real program!
- Symbols are a **persistent** representation of opaque values
- Checkers add new nodes to the graph with `addTransition`

Stream Checker Recipe

✓ Define the state of a file descriptor

✓ Add state transition corresponding to `fopen`

- Add transitions driven by `fclose`
- Report error on double close
- Report error on leak

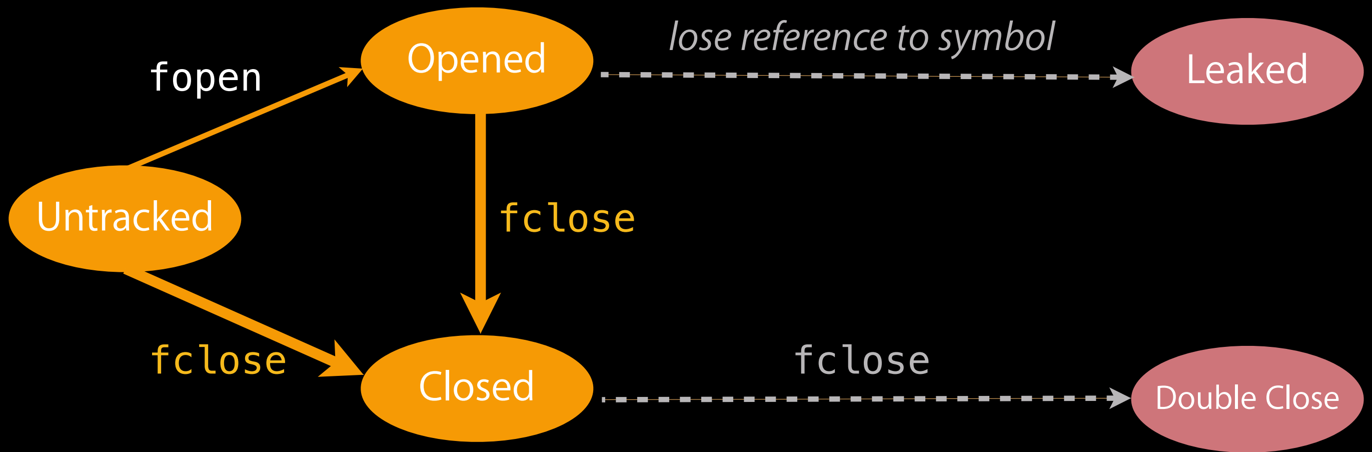


Stream Checker Recipe

✓ Define the state of a file descriptor

✓ Add state transition corresponding to `fopen`

- Add transitions driven by `fclose`
- Report error on double close
- Report error on leak



Register for `fclose`

```
class SimpleStreamChecker : public Checker <check::PostCall,
                                            check::PreCall > {

public:
  /// Process fopen.
  void checkPostCall(const CallEvent &Call, CheckerContext &C) const;

  /// Process fclose.
  void checkPreCall(const CallEvent &Call, CheckerContext &C) const;
};
```

- `PreCall` allows us to check the parameters before the call is processed

Process fclose

```
void SimpleStreamChecker::checkPreCall(const CallEvent &Call,
                                       CheckerContext &C) const {
  // Prototype for fclose is
  //   int fclose(FILE *FileDesc);
  if (!Call.isGlobalCFunction("fclose") || Call.getNumArgs() != 1)
    return;










}
```

Process fclose

```
void SimpleStreamChecker::checkPreCall(const CallEvent &Call,
                                       CheckerContext &C) const {
  // Prototype for fclose is
  //   int fclose(FILE *FileDesc);
  if (!Call.isGlobalCFunction("fclose") || Call.getNumArgs() != 1)
    return;

  // Get the symbolic value corresponding to the file handle.
  SymbolRef FileDesc = Call.getArgSVal(0).getAsSymbol();
  if (!FileDesc)
    return;





}
```

Process `fclose`

```
void SimpleStreamChecker::checkPreCall(const CallEvent &Call,
                                       CheckerContext &C) const {
  // Prototype for fclose is
  //   int fclose(FILE *FileDesc);
  if (!Call.isGlobalCFunction("fclose") || Call.getNumArgs() != 1)
    return;

  // Get the symbolic value corresponding to the file handle.
  SymbolRef FileDesc = Call.getArgSVal(0).getAsSymbol();
  if (!FileDesc)
    return;

  // Generate the next transition, in which the stream is closed.
  ProgramStateRef State = C.getState();
  State = State->set<StreamMap>(FileDesc, StreamState::getClosed());
  C.addTransition(State);
}
```

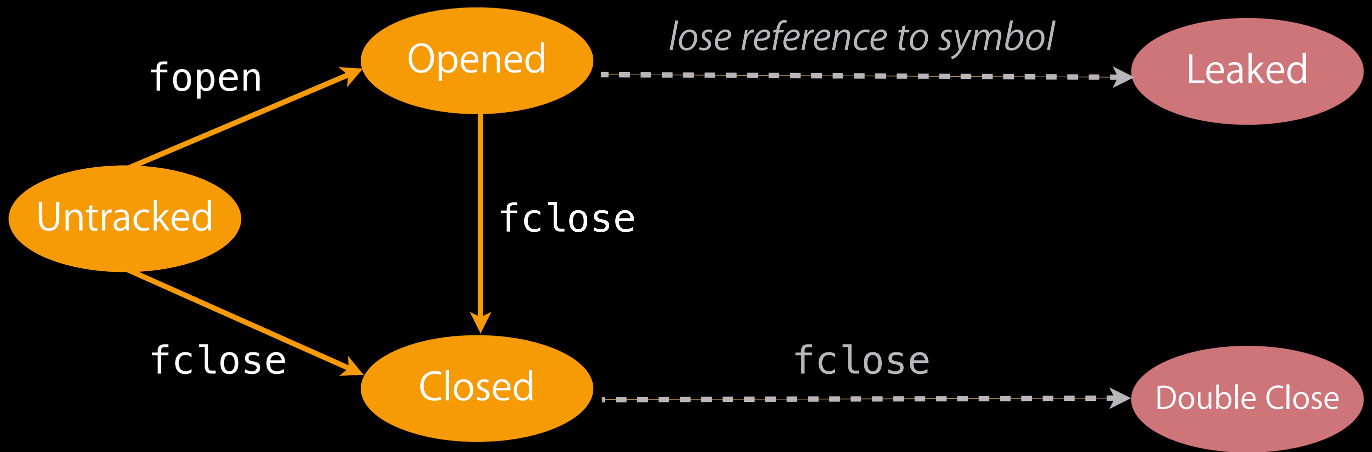
Stream Checker Recipe

✓ Define the state of a file descriptor

✓ Add state transition corresponding to `fopen`

✓ Add transitions driven by `fclose`

- Report error on double close
- Report error on leak



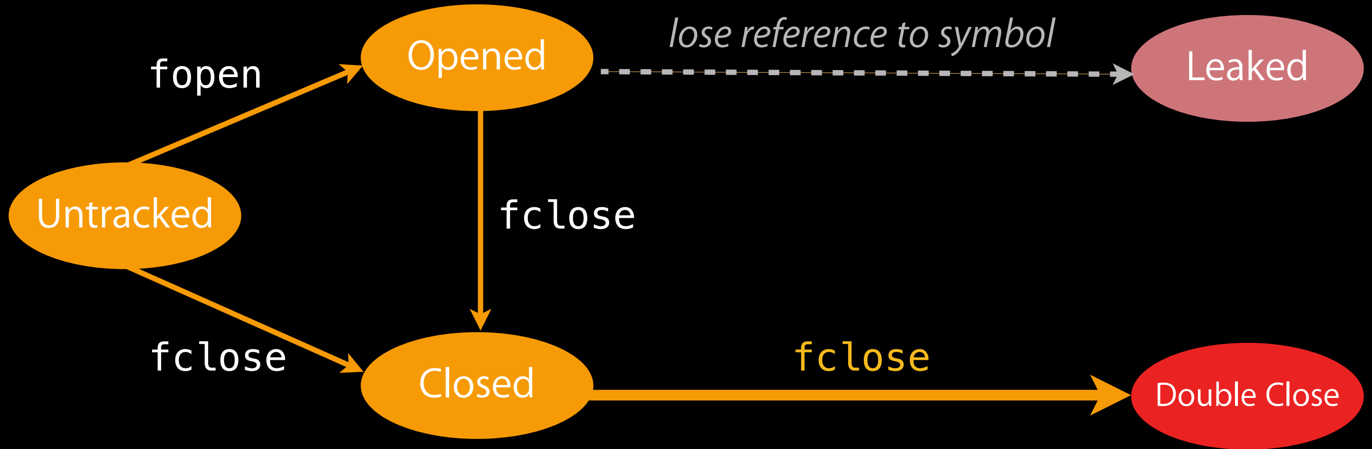
Stream Checker Recipe

✓ Define the state of a file descriptor

✓ Add state transition corresponding to `fopen`

✓ Add transitions driven by `fclose`

- Report error on double close
- Report error on leak



Report Double Close

```
void SimpleStreamChecker::checkPreCall(const CallEvent &Call,
                                       CheckerContext &C) const {
  // Prototype for fclose is
  //   int fclose(FILE *FileDesc);
  if (!Call.isGlobalCFunction("fclose") || Call.getNumArgs() != 1)
    return;

  // Get the symbolic value corresponding to the file handle.
  SymbolRef FileDesc = Call.getArgSVal(0).getAsSymbol();
  if (!FileDesc)
    return;

  // Generate the next transition, in which the stream is closed.
  ProgramStateRef State = C.getState();
  State = State->set<StreamMap>(FileDesc, StreamState::getClosed());
  C.addTransition(State);
}
```

Report Double Close

```
void SimpleStreamChecker::checkPreCall(const CallEvent &Call,
                                       CheckerContext &C) const {
  // ...


  // Check if the stream has already been closed.
  const StreamState *SS = C.getState()->get<StreamMap>(FileDesc);
  if (SS && SS->isClosed()) {
    reportDoubleClose(FileDesc, Call, C);
    return;
  }

  // Generate the next transition, in which the stream is closed.
  ProgramStateRef State = C.getState();
  State = State->set<StreamMap>(FileDesc, StreamState::getClosed());
  C.addTransition(State);
}
```

Generating a BugReport

```
void SimpleStreamChecker::reportDoubleClose(SymbolRef FileDescSym,
                                            const CallEvent &Call,
                                            CheckerContext &C) const {
  // We reached a bug, stop exploring the path here by generating a sink.
  ExplodedNode *ErrNode = C.generateSink();









}
```

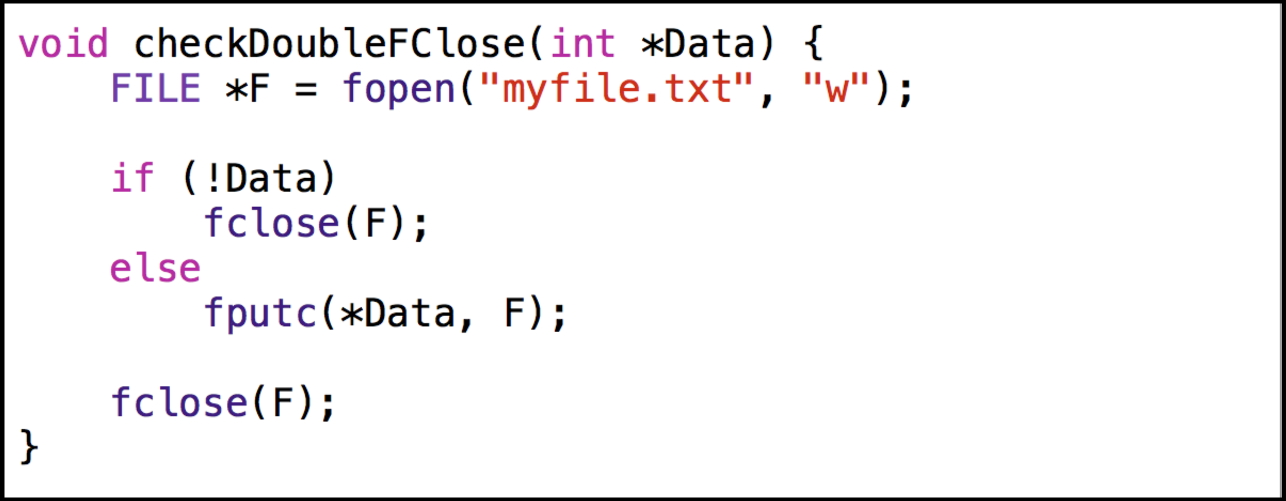
Generating a BugReport

```
void SimpleStreamChecker::reportDoubleClose(SymbolRef FileDescSym,
                                            const CallEvent &Call,
                                            CheckerContext &C) const {
  // We reached a bug, stop exploring the path here by generating a sink.
  ExplodedNode *ErrNode = C.generateSink();

  // If we've already reached this node on another path, return.
  if (!ErrNode)
    return;






}
```

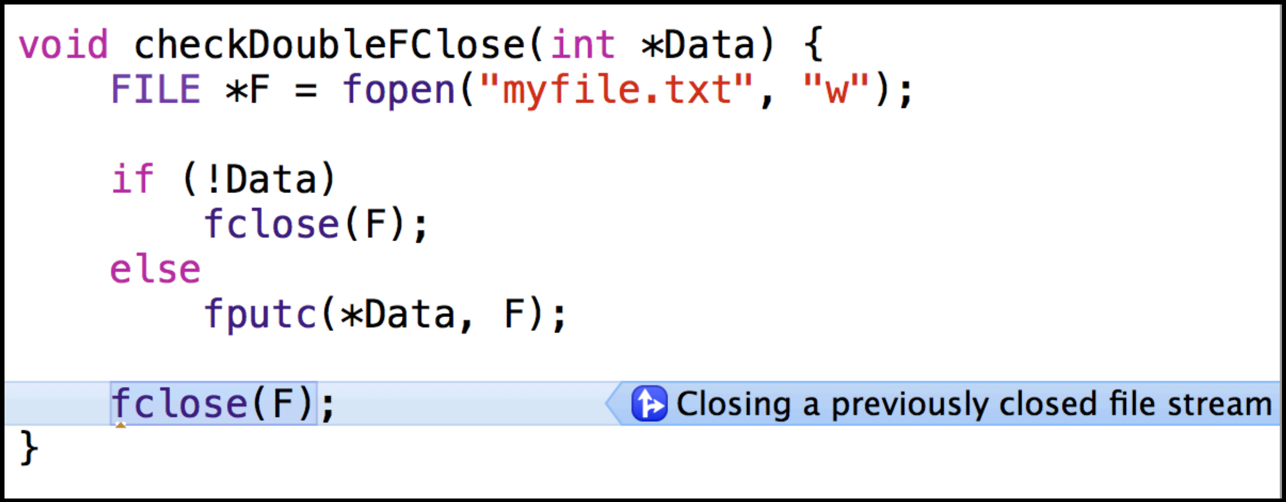
Generating a BugReport

```
void SimpleStreamChecker::reportDoubleClose(SymbolRef FileDescSym,
                                            const CallEvent &Call,
                                            CheckerContext &C) const {
  // We reached a bug, stop exploring the path here by generating a sink.
  ExplodedNode *ErrNode = C.generateSink();

  // If we've already reached this node on another path, return.
  if (!ErrNode)
    return;

  // Generate the report.
  BugReport *R = new BugReport(*DoubleCloseBugType,
      "Closing a previously closed file stream", ErrNode);
  R->addRange(Call.getSourceRange());
  R->markInteresting(FileDescSym);
  C.emitReport(R);
}
```

Test Double Close

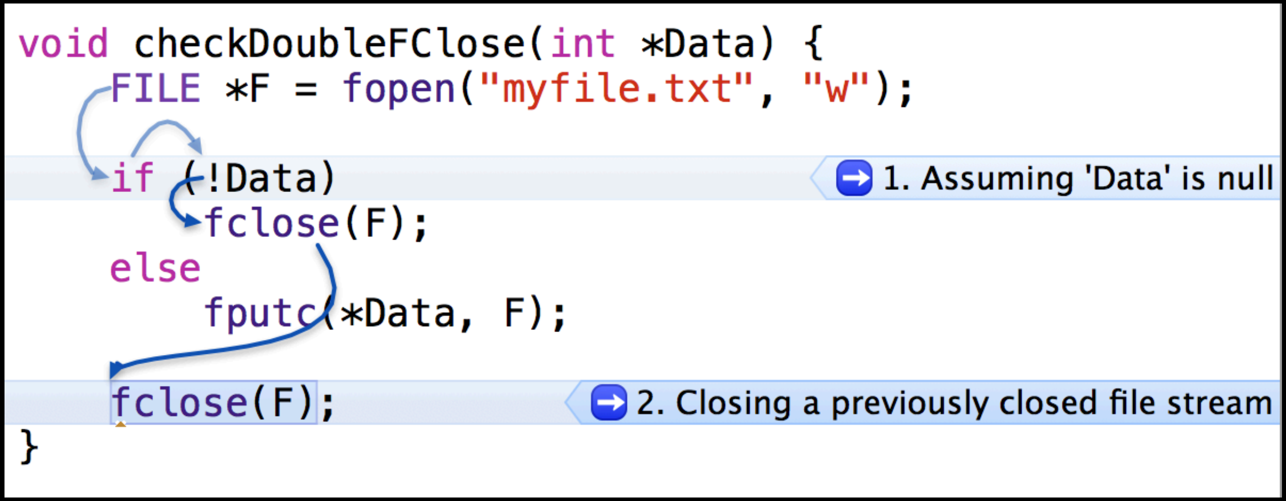

```
void checkDoubleFClose(int *Data) {
    FILE *F = fopen("myfile.txt", "w");

    if (!Data)
        fclose(F);
    else
        fputc(*Data, F);

    fclose(F);
}
```

Test Double Close



```
void checkDoubleFClose(int *Data) {
    FILE *F = fopen("myfile.txt", "w");

    if (!Data)
        fclose(F);
    else
        fputc(*Data, F);

    fclose(F);                          Closing a previously closed file stream
}
```

Test Double Close



```
void checkDoubleFClose(int *Data) {
    FILE *F = fopen("myfile.txt", "w");

    if (!Data)
        fclose(F);
    else
        fputc(*Data, F);

    fclose(F);
}
```

1. Assuming 'Data' is null

2. Closing a previously closed file stream

Stream Checker Recipe

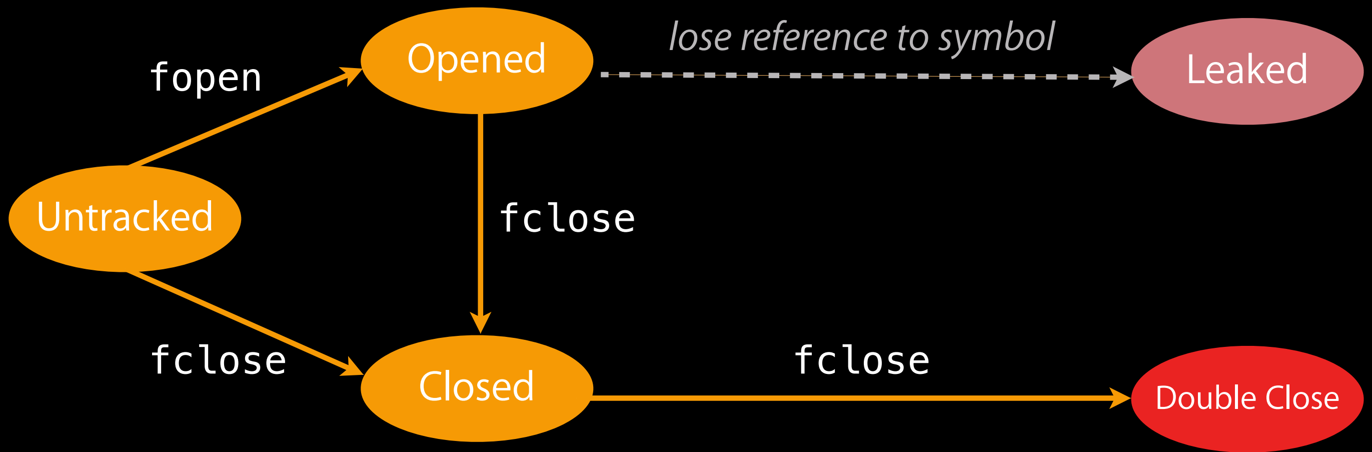
✓ Define the state of a file descriptor

✓ Add state transition corresponding to `fopen`

✓ Add transitions driven by `fclose`

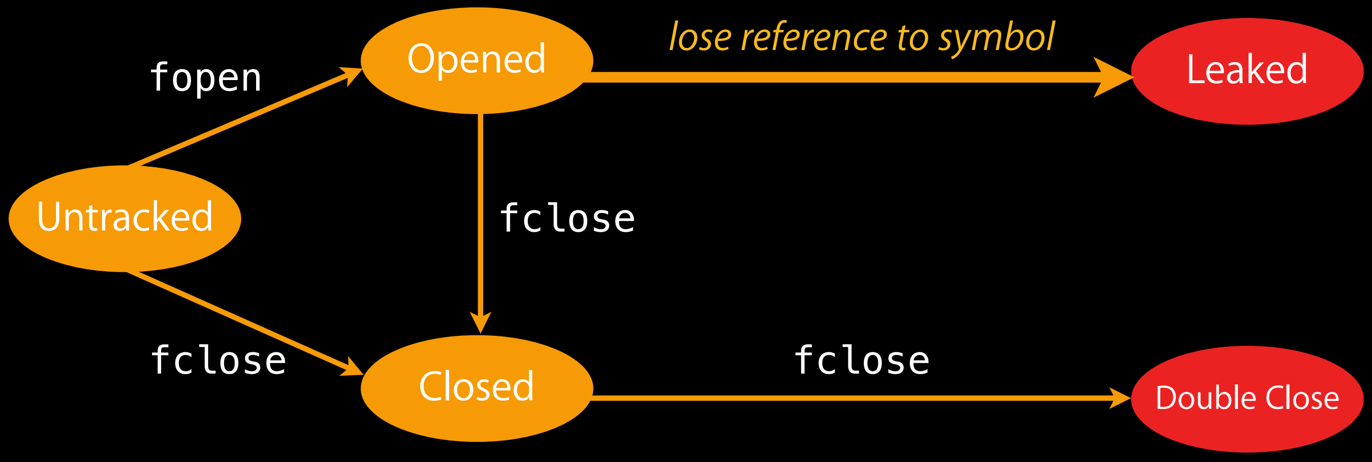
✓ Report error on double close

- Report error on leak



Stream Checker Recipe

- ✓ Define the state of a file descriptor
- ✓ Add state transition corresponding to `fopen`
- ✓ Add transitions driven by `fclose`
- ✓ Report error on double close
- Report error on leak



Register for Dead Symbols

- A “dead” symbol can never be referenced again along this path
- Checkers can be notified when symbols die

```
class SimpleStreamChecker : public Checker<check::PostCall,
                                           check::PreCall,
                                           check::DeadSymbols > {
  ...
  void checkDeadSymbols(SymbolReaper &SymReaper, CheckerContext &C) const;
};
```

Collect and Report Leaks

```
void SimpleStreamChecker::checkDeadSymbols(SymbolReaper &SymReaper,
                                           CheckerContext &C) const {
  ProgramStateRef State = C.getState();
  SymbolVector LeakedStreams;









}
```

Collect and Report Leaks

```
void SimpleStreamChecker::checkDeadSymbols(SymbolReaper &SymReaper,
                                           CheckerContext &C) const {
  ProgramStateRef State = C.getState();
  SymbolVector LeakedStreams;
  StreamMapTy TrackedStreams = State->get<StreamMap>();
  for (StreamMapTy::iterator I = TrackedStreams.begin(),
                             E = TrackedStreams.end(); I != E; ++I) {
    SymbolRef Sym = I->first;








  }


}
```

Collect and Report Leaks

```
void SimpleStreamChecker::checkDeadSymbols(SymbolReaper &SymReaper,
                                           CheckerContext &C) const {
  ProgramStateRef State = C.getState();
  SymbolVector LeakedStreams;
  StreamMapTy TrackedStreams = State->get<StreamMap>();
  for (StreamMapTy::iterator I = TrackedStreams.begin(),
                             E = TrackedStreams.end(); I != E; ++I) {
    SymbolRef Sym = I->first;
    bool IsSymDead = SymReaper.isDead(Sym);






  }


}
```

Collect and Report Leaks

```
void SimpleStreamChecker::checkDeadSymbols(SymbolReaper &SymReaper,
                                           CheckerContext &C) const {
  ProgramStateRef State = C.getState();
  SymbolVector LeakedStreams;
  StreamMapTy TrackedStreams = State->get<StreamMap>();
  for (StreamMapTy::iterator I = TrackedStreams.begin(),
                             E = TrackedStreams.end(); I != E; ++I) {
    SymbolRef Sym = I->first;
    bool IsSymDead = SymReaper.isDead(Sym);

    if (isLeaked(Sym, I->second, IsSymDead))
      LeakedStreams.push_back(Sym);



  }


}
```

Collect and Report Leaks

```
static bool isLeaked(SymbolRef Sym, const StreamState &SS,
                     bool IsSymDead) {
  if (IsSymDead && SS.isOpened()) {
    return true;
  }
  return false;
}
```

- Future expressions cannot refer to a dead symbol
- If a dead stream is still open, it’s a leak!

Collect and Report Leaks

```
void SimpleStreamChecker::checkDeadSymbols(SymbolReaper &SymReaper,
                                           CheckerContext &C) const {
  ProgramStateRef State = C.getState();
  SymbolVector LeakedStreams;
  StreamMapTy TrackedStreams = State->get<StreamMap>();
  for (StreamMapTy::iterator I = TrackedStreams.begin(),
                             E = TrackedStreams.end(); I != E; ++I) {
    SymbolRef Sym = I->first;
    bool IsSymDead = SymReaper.isDead(Sym);

    if (isLeaked(Sym, I->second, IsSymDead))
      LeakedStreams.push_back(Sym);



  }


}
```

Collect and Report Leaks

```
void SimpleStreamChecker::checkDeadSymbols(SymbolReaper &SymReaper,
                                           CheckerContext &C) const {
  ProgramStateRef State = C.getState();
  SymbolVector LeakedStreams;
  StreamMapTy TrackedStreams = State->get<StreamMap>();
  for (StreamMapTy::iterator I = TrackedStreams.begin(),
                             E = TrackedStreams.end(); I != E; ++I) {
    SymbolRef Sym = I->first;
    bool IsSymDead = SymReaper.isDead(Sym);

    if (isLeaked(Sym, I->second, IsSymDead))
      LeakedStreams.push_back(Sym);



  }
  ExplodedNode *N = C.addTransition(State);
  reportLeaks(LeakedStreams, C, N);
}
```

- Don’t create a sink node to keep exploring the given path

Don’t Forget to Clean Out the State

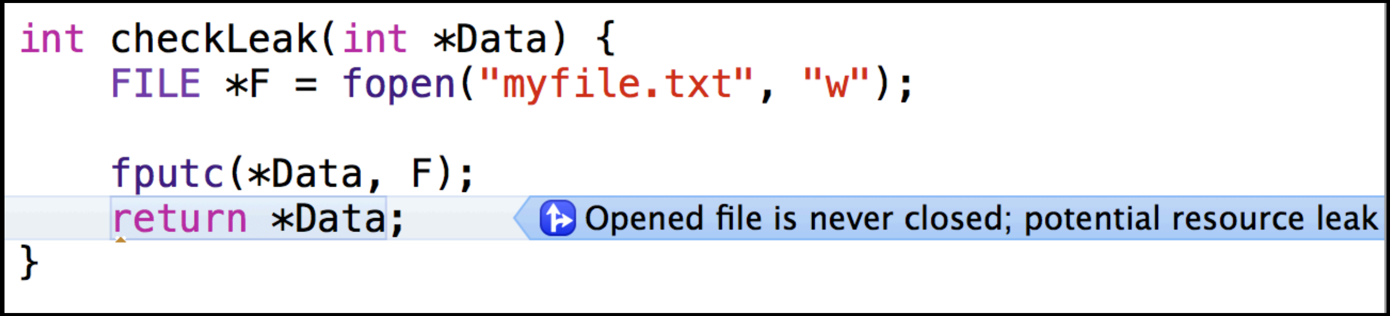
```
void SimpleStreamChecker::checkDeadSymbols(SymbolReaper &SymReaper,
                                           CheckerContext &C) const {
  ProgramStateRef State = C.getState();
  SymbolVector LeakedStreams;
  StreamMapTy TrackedStreams = State->get<StreamMap>();
  for (StreamMapTy::iterator I = TrackedStreams.begin(),
                             E = TrackedStreams.end(); I != E; ++I) {
    SymbolRef Sym = I->first;
    bool IsSymDead = SymReaper.isDead(Sym);

    if (isLeaked(Sym, I->second, IsSymDead))
      LeakedStreams.push_back(Sym);

    if (IsSymDead)
      State = State->remove<StreamMap>(Sym);
  }
  ExplodedNode *N = C.addTransition(State);
  reportLeaks(LeakedStreams, C, N);
}
```

- Don’t create a sink node to keep exploring the given path
- Will never refer to these symbols again, so keep `ProgramState` lean

Test Leak

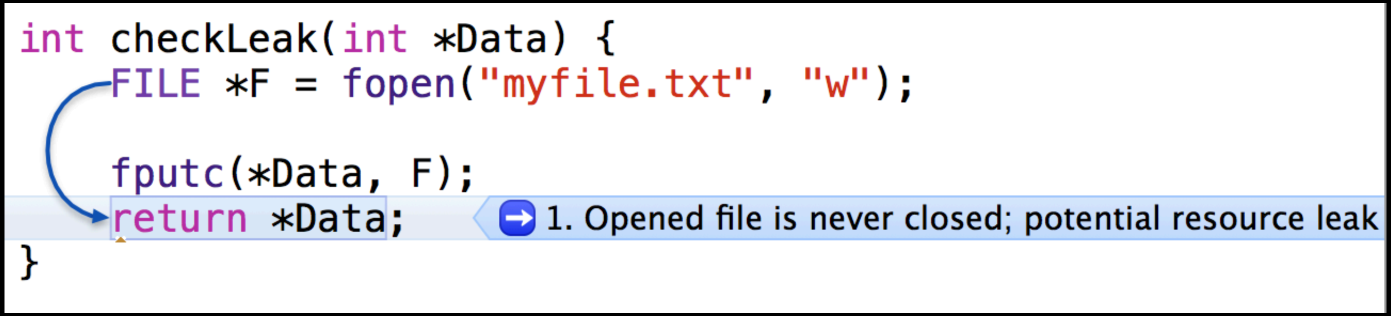
```
int checkLeak(int *Data) {
    FILE *F = fopen("myfile.txt", "w");

    fputc(*Data, F);
    return *Data;
}
```

Test Leak



```
int checkLeak(int *Data) {
    FILE *F = fopen("myfile.txt", "w");

    fputc(*Data, F);
    return *Data;          Opened file is never closed; potential resource leak
}
```

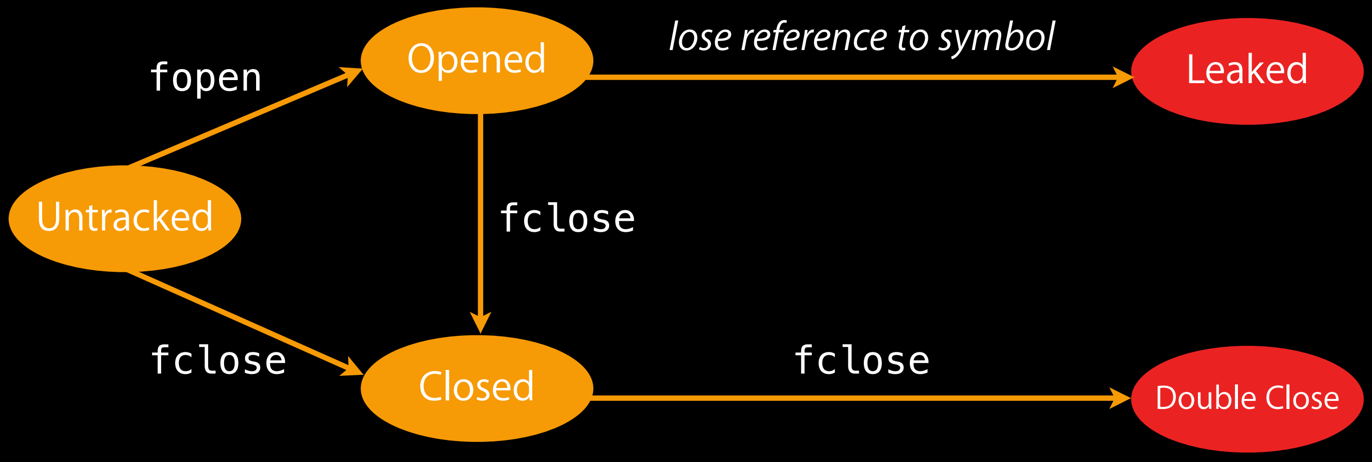
Test Leak



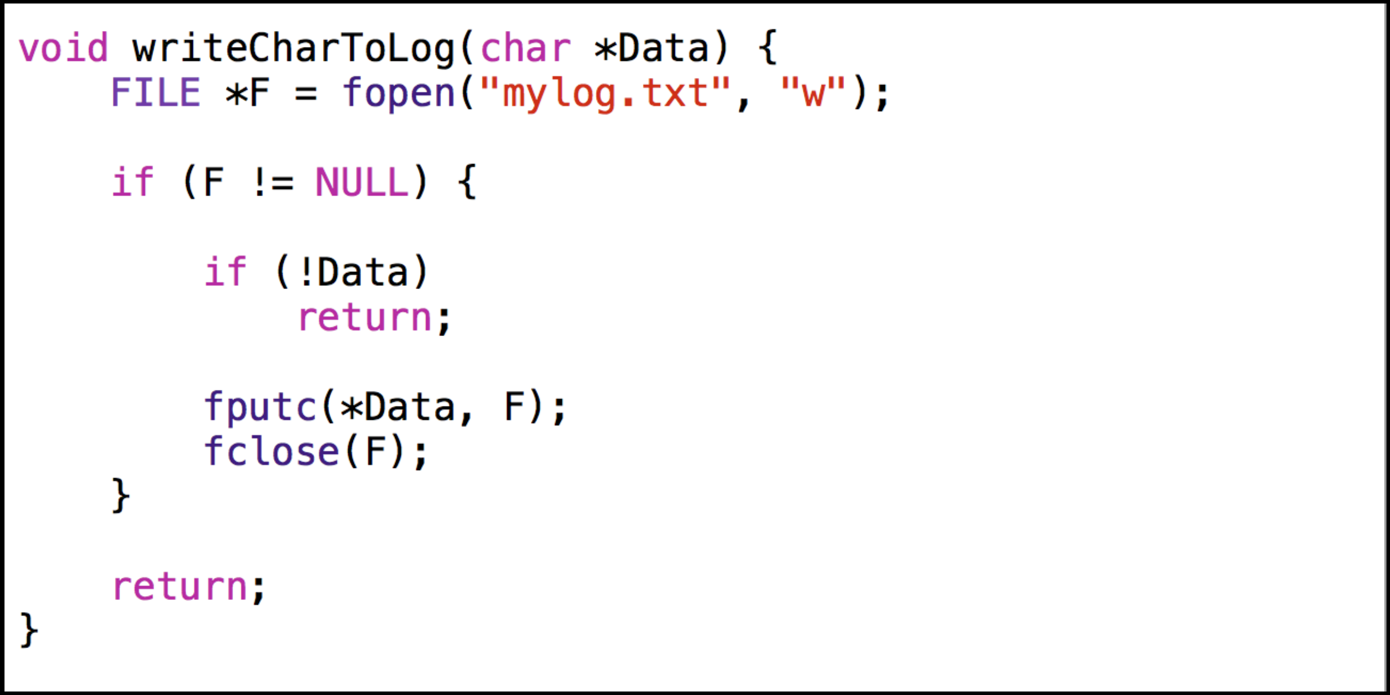
```
int checkLeak(int *Data) {
    FILE *F = fopen("myfile.txt", "w");

    fputc(*Data, F);
    return *Data;        1. Opened file is never closed; potential resource leak
}
```

Stream Checker Recipe

- ✓Define the state of a file descriptor
- ✓Add state transition corresponding to `fopen`
- ✓Add transitions driven by `fclose`
- ✓Report error on double close
- ✓Report error on leak

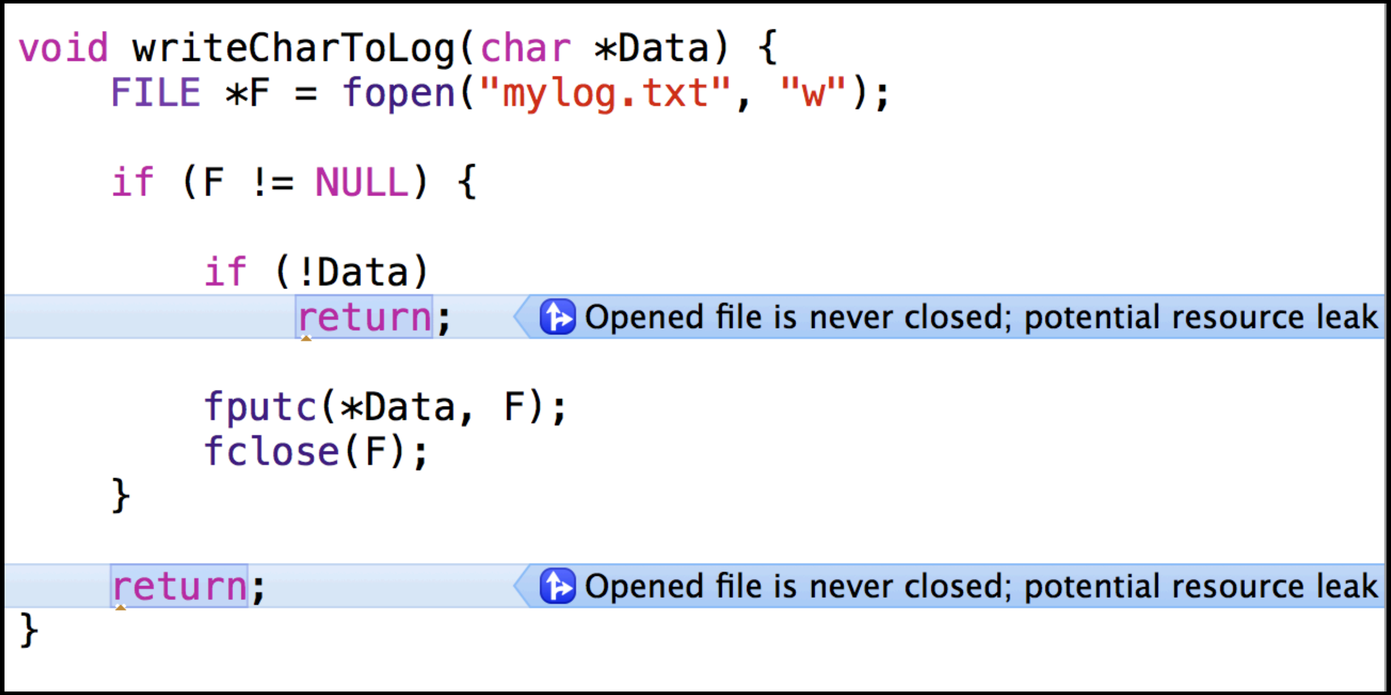


Let’s Use the Intro Testcase



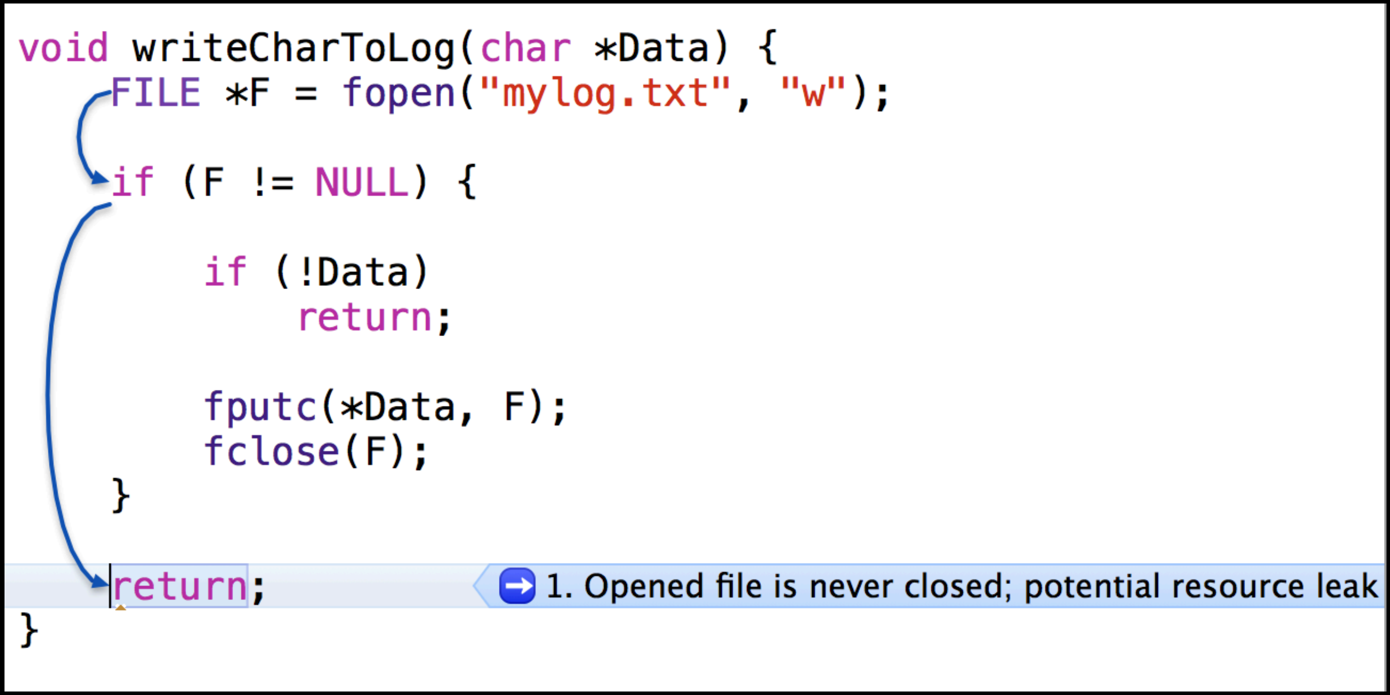
```c
void writeCharToLog(char *Data) {
    FILE *F = fopen("mylog.txt", "w");

    if (F != NULL) {

        if (!Data)
            return;

        fputc(*Data, F);
        fclose(F);
    }

    return;
}
```

Ooops...



```
void writeCharToLog(char *Data) {
    FILE *F = fopen("mylog.txt", "w");

    if (F != NULL) {

        if (!Data)
            return;        Opened file is never closed; potential resource leak

        fputc(*Data, F);
        fclose(F);
    }

    return;                Opened file is never closed; potential resource leak
}
```

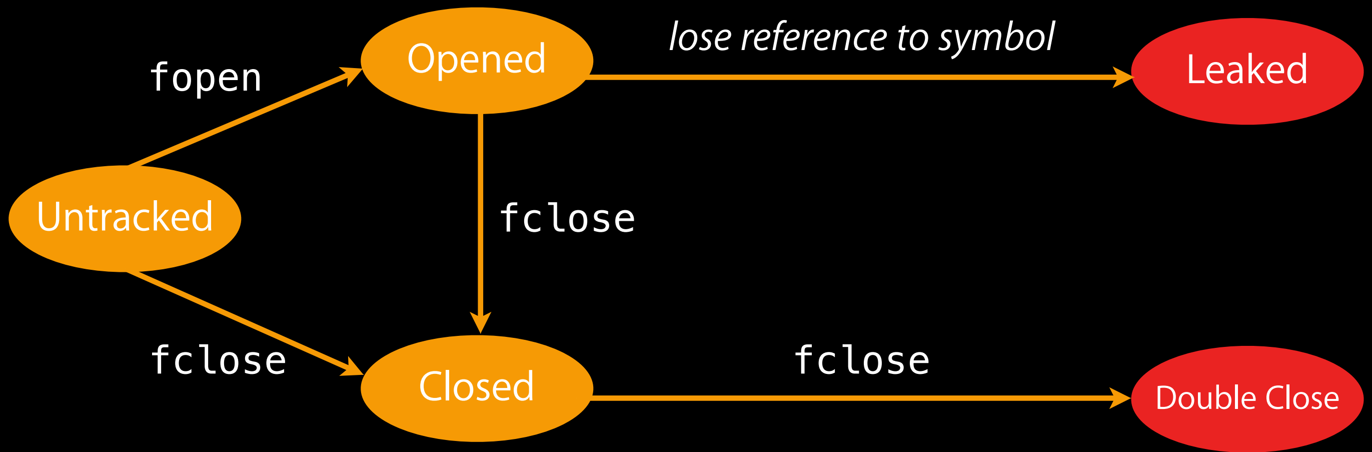
Moral: Test Well!



```
void writeCharToLog(char *Data) {
    FILE *F = fopen("mylog.txt", "w");

    if (F != NULL) {

        if (!Data)
            return;

        fputc(*Data, F);
        fclose(F);
    }

    return;          1. Opened file is never closed; potential resource leak
}
```

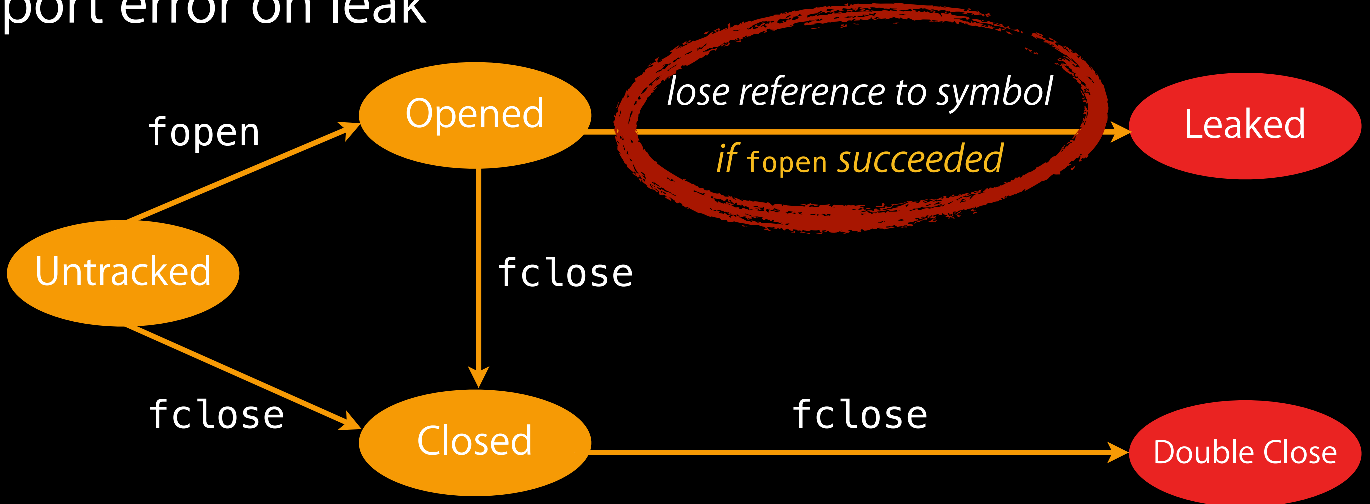
Stream Checker Recipe

- ✓ Define the state of a file descriptor
- ✓ Add state transition corresponding to `fopen`
- ✓ Add transitions driven by `fclose`
- ✓ Report error on double close
- ✓ Report error on leak



Stream Checker Recipe

- ✓ Define the state of a file descriptor
- ✓ Add state transition corresponding to `fopen`
- ✓ Add transitions driven by `fclose`
- ✓ Report error on double close
- ✓ Report error on leak



Don’t Warn If `fopen` Fails

- Why don’t we just register for `PreStmt<IfStmt>`?

```
if (F != 0) {
  fclose(F);
}
```

```
if (F == 0) {
  ;
} else {
  fclose(F);
}
```

```
while (F != 0) {
  fclose(F);
  break;
}
```

```
FILE *C = F;
if (C != 0) {
  fclose(F);
}
```

- Need to know if the file handle is **constrained** to `NULL`

Let’s Refine the Leak Definition

```
static bool isLeaked(SymbolRef Sym, const StreamState &SS,
                     bool IsSymDead) {
  if (IsSymDead && SS.isOpened()) {
    return true;
  }
  return false;
}
```

Let’s Refine the Leak Definition

```
static bool isLeaked(SymbolRef Sym, const StreamState &SS,
                     bool IsSymDead, ProgramStateRef State) {
  if (IsSymDead && SS.isOpened()) {
    // If a symbol is NULL, assume that fopen failed on this path.
    // A symbol should only be considered leaked if it is non-null.
    ConstraintManager &CMgr = State->getConstraintManager();


  }
  return false;
}
```

Let’s Refine the Leak Definition

```
static bool isLeaked(SymbolRef Sym, const StreamState &SS,
                     bool IsSymDead, ProgramStateRef State) {
  if (IsSymDead && SS.isOpened()) {
    // If a symbol is NULL, assume that fopen failed on this path.
    // A symbol should only be considered leaked if it is non-null.
    ConstraintManager &CMgr = State->getConstraintManager();
    ConditionTruthVal OpenFailed = CMgr.isNull(State, Sym);

  }
  return false;
}
```

Let's Refine the Leak Definition

```
static bool isLeaked(SymbolRef Sym, const StreamState &SS,
                     bool IsSymDead, ProgramStateRef State) {
  if (IsSymDead && SS.isOpened()) {
    // If a symbol is NULL, assume that fopen failed on this path.
    // A symbol should only be considered leaked if it is non-null.
    ConstraintManager &CMgr = State->getConstraintManager();
    ConditionTruthVal OpenFailed = CMgr.isNull(State, Sym);

  }
  return false;
}
```

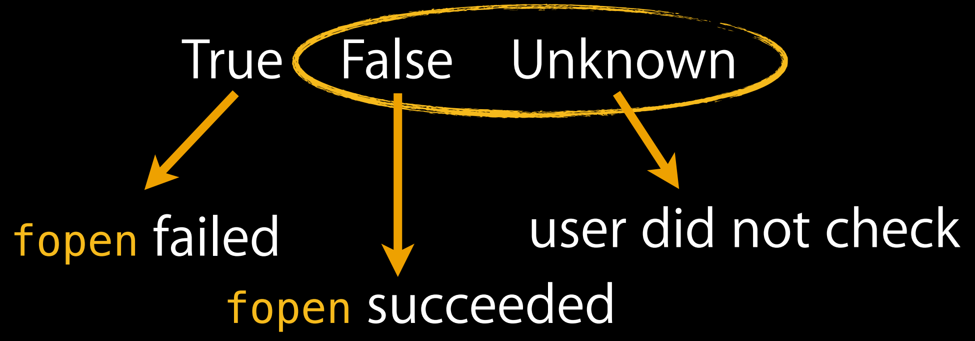
- `ConditionTruthVal` is a tri-state:

$$\begin{tikzcd}
 & \text{True} \arrow[dl] & \text{False} \arrow[dd] & \text{Unknown} \arrow[dr] & \\
\text{fopen failed} & & & & \text{user did not check} \\
 & & \text{fopen succeeded} & &
\end{tikzcd}$$

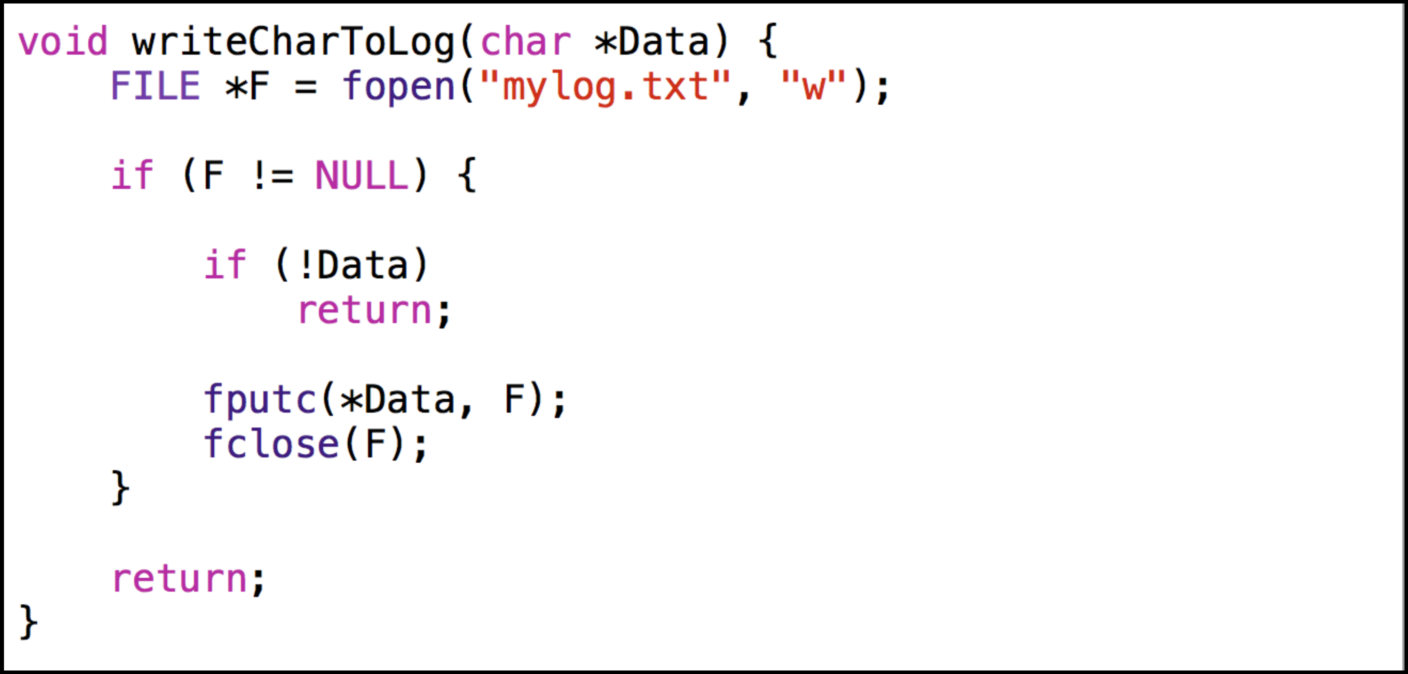
Let’s Refine the Leak Definition

```
static bool isLeaked(SymbolRef Sym, const StreamState &SS,
                     bool IsSymDead, ProgramStateRef State) {
  if (IsSymDead && SS.isOpened()) {
    // If a symbol is NULL, assume that fopen failed on this path.
    // A symbol should only be considered leaked if it is non-null.
    ConstraintManager &CMgr = State->getConstraintManager();
    ConditionTruthVal OpenFailed = CMgr.isNull(State, Sym);
    return !OpenFailed.isConstrainedTrue();
  }
  return false;
}
```

- `ConditionTruthVal` is a tri-state:

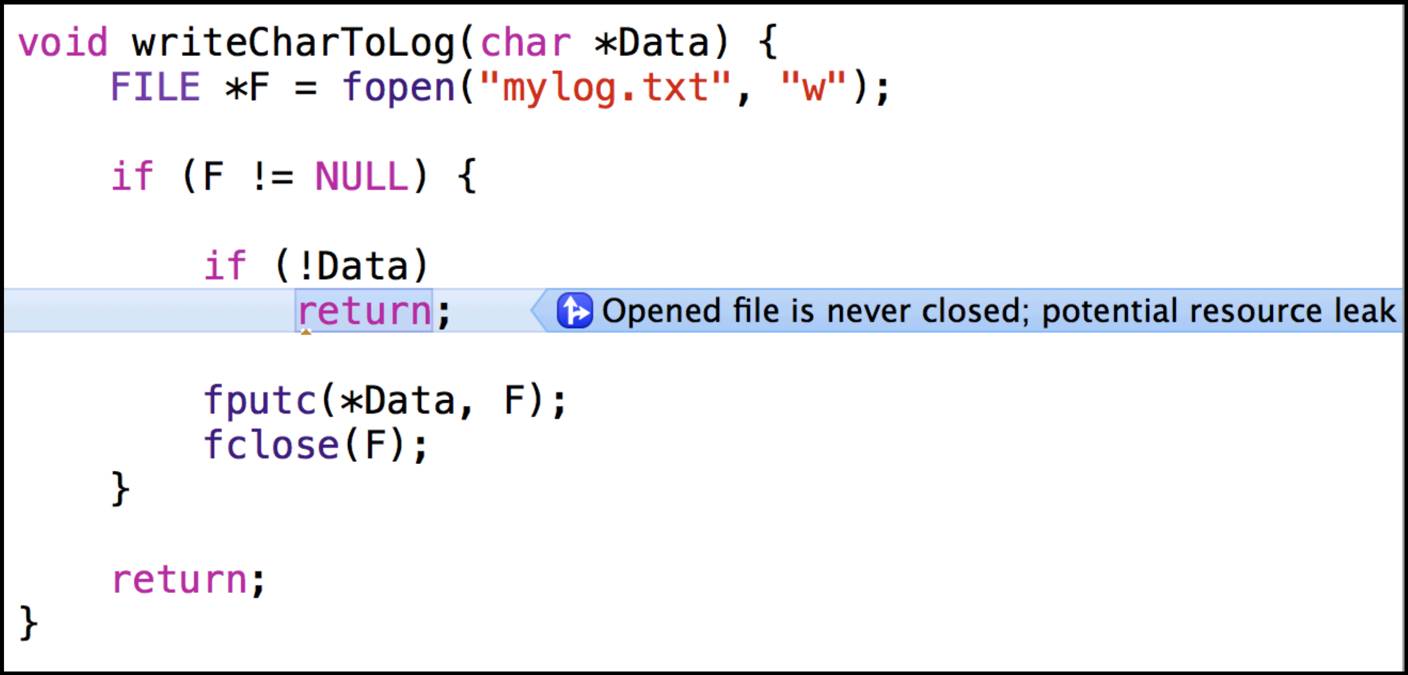


Let’s See if the False Positive is Gone



```
void writeCharToLog(char *Data) {
    FILE *F = fopen("mylog.txt", "w");

    if (F != NULL) {

        if (!Data)
            return;

        fputc(*Data, F);
        fclose(F);
    }

    return;
}
```

We did it!



```
void writeCharToLog(char *Data) {
    FILE *F = fopen("mylog.txt", "w");

    if (F != NULL) {

        if (!Data)
            return;       Opened file is never closed; potential resource leak

        fputc(*Data, F);
        fclose(F);
    }

    return;
}
```

Homework

- Checker registration
- Improving diagnostics
- Relinquishing ownership
- Writing more regression tests

- The checker we wrote today is available at clang/lib/StaticAnalyzer/Checkers/**SimpleStreamChecker.cpp**
- The tests for the checker can be found at clang/tests/Analysis/**simple-stream-checker.c**

Current Limitations

- Constraint solver is limited
  - Bitwise operations (`$F & 0x10`)
  - Constraints involving multiple symbols (`$X > $Y`)
- Analysis is inter-procedural, but not (yet) cross-translation-unit
- The analyzer is only as good as its checkers!
  - http://clang-analyzer.llvm.org/potential_checkers.html
  - Patches welcome :-)

Summary

- The analyzer performs a symbolic, path-sensitive execution of a program
- Extendable with custom checks
- Provides comprehensive diagnostics
- Both plist (Xcode) and HTML output formats are available
- It is possible to write syntactic (AST-based) checkers as well

- For more info go to http://clang-analyzer.llvm.org/
- Send your questions to **cfe-dev** mailing list

Clang Static Analyzer Hackathon

[redacted]