It’s not difficult, and it’s unbelievably useful

# Slide accessibility template

- Title and subtitle text
- Body text, including *italics* (no underline or bold)
- `Code and code-specific references`
- `Important/highlighted code`
- `Commands and file names`

Not all text will be the same size; feel free to move closer or ask me to zoom in or pause so you can read something.

# About Me

- Jonathan Smith,
  Principal Software Engineer @ FiveTwelve
- U.S. Navy veteran and former cyber operator
- Compiler and defensive software engineering for seven years
- Snowboarding, singing, failing to learn guitar
- All social contact info @ https://jvste.ch

## Why Windows?

- #beginners channel on the LLVM Discord
- Beginners category on the LLVM Discourse (https://discourse.llvm.org)
- Everyone must start somewhere, and most people start on Windows
- WSL[2] provides easy, first-class access to Linux

# Why Visual Studio? Why not VSCode?

- Visual Studio (community edition) is free and "batteries are included"
- Subjective: Visual Studio's debugger is easier to learn than GDB (or LLDB)
- I have never used VSCode for LLVM development

# Bootstrapping Clang and LLVM

CMake presets instead of toolchains // Ninja instead of MSBuild

# CMake Presets

https://cmake.org/cmake/help/latest/manual/cmake-presets.7.html

- JSON configuration files natively supported by both CMake and Visual Studio
- Less verbose than writing CMake cache or toolchain scripts
- Completely composable
- `CMakePresets.json` – project-provided
  `CMakeUserPresets.json` – user-specific

# Initial CMakeUserPresets.json

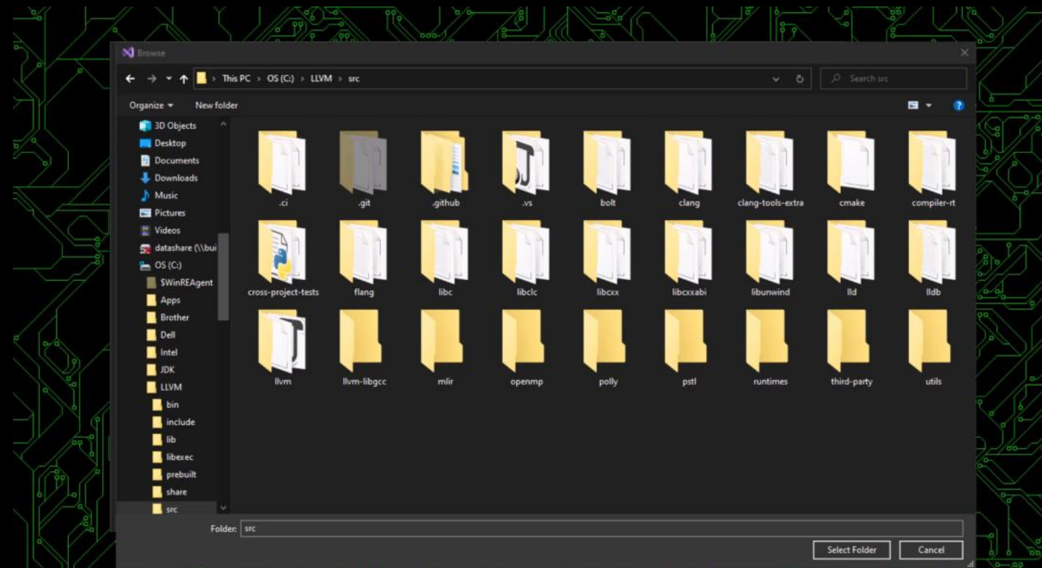
```
{
  "version": 5,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 23,
    "patch": 0
  },
  "configurePresets": [
    {
      "name": "default",
      "hidden": true,
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/build/${presetName}/build",
      "install Dir": "${sourceDir}/build/${presetName}/install"
    },
    {
      "name": "release",
      "inherits": "default",
      "hidden": true,
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Release"
      }
    },
    {
      "name": "windows-default",
      "inherits": "default",
      "hidden": true,
      "architecture": {
        "value": "x64",
        "strategy": "external"
      }
    },
    {
      "name": "windows-release",
      "inherits": [ "windows-default", "release" ],
      "hidden": true
    },
    {
      "name": "bootstrap",
      "inherits": "windows-release",
      "cacheVariables": {
        "LLVM_INCLUDE_BENCHMARKS": false,
        "LLVM_INCLUDE_EXAMPLES": false,
        "LLVM_INCLUDE_RUNTIMES": false,
        "LLVM_INCLUDE_TESTS": false,
        "LLVM_ENABLE_PROJECTS": "clang;lld",
        "LLVM_PARALLEL_LINK_JOBS": "1",
        "LLVM_TARGETS_TO_BUILD": "X86"
      }
    }
  ]
}
```

# Initial CMakeUserPresets.json (zoom)

```json
{
  "name": "bootstrap",
  "inherits": "windows-release",
  "cacheVariables": {
    "LLVM_INCLUDE_BENCHMARKS": false,
    "LLVM_INCLUDE_EXAMPLES": false,
    "LLVM_INCLUDE_RUNTIMES": false,
    "LLVM_INCLUDE_TESTS": false,
    "LLVM_ENABLE_PROJECTS": "clang;lld",
    "LLVM_PARALLEL_LINK_JOBS": "1",
    "LLVM_TARGETS_TO_BUILD": "X86"
}
```

# Building and installing

Link: https://youtu.be/T8zDXvBvaiU



Optimize for speed where it matters

# Key points for debug builds

- `CMAKE_BUILD_TYPE="Debug"` (not "RelWithDebInfo")
- Build llvm-tablegen *with* optimizations
- Disable debug iterator support in the Visual C++ run-time
- *DO NOT* use the release version of the Visual C++ run-time

# Base debug configuration

```json
{
  "name": "debug",
  "inherits": "default",
  "hidden": true,
  "cacheVariables": {
    "CMAKE_BUILD_TYPE": "Debug"
  }
}
```

# Windows-specific debug configuration

```json
{
  "name": "windows-debug",
  "inherits": [ "windows-default", "debug" ],
  "hidden": true,
  "cacheVariables": {
    "CMAKE_C_FLAGS_DEBUG": "-D_ITERATOR_DEBUG_LEVEL=0",
    "CMAKE_CXX_FLAGS_DEBUG": "-D_ITERATOR_DEBUG_LEVEL=0"
  }
}
```

# LLVM settings for both debug and release builds

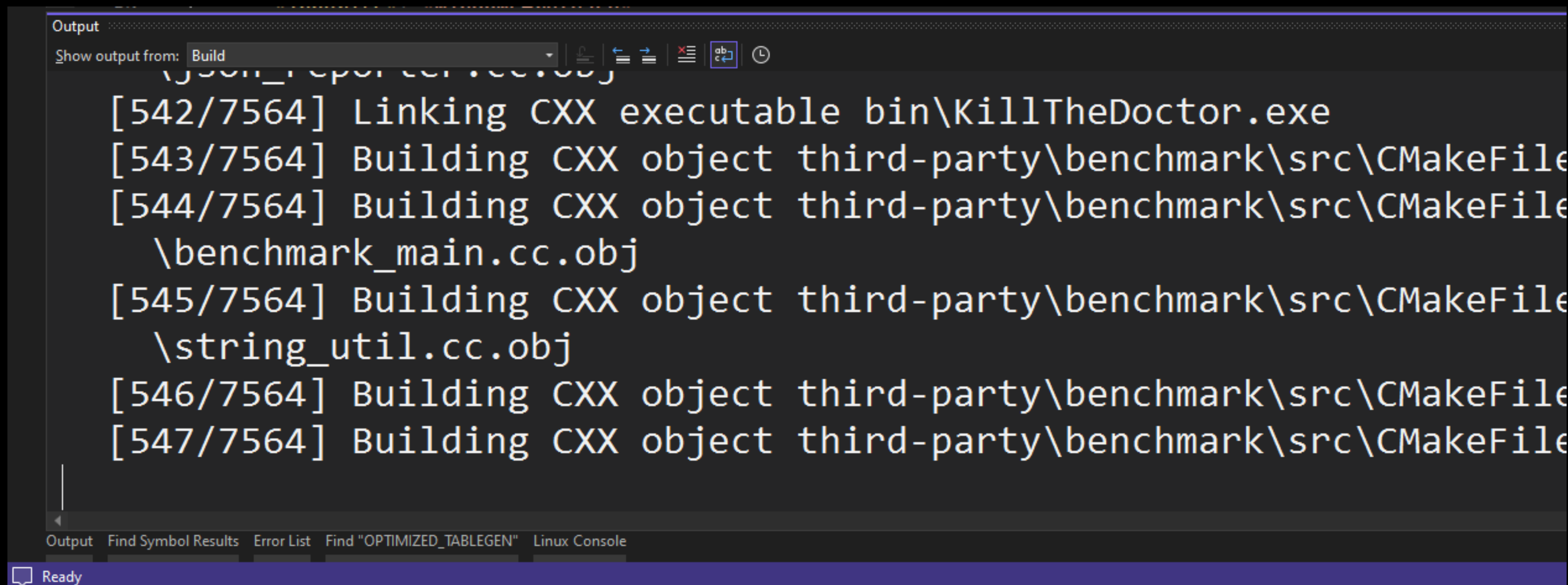
```
{
  "name": "common-llvm-settings",
  "hidden": true,
  "cacheVariables": {
    "CMAKE_C_COMPILER": "${sourceDir}/build/bootstrap/install/bin/clang-cl.exe",
    "CMAKE_CXX_COMPILER": "${sourceDir}/build/bootstrap/install/bin/clang-cl.exe",
    "CMAKE_EXPORT_COMPILE_COMMANDS": true,
    "LLVM_ENABLE_LLD": true,
    "LLVM_ENABLE_PROJECTS": "clang;clang-tools-extra;mlir;lld",
    "LLVM_PARALLEL_LINK_JOBS": "1"
  }
}
```

# LLVM-specific debug configuration

```
{
  "name": "llvm-debug",
  "inherits": [ "windows-debug", "common-llvm-settings" ],
  "cacheVariables": {
    "LLVM_OPTIMIZED_TABLEGEN": true
  }
}
```

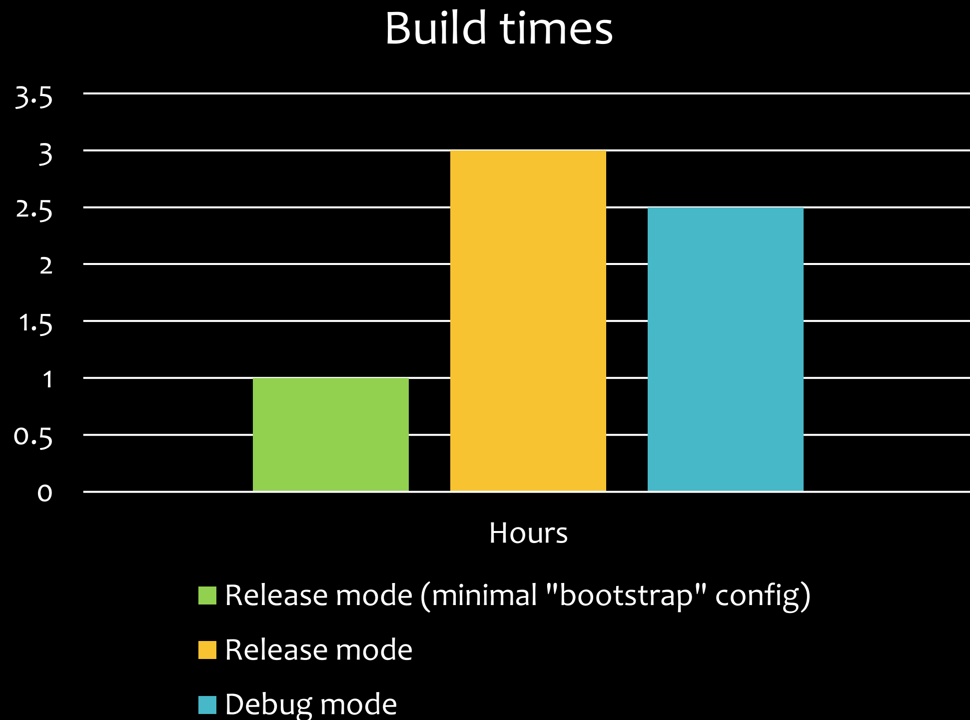
# Debug mode building and installing

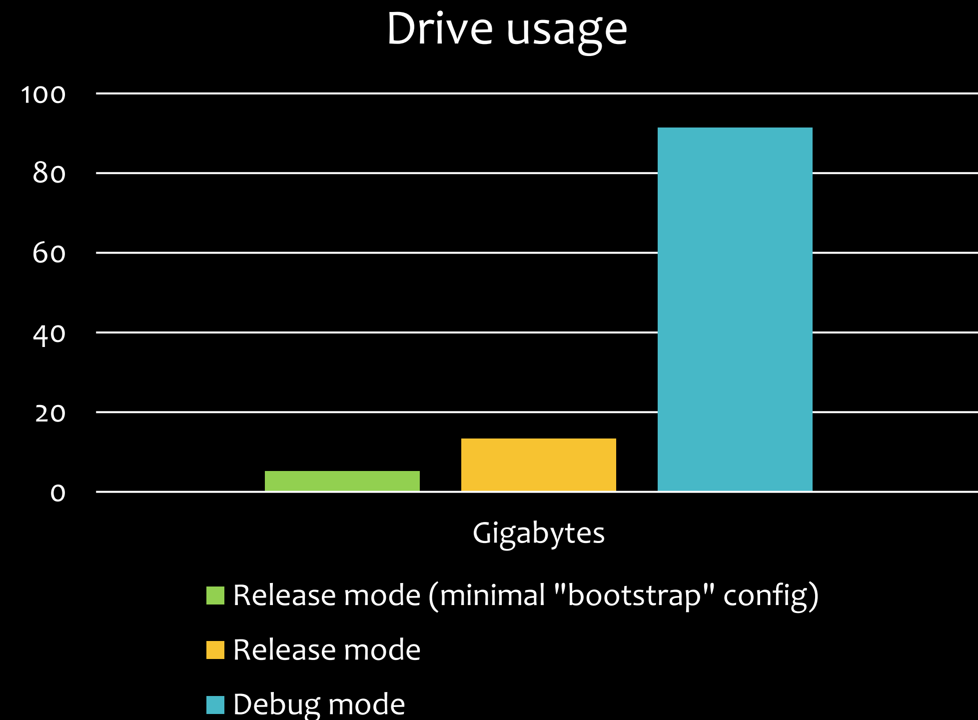
Expect it to take *at least* twice as long to build and use far more storage space.



# Debug mode building and installing

Expect it to take *at least* twice as long to build and use far more storage space.



# Release mode suggestion

- Enable everything you want
- Turn on `LLVM_ENABLE_ASSERTIONS`
- You *may* still need to limit `LLVM_PARALLEL_LINK_JOBS` to 1

# Creating a pass plugin DLL

Things become much faster here, I promise

# Essential CMake settings

- You can build with your bootstrap, release-, or debug-mode version of Clang (or MSVC if you really want to, but we've made it this far already, so… ); just be sure to use `clang-cl.exe` instead of `clang.exe`
- `CMAKE_PREFIX_PATH` needs to point at the install directory of the debug or release build you created.
  - ```
    "cacheVariables": {
        "CMAKE_PREFIX_PATH": "C:/LLVM/src/llvm/build/llvm-debug/install"
    }
    ```
    ```

# Essential CMake settings (continued)

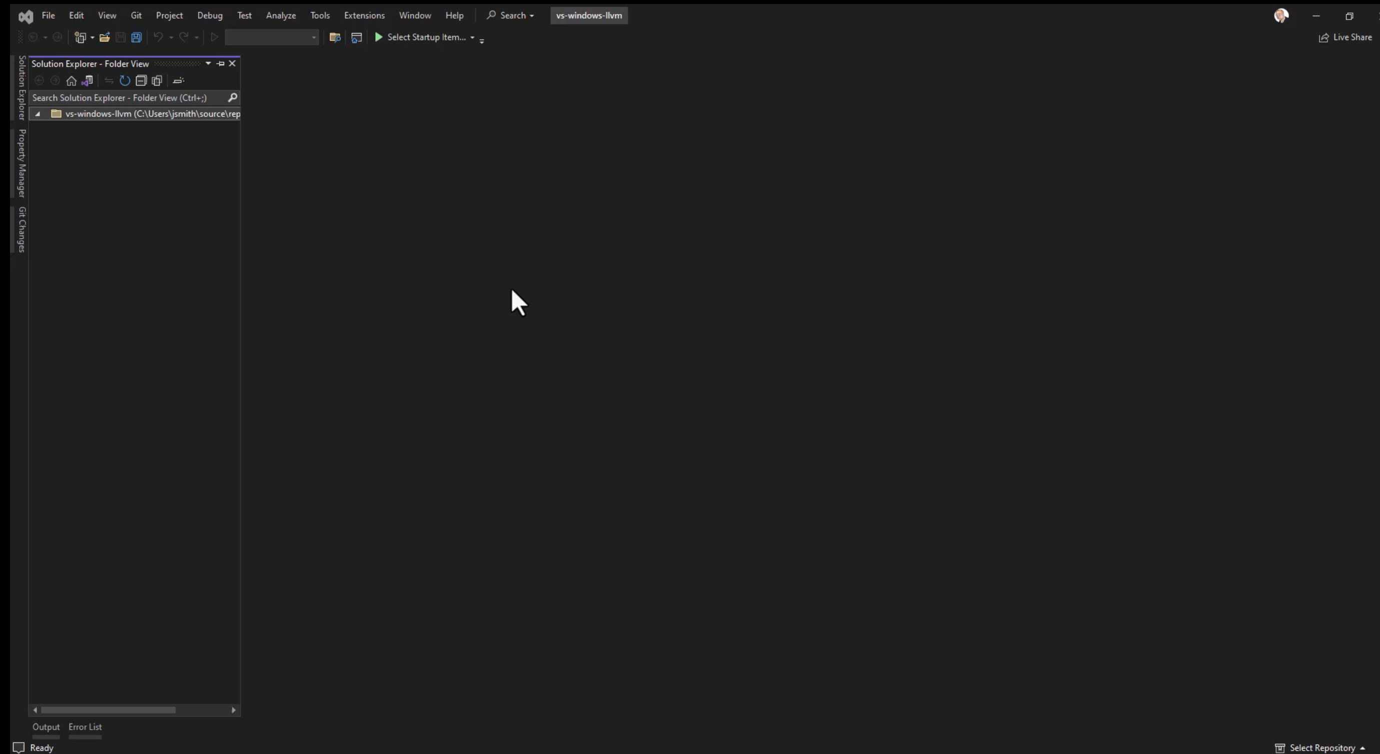
- Debug mode `_ITERATOR_DEBUG_LEVEL` *must* match that of your debug build of LLVM
- You *may* have to configure `CMAKE_MT` to be `"mt.exe"`
- Do *not* use `add_llvm_pass_plugin` – use `add_library` instead.

# Writing the passes

- StringAnalyzer – analysis pass
  - StringAnalyzerPrinter – printing pass
- StringReverser – transformation pass

# Writing the passes (demo)

Link: https://youtu.be/QiV8CeSkp2E



# `Implicit vs. explicit linking`

- Implicit linking (a.k.a. static load or load-time dynamic linking): the operating system automatically resolves symbols from and loads external libraries (DLLs/shared objects) when the process is loaded
- Explicit linking (a.k.a. on-demand runtime linking): an already running process requests the operating system to load external libraries into its process space and *manually* resolves symbols for use; loading and unloading is explicitly performed by the process
  - Windows: `LoadLibrary`, `GetProcAddress`, `FreeLibrary`
  - Linux, MacOS: `dlopen`, `dlsym`, `dlclose`

## `Which linking will be used?`

- Our pass plugin DLL links against LLVM using *implicit* linking.*
  - * That is, we *would* be if shared library builds weren’t disabled by default on Windows in LLVM’s CMake scripts. Our plugin will statically link against LLVM’s libraries.
- LLVM loads our pass plugin DLL (and many other types of plugins) using *explicit* linking.
- We need to *export* `llvmGetPassPluginInfo` for LLVM to find it in our DLL.

# Methods for exporting public symbols

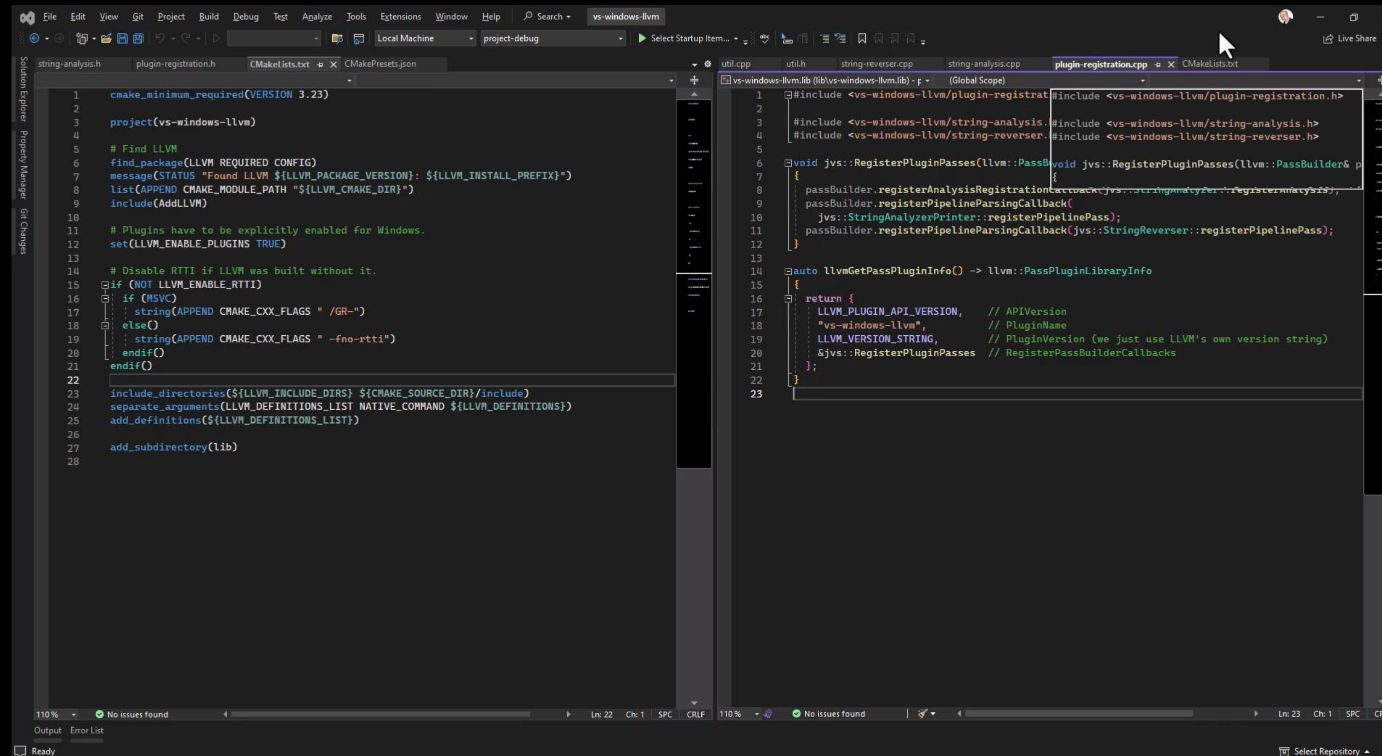
- __declspec(dllexport)

- CMake target property
  WINDOWS_EXPORT_ALL_SYMBOLS

- Module definition (.def) file ← provides the most granular control

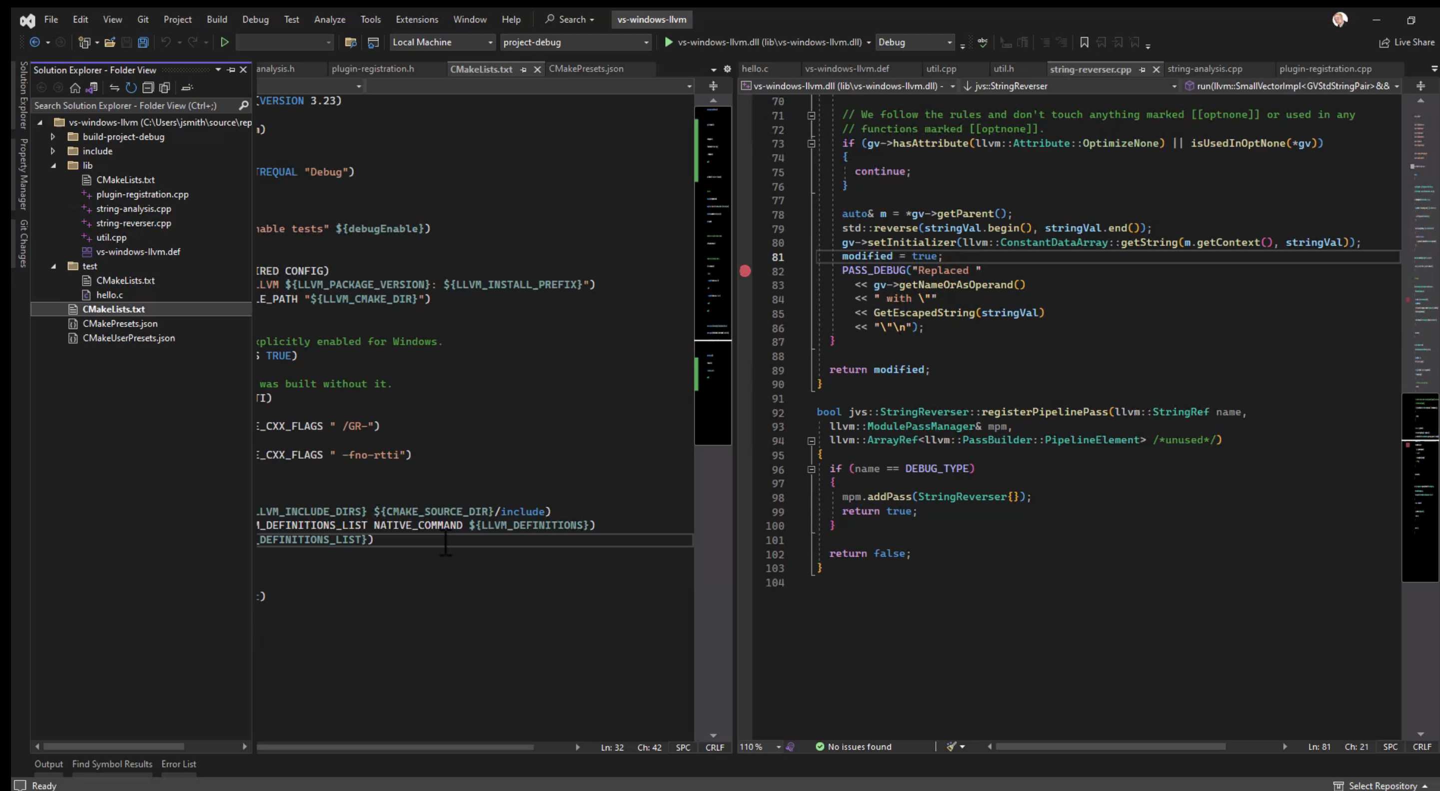
# Exporting llvmGetPassPluginInfo

Link: https://youtu.be/a0Dz-cX7W2o



# Running the passes via opt.exe

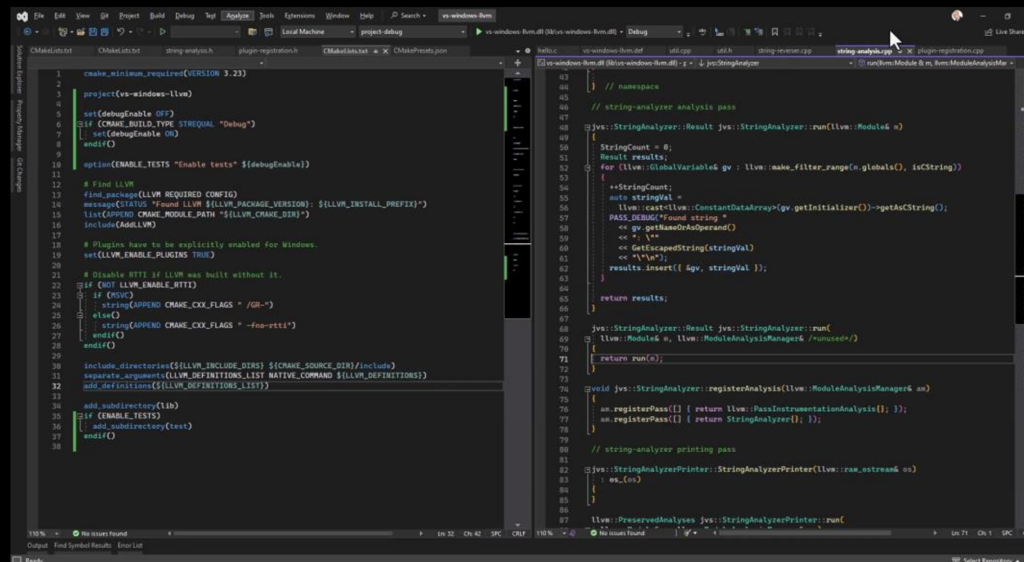
Link: https://youtu.be/X-o8814tbNs



# Debugging the passes inside Visual Studio

We can't debug the pass plugin DLL directly; we run `opt.exe --load-pass-plugin plugin.dll` and debug the `opt.exe` process instead.

Link: https://youtu.be/bPFr5L_feWU



# `Adding utility code to help with debugging`

- Visual Studio's debugger *can* display LLVM types natively – with some help.
  - `llvm::Module`
  - `llvm::Type`
  - `llvm::Value` (covers just about everything else)
- Natvis covers a lot on its own

# Adding utility code to help with debugging (demo)

Link: https://youtu.be/iUw9H9iaX-o

# Running the passes via Clang

Link: https://youtu.be/VKge6lctkO4

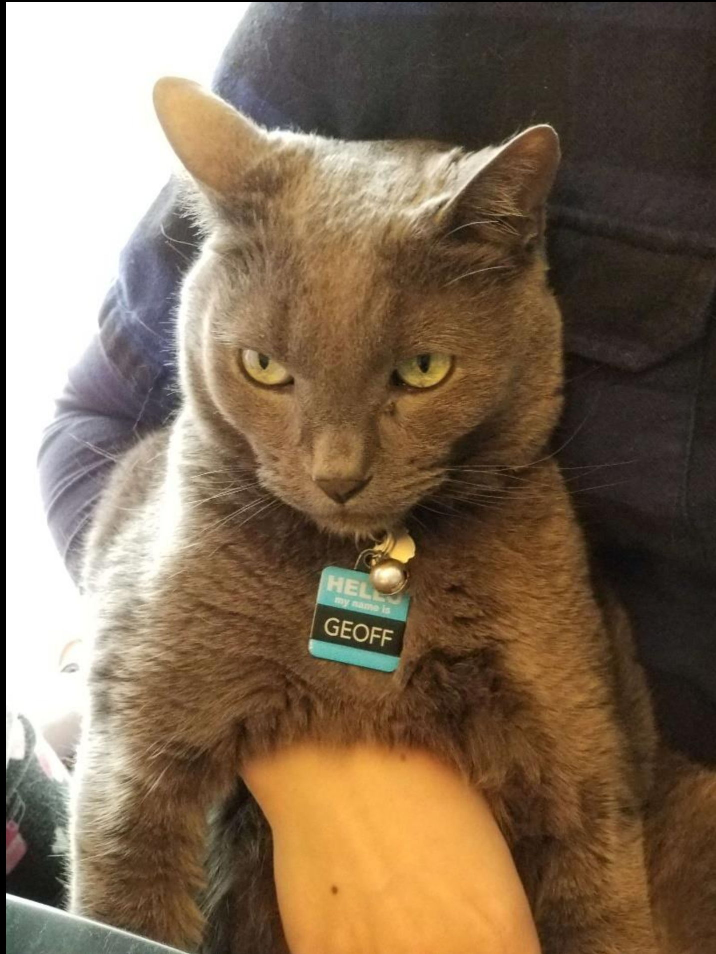
```cpp
namespace
{

using GVStringRefPair = jvs::StringAnalyzer::Result::value_type;
using GVStdStringPair = std::pair<llvm::GlobalVariable*, std::string>;

bool isUsedInOptNone(llvm::GlobalVariable& gv) noexcept
{
  auto gvInstUsers = llvm::map_range(gv.users(), [](const llvm::User* user)
    {
      return llvm::dyn_cast<llvm::Instruction>(user);
    });

  return llvm::any_of(gvInstUsers, [](const llvm::Instruction* inst)
    {
      return (inst != nullptr &&
        inst->getFunction()->hasFnAttribute(llvm::Attribute::OptimizeNone));
    });
}

GVStdStringPair toStdString(GVStringRefPair gvStringRefPair)
{
  return { gvStringRefPair.first, gvStringRefPair.second.str() };
}

}  // namespace

llvm::PreservedAnalyses jvs::StringReverser::run(llvm::Module& m,
  llvm::ModuleAnalysisManager& am)
{
  auto results = llvm::PreservedAnalyses::all();
  auto globalStrings =
    llvm::to_vector(llvm::map_range(am.getResult<StringAnalyzer>(m), toStdString));
  if (run(std::move(globalStrings)))
  {
    results.intersect(llvm::PreservedAnalyses::allInSet<llvm::CFGAnalyses>());
  }

  return results;
}

bool jvs::StringReverser::run(
  llvm::SmallVectorImpl<GVStdStringPair>&& globalStrings)
{
  bool modified = false;
  while (!globalStrings.empty())
```

```cpp
    }

    return true;
  }

}  // namespace

// string-analyzer analysis pass

jvs::StringAnalyzer::Result jvs::StringAnalyzer::run(llvm::Module& m)
{
  StringCount = 0;
  Result results;
  for (llvm::GlobalVariable& gv : llvm::make_filter_range(m.globals(), isCString))
  {
    ++StringCount;
    auto stringVal =
      llvm::cast<llvm::ConstantDataArray>(gv.getInitializer())->getAsCString();
    PASS_DEBUG("Found string "
      << gv.getNameOrAsOperand()
      << ": \""
      << GetEscapedString(stringVal)
      << "\"\n");
    results.insert({ &gv, stringVal });
  }

  return results;
}

jvs::StringAnalyzer::Result jvs::StringAnalyzer::run(
  llvm::Module& m, llvm::ModuleAnalysisManager& /*unused*/)
{
  return run(m);
}

void jvs::StringAnalyzer::registerAnalysis(llvm::ModuleAnalysisManager& am)
{
  am.registerPass([] { return llvm::PassInstrumentationAnalysis{}; });
  am.registerPass([] { return StringAnalyzer{}; });
}

// string-analyzer printing pass

jvs::StringAnalyzerPrinter::StringAnalyzerPrinter(llvm::raw_ostream& os)
  : os_(os)
```

# Cross-compiling for Linux in Visual Studio using WSL

- Use SSH for building LLVM; *not* direct WSL file system access
- Remote debugging: build debug-mode with `-gdwarf-3` for best results (YMMV)

# Questions?
# Contact info




Source code:
https://github.com/jvstech/vs-windows-llvm

- https://jvste.ch
- GitHub: jvstech
- X-Twitter: @jvs_tech
- Mastodon: @jvstech@hachyderm.io
- Twitch: jvstech
- YouTube: @jvstech