Introduction to clang tools scan-build and clang-tidy.

Jan 5th, 2017

Intro to clang tools

- Scan-build – static analyzer.
- Clang-tidy – static analyzer and more.

Intro to clang tools: scan-build

- Scan-build is a command-line utility that runs a static analyzer.
- Web-interface for output.
- Part of clang.
- Runs slower than compilation.

Intro to clang tools: scan-build

Static code analyzer tries to find bugs in a program without executing it. This is opposite to sanitazers and dynamic code analysis tools, such as valgrind, that finds bugs in a program when executing it. The downside of dynamic code analysis tools is that the whole program must be executed which can be time-consuming.

Intro to clang tools: scan-build

- It works during a project build, as source files are compiled they are also analyzed in tandem by the static analyzer.
- It has no knowledge of a build so it works by overriding CC or CXX variables with a “fake” compiler, hence the makefile should make use of CC or CXX:
- $ export CXX=clang++
- It's recommended to run it in “debug” configuration.

Intro to clang tools: scan-build

- It can be run by supplying files:

$ scan-build -v -o . clang++ -g -O0 -std=c++14 ttmain.cpp ttlib.cpp

- It can be run with make file:

$ scan-build -v -o . make

Intro to clang tools: scan-build

Run example.

Intro to clang tools: scan-build

test1 - scan-build results

- User: khavrych@qipk2dyy8
- Working Directory: /home/khavrych/test1
- Command Line: make
- Clang Version: Debian clang version 3.5.2-3~bpo8+2 (tags/RELEASE_352/final) (based on LLVM 3.5.2)
- Date: Thu Jan 5 15:30:48 2017

Intro to clang tools: clang-tidy

- clang-tidy is a clang-based C++ “linter” tool.
- It runs Clang static analysis but it also does it's own checks.
- clang-tidy is a Clang LibTooling-based tool.
- You can add your own checks!

Intro to clang tools: clang-tidy

- boost- Checks related to Boost library.
- cert- Checks related to CERT Secure Coding Guidelines.
- cppcoreguidelines- Checks related to C++ Core Guidelines.
- clang-analyzer- Clang Static Analyzer checks.
- google- Checks related to the Google coding conventions.
- llvm- Checks related to the LLVM coding conventions.
- misc- Checks that we didn't have a better category for.

Intro to clang tools: clang-tidy

- modernize- Checks that advocate usage of modern (currently “modern” means “C++11”) language constructs.
- mpi- Checks related to MPI (Message Passing Interface).
- performance- Checks that target performance-related issues.
- readability- Checks that target readability-related issues that don’t relate to any particular coding style.

Intro to clang tools: clang-tidy

- It can be run by supplying files:

$ clang-tidy ttmain.cpp ttlib.cpp -checks=*,-clang-analyzer-alpha.*,-clang-analyzer-osx.* -- -g -O0 -std=c++14

- It can be run using a compilation database.
- You can see checks with “-list-checks”

$ clang-tidy ttmain.cpp -list-checks -checks=* -- -std=c++14

Intro to clang tools: clang-tidy

Run example.

Intro to clang tools: source code

Makefile

```
CXXFLAGS := -g -O0 -std=c++14
LDFLAGS :=
LDLIBS :=
CXXWARN :=\
	-W'all'\
	-W'extra'\
	-W'unused'\
	-W'pedantic'\
	-W'shadow'\
	-W'missing-declarations'\
	-W'inline'\
	-W'write-strings'\
	-W'conversion'\
	-W'overflow'\
	-W'strict-overflow=5'\
```

Intro to clang tools: source code

```
        -W'format-security'\
        -W'format-nonliteral'\

all:
                $(CXX) $(CXXFLAGS) $(CXXWARN) ttmain.cpp ttlib.cpp -o
ttmain

.PHONY: clean
clean:
                        rm -f *.o ttmain a.out *.plist
```

Intro to clang tools: source code

```
// ttlib.cpp
#include <iostream>
#include <string>

/**
 * Prints first argument to cout.
 *
 * @param[in] str1 is a string to print to stdout.
 * @param[in] str2 is a string that is not used.
 */
void print_message(std::string str1, std::string str2);

/**
 * Static function that is never used.
 */
static void never_used(void);
```

Intro to clang tools: source code

```
void print_message(std::string str1, std::string str2)
{
   std::cout << "Message: \"" << str1 << "\"\n";
}

static void never_used(void)
{
   std::cout << "This function is never used\n";
}
```

Intro to clang tools: source code

```
// ttmain.cpp

#include <iostream>
#include <cstdlib>

class A
{
public:
    A(void) {i = 99;}

private:
    int i = 0;
};
```

Intro to clang tools: source code

```
class B
{
public:
    B(void) = delete;
    explicit B(A& p) : a(p) {}

private:
    A& a; // Should give a warning about using reference.
};

extern void print_message(std::string str1, std::string str2);
```

Intro to clang tools: source code

```
int main(void)
{
   std::cout << "Start.\n";

   int i = 99; // Var is alright.
   ++i;

   int i_not_used = 100; // Var is never used.

   int i_shadowed = 101; // Var shadowed.
   ++i_shadowed;
   {
      int i_shadowed = 102; // Var shadows.
      ++i_shadowed;
   }
```

Intro to clang tools: source code

```
print_message("Hello Alice!", "Hello Veronika!");

int *pi = new int;
*pi = 103;
std::cout << *pi << std::endl;
delete pi;
std::cout << *pi << std::endl; // Using deallocated object.

int i_dived_by_zero = 10/0; // Divide by zero.

printf("Numbers: %d and %f\n", 103); // xlc ignores this.

int arr[10] = {105};
std::cout << arr[0] << std::endl; // Prints "105".
std::cout << arr[10] << std::endl; // Index out of bound.
```

Intro to clang tools: source code

```
    A a;
    B b(a); //

    delete pi; // Double free.

    return EXIT_SUCCESS;
}
```