



Starting LLVM Development in Visual Studio on Windows

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

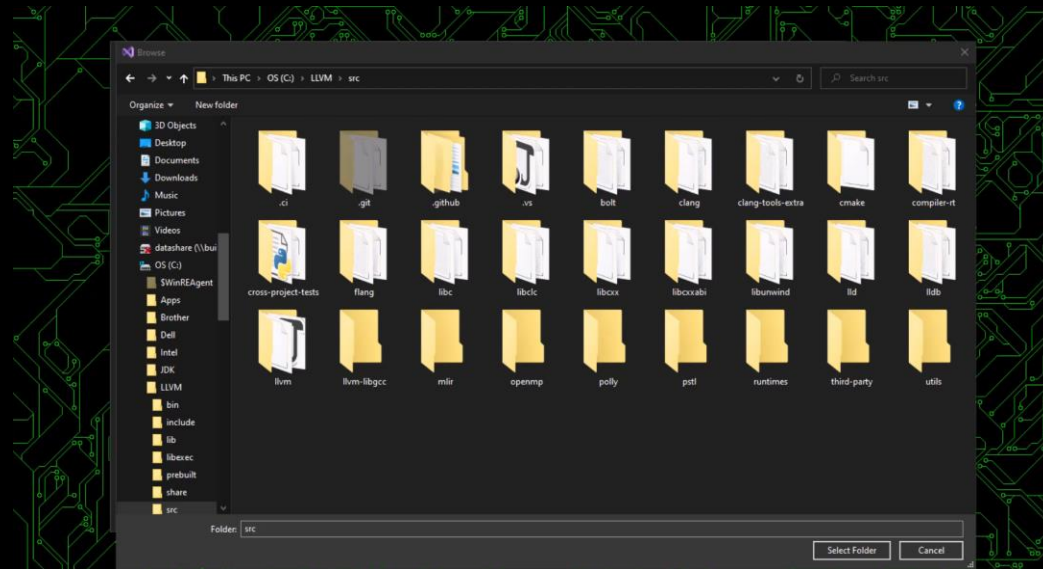
```
{
  "version": 5,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 23,
    "patch": 0
  },
  "configurePresets": [
    {
      "name": "default",
      "hidden": true,
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/build/${presetName}/build",
      "installDir": "${sourceDir}/build/${presetName}/install"
    },
    {
      "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)

```
{
  "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>





Creating a debug-mode Clang + LLVM toolchain

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

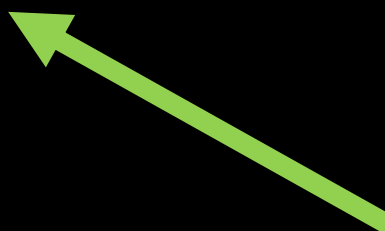
```
{  
  "name": "debug",  
  "inherits": "default",  
  "hidden": true,  
  "cacheVariables": {  
    "CMAKE_BUILD_TYPE": "Debug"  
  }  
}
```

Windows-specific debug configuration

```
{  
  "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"  
  }  
}
```



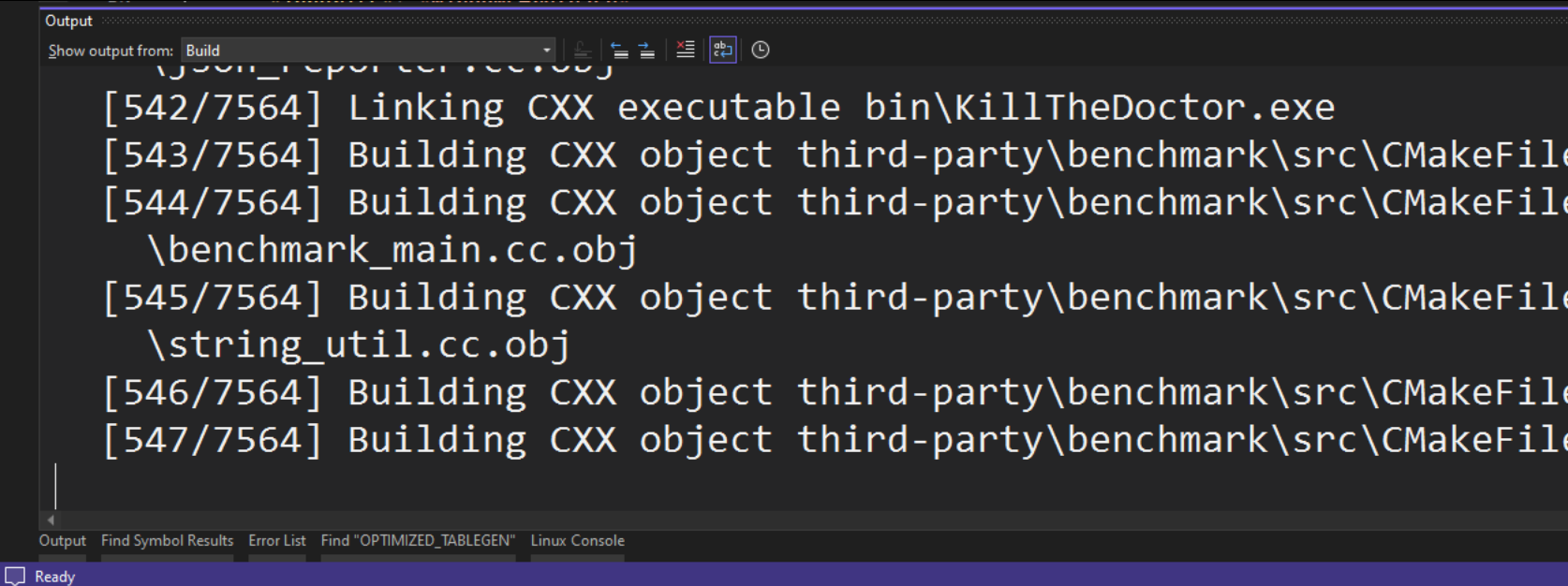
maybe

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.



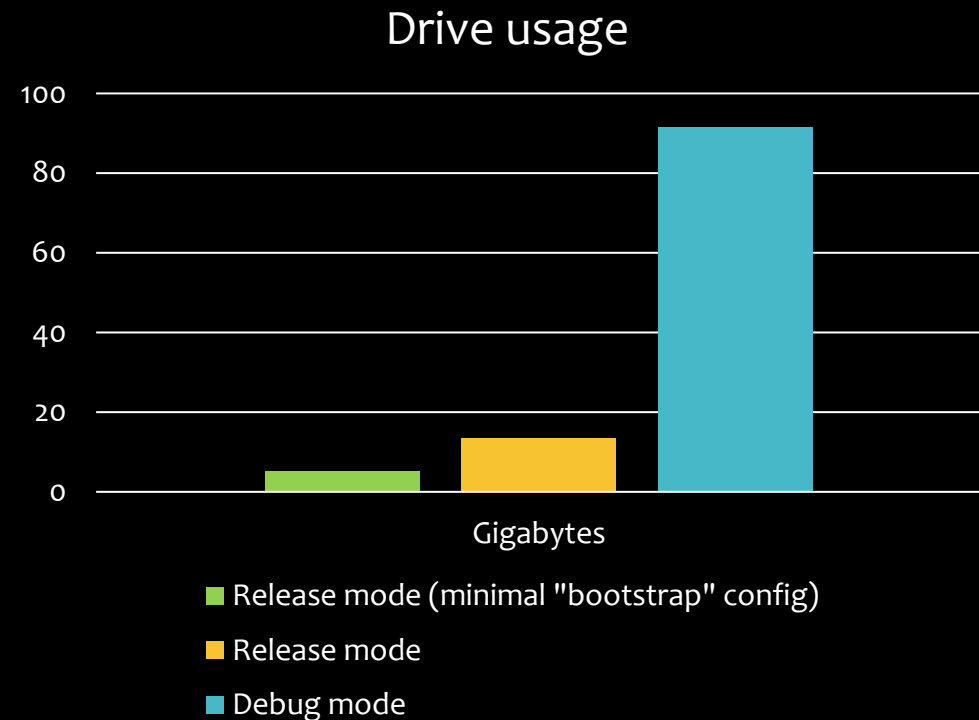
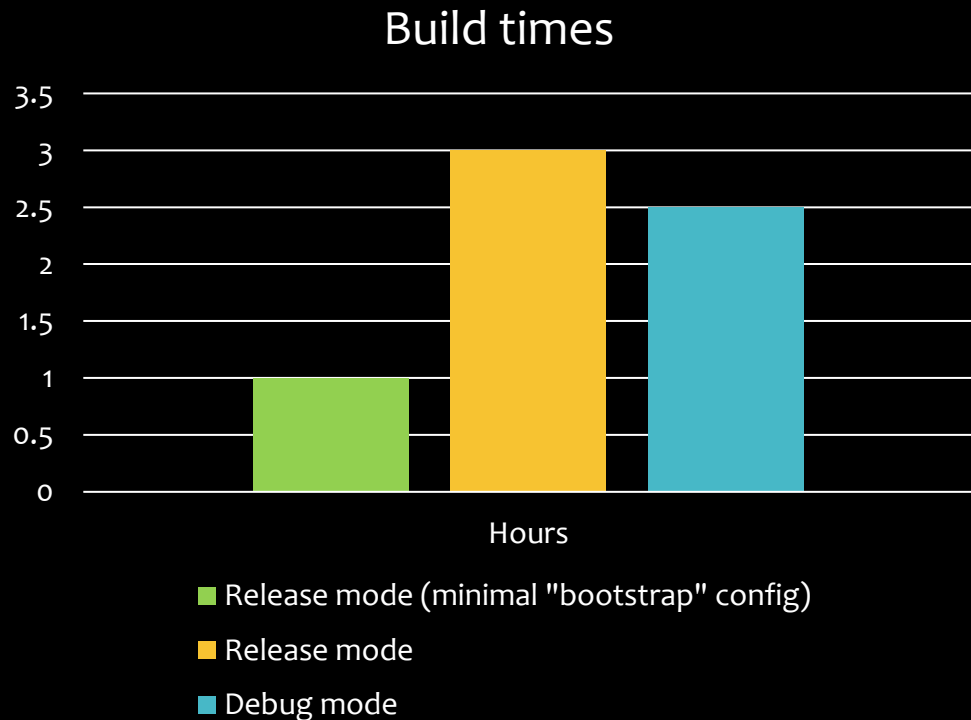
The screenshot shows the Output window of a development environment. The 'Show output from:' dropdown is set to 'Build'. The output text is as follows:

```
[542/7564] Linking CXX executable bin\KillTheDoctor.exe
[543/7564] Building CXX object third-party\benchmark\src\CMakeFile
[544/7564] Building CXX object third-party\benchmark\src\CMakeFile
\benchmark_main.cc.obj
[545/7564] Building CXX object third-party\benchmark\src\CMakeFile
\string_util.cc.obj
[546/7564] Building CXX object third-party\benchmark\src\CMakeFile
[547/7564] Building CXX object third-party\benchmark\src\CMakeFile
```

At the bottom of the window, there are tabs for 'Output', 'Find Symbol Results', 'Error List', 'Find "OPTIMIZED_TABLEGEN"', and 'Linux Console'. The status bar at the very bottom indicates 'Ready'.

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)

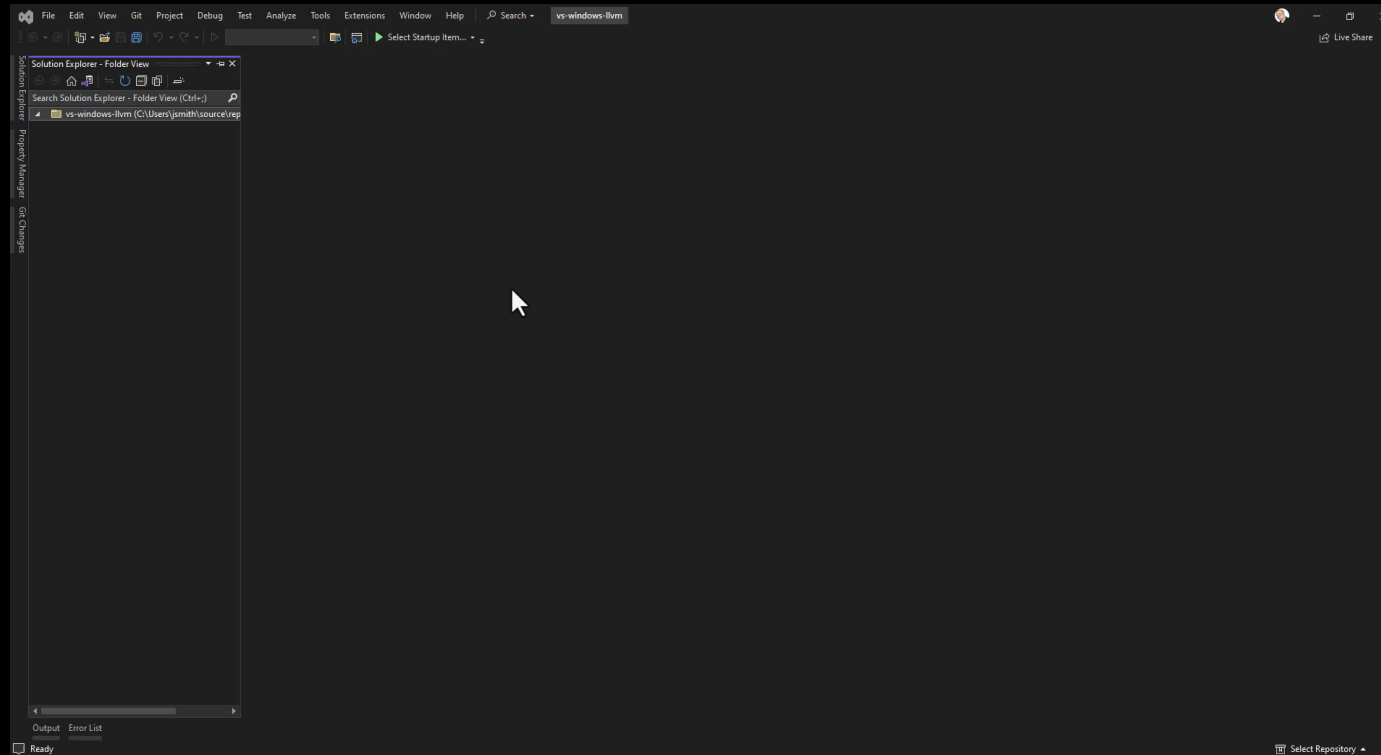
- Debug mode `_ITERATOR_DEBUG_LEVEL` must match that of your debug build of LLVM
- You *may* have to configure `CMAKE_MT` to be `"mt.exe"`
- Do *not* use `add_llvm_pass_plugin` – use `add_library` instead.

## 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?

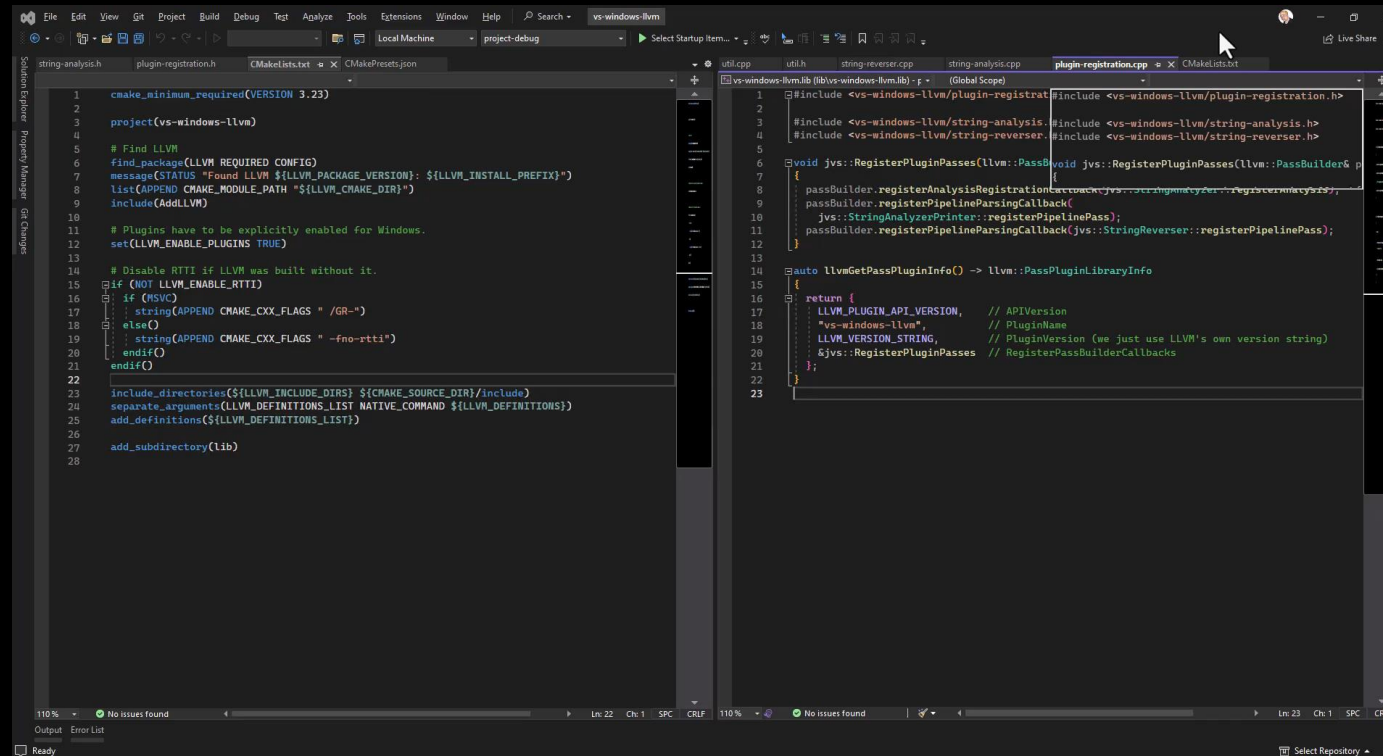
- Our pass plugin DLL links against LLVM using *implicit linking*.
  - \* That is, we would be if shared library builds weren't disabled by default on Windows in LLVM's CMake scripts. Our plugin will statically link against LLVM's libraries.
- LLVM loads our pass plugin DLL (and many other types of plugins) 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/aoDz-cX7W2o>



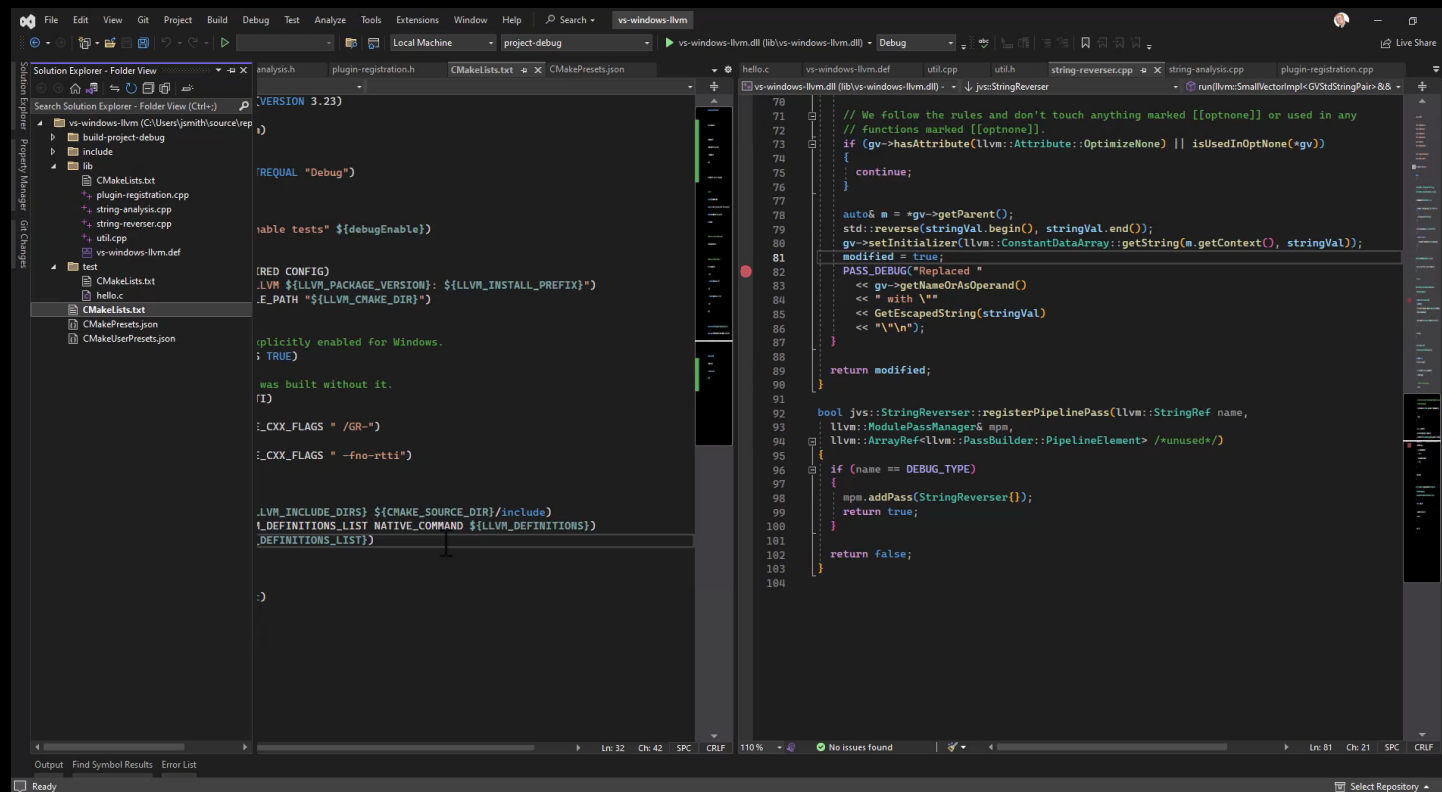
The screenshot displays the Visual Studio Code editor with two files open: CMakeLists.txt and plugin-registration.cpp. The CMakeLists.txt file on the left contains CMake configuration for a project named 'vs-windows-llvm', including package finding, RTTI disabling, and include directory management. The plugin-registration.cpp file on the right shows the implementation of the llvmGetPassPluginInfo function, which registers analysis and string reversal passes.

```
1 cmake_minimum_required(VERSION 3.23)
2
3 project(vs-windows-llvm)
4
5 # Find LLVM
6 find_package(LLVM REQUIRED CONFIG)
7 message(STATUS "Found LLVM ${LLVM_PACKAGE_VERSION}: ${LLVM_INSTALL_PREFIX}")
8 list(APPEND CMAKE_MODULE_PATH "${LLVM_CMAKE_DIR}")
9 include(AddLLVM)
10
11 # Plugins have to be explicitly enabled for Windows.
12 set(LLVM_ENABLE_PLUGINS TRUE)
13
14 # Disable RTTI if LLVM was built without it.
15 if (NOT LLVM_ENABLE_RTTI)
16 if (MSVC)
17 string(APPEND CMAKE_CXX_FLAGS " /GR-")
18 else()
19 string(APPEND CMAKE_CXX_FLAGS " -fno-rtti")
20 endif()
21 endif()
22
23 include_directories(${LLVM_INCLUDE_DIRS} ${CMAKE_SOURCE_DIR}/include)
24 separate_arguments(LLVM_DEFINITIONS_LIST NATIVE_COMMAND ${LLVM_DEFINITIONS})
25 add_definitions(${LLVM_DEFINITIONS_LIST})
26
27 add_subdirectory(lib)
28
```

```
1 #include <vs-windows-llvm/plugin-registration.h>
2
3 #include <vs-windows-llvm/string-analysis.h>
4 #include <vs-windows-llvm/string-reverser.h>
5
6 void jvs::RegisterPluginPasses(llvm::PassBuilder & passBuilder) {
7 passBuilder.registerAnalysisRegistrationCallback(jvs::StringAnalyzer::RegisterCallbacks);
8 passBuilder.registerPipelineParsingCallback(
9 jvs::StringAnalyzerPrinter::registerPipelinePass);
10 passBuilder.registerPipelineParsingCallback(jvs::StringReverser::registerPipelinePass);
11 }
12
13
14 auto llvmGetPassPluginInfo() -> llvm::PassPluginLibraryInfo
15 {
16 return {
17 LLVM_PLUGIN_API_VERSION, // APIVersion
18 "vs-windows-llvm", // PluginName
19 LLVM_VERSION_STRING, // PluginVersion (we just use LLVM's own version string)
20 &jvs::RegisterPluginPasses // RegisterPassBuilderCallbacks
21 };
22 }
23
```

# Running the passes via opt.exe

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



The screenshot displays the Visual Studio Code interface for a project named 'vs-windows-llvm'. The left sidebar shows the Solution Explorer with the following structure:

- vs-windows-llvm (C:\Users\jsmith\source\lep)
  - build-project-debug
  - include
  - lib
    - CMakeLists.txt
    - plugin-registration.cpp
    - string-analysis.cpp
    - string-reverser.cpp
    - util.cpp
  - test
    - CMakeLists.txt
    - hello.c
  - CMakeLists.txt
  - CMakePresets.json
  - CMakeUserPresets.json

The main editor area is split into two panes. The left pane shows the 'CMakeLists.txt' file with the following content:

```
VERSION 3.23

'REQUAL "Debug")

variable tests "${debugEnable}"

IRED CONFIG
LLVM ${LLVM_PACKAGE_VERSION}: ${LLVM_INSTALL_PREFIX}
E_PATH "${LLVM_CMAKE_DIR}"

explicitly enabled for Windows.
: TRUE)

was built without it.
()

CXX_FLAGS " /GR-"

CXX_FLAGS "-fno-rtti"

LLVM_INCLUDE_DIRS ${CMAKE_SOURCE_DIR}/include
DEFINITIONS_LIST NATIVE_COMMAND ${LLVM_DEFINITIONS}
DEFINITIONS_LIST))

)
```

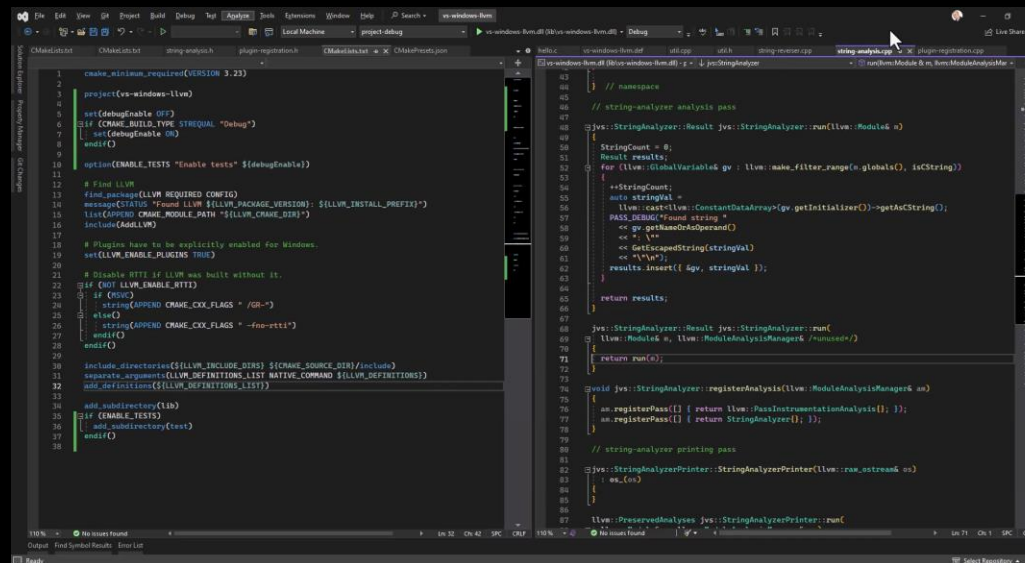
The right pane shows the 'jsStringReverser.cpp' file with the following content:

```
70
71 // We follow the rules and don't touch anything marked [[optnone]] or used in any
72 // functions marked [[optnone]].
73 if (gv->hasAttribute(LLVM::Attribute::OptimizeNone) || isUsedInOptNone(*gv))
74 {
75 continue;
76 }
77
78 auto& m = *gv->getParent();
79 std::reverse(stringVal.begin(), stringVal.end());
80 gv->setInitializer(LLVM::ConstantDataArray::getString(m.getContext(), stringVal));
81 modified = true;
82 PASS_DEBUG("Replaced "
83 << gv->getNameOrAsOperand()
84 << " with \"
85 << GetEscapedString(stringVal)
86 << "\"\n");
87 }
88 return modified;
89
90
91 bool jvs::StringReverser::registerPipelinePass(LLVM::StringRef name,
92 LLVM::ModulePassManager& mpm,
93 LLVM::ArrayRef<LLVM::PassBuilder::PipelineElement> /*unused*/)
94 {
95 if (name == DEBUG_TYPE)
96 {
97 mpm.addPass(StringReverser{});
98 return true;
99 }
100 return false;
101 }
102
103
104
```

# 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](https://youtu.be/bPFR5L_feWU)



The screenshot shows the Visual Studio IDE with two source files open. The left pane shows `plugin-registration.cpp` with the following code:

```
1 cmake_minimum_required(VERSION 3.23)
2
3 project(vcs-windows-llvm)
4
5 set(debugEnable OFF)
6 if (CHAKE_BUILD_TYPE STREQUAL "Debug")
7 | set(debugEnable ON)
8 | endif()
9
10 | option(ENABLE_TESTS "Enable tests" ${debugEnable})
11
12 | # Find LLVM
13 | find_package(LLVM REQUIRED CONFIG)
14 | message(STATUS "Found LLVM ${LLVM_PACKAGE_VERSION}: ${LLVM_INSTALL_PREFIX}")
15 | list(APPEND CHAKE_MODULE_PATH "${LLVM_CHAKE_DIR}")
16 | include(AddLLVM)
17
18 | # Plugins have to be explicitly enabled for Windows.
19 | set(LLVM_ENABLE_PLUGINS TRUE)
20
21 | # Disable RTTI if LLVM was built without it.
22 | if (NOT LLVM_ENABLE_RTTI)
23 | | string(APPEND CHAKE_CXX_FLAGS "/GR-")
24 | |
25 | | else()
26 | | string(APPEND CHAKE_CXX_FLAGS "/GR-rtti")
27 | | endif()
28 | endif()
29
30 | include_directories(${LLVM_INCLUDE_DIRS} ${CHAKE_SOURCE_DIR}/include)
31 | separate_arguments(LLVM_DEFINITIONS_LIST NATIVE_COMMAND ${LLVM_DEFINITIONS})
32 | add_definitions(${LLVM_DEFINITIONS_LIST})
33
34 | add_subdirectory(lib)
35 | if (ENABLE_TESTS)
36 | | add_subdirectory(test)
37 | | endif()
38 |
```

The right pane shows `string-analyzer.cpp` with the following code:

```
43 | // namespace
44 |
45 | // string-analyzer analysis pass
46 |
47 | jvs::StringAnalyzer::Result jvs::StringAnalyzer::run(llvm::Module & m)
48 | {
49 | StringCount = 0;
50 | Result results;
51 | for (llvm::GlobalVariable& gv : llvm::make_filter_range(m.globals(), isCString))
52 | {
53 | ++StringCount;
54 | llvm::cast<llvm::ConstantDataArray<gv>::getInitializer()->getAsCString();
55 | PASS_DEBUG("found string "
56 | << gv.getAsString().getAsString()
57 | << "\n");
58 | << GetEscapedString(stringVal)
59 | << "\n");
60 | results.insert({ gv, stringVal });
61 | }
62 | return results;
63 | }
64 |
65 |
66 |
67 | jvs::StringAnalyzer::Result jvs::StringAnalyzer::run(
68 | llvm::Module & m, llvm::ModuleAnalysisManager & manager)
69 | {
70 | return run(m);
71 | }
72 |
73 | void jvs::StringAnalyzer::registerAnalysis(llvm::ModuleAnalysisManager & m)
74 | {
75 | m.registerPass([] { return llvm::PassInstrumentationAnalysis(); });
76 | m.registerPass([] { return StringAnalyzer(); });
77 | }
78 |
79 | // string-analyzer printing pass
80 |
81 | jvs::StringAnalyzerPrinter::StringAnalyzerPrinter(llvm::raw_ostream & os)
82 | : os(os)
83 | {}
84 |
85 |
86 |
87 | llvm::PassesAndAnalyses jvs::StringAnalyzerPrinter::runC
```

# 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/iUwgHgiaX-o>

```
59 llvm::SmallVectorImpl<GVStdStringPair>& globalStrings)
60 {
61 bool modified = false;
62 while (!globalStrings.empty())
63 {
64 auto [gv, stringVal] = globalStrings.pop_back_val();
65 if (stringVal.empty())
66 // Difficult to reverse an empty string.
67 continue;
68
69 // We follow the rules and don't touch anything marked [[optnone]] or used in any
70 // functions marked [[optnone]].
71 if (gv->hasAttribute(llvm::Attribute::OptimizeNone) || isUsedInOptNone(*gv))
72 continue;
73
74 auto& m = *gv->getParent();
75 std::reverse(stringVal.begin(), stringVal.end());
76 gv->setInitializer(llvm::ConstantDataArray::getString(m.getContext(), stringVal));
77 modified = true;
78 PASS_DEBUG("Replaced "
79 << gv->getNameOrAsOperand()
80 << " with \""
81 << GetEscapedString(stringVal)
82 << "\"");
83 }
84 return modified;
85 }
86
87 bool jvs::StringReverser::registerPipelinePass(llvm::StringRef name,
88 llvm::ModulePassManager& mpm,
89 llvm::ArrayRef<llvm::PassBuilder::PipelineElement> /*unused*/)
90 {
91 if (name == DEBUG_TYPE)
92 {
93 mpm.addPass(StringReverser{});
94 return true;
95 }
96 return false;
97 }
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```



# Running the passes via Clang

Link: <https://youtu.be/VKge6lctkO4>

The image shows a screenshot of the Visual Studio Code editor with two C++ source files open. The left file, `string-reverser.cpp`, defines a `StringReverser` pass that iterates over global strings and reverses them. The right file, `string-analyzer.cpp`, defines a `StringAnalyzer` pass that identifies string literals in the code. Both files are part of an LLVM project, as indicated by the headers and the `llvm::Module` and `llvm::ModuleAnalysisManager` usage.

```
17 namespace
18 {
19
20 using GVStringRefPair = jvs::StringAnalyzer::Result::value_type;
21 using GVStdStringPair = std::pair<llvm::GlobalVariable*, std::string>;
22
23 bool isUsedInOptNone(llvm::GlobalVariable& gv) noexcept
24 {
25 auto gvInstUsers = llvm::map_range(gv.users(), [](const llvm::User* user)
26 {
27 return llvm::dyn_cast<llvm::Instruction>(user);
28 });
29
30 return llvm::any_of(gvInstUsers, [](const llvm::Instruction* inst)
31 {
32 return (inst != nullptr &&
33 inst->getFunction()->hasFnAttribute(llvm::Attribute::OptimizeNone));
34 });
35 }
36
37 GVStdStringPair toStdString(GVStringRefPair gvStringRefPair)
38 {
39 return { gvStringRefPair.first, gvStringRefPair.second.str() };
40 }
41
42 // namespace
43
44 llvm::PreservedAnalyses jvs::StringReverser::run(llvm::Module& m,
45 llvm::ModuleAnalysisManager& am)
46 {
47 auto results = llvm::PreservedAnalyses::all();
48 auto globalStrings =
49 llvm::to_vector(llvm::map_range(am.getResult<StringAnalyzer>(m), toStdString));
50 if (run(std::move(globalStrings)))
51 {
52 results.intersect(llvm::PreservedAnalyses::allInSet<llvm::CFGAnalyses>());
53 }
54
55 return results;
56 }
57
58 bool jvs::StringReverser::run(
59 llvm::SmallVectorImpl<GVStdStringPair>&& globalStrings)
60 {
61 bool modified = false;
62 while (!globalStrings.empty())
```

```
39
40
41 return true;
42 }
43
44 // namespace
45
46 // string-analyzer analysis pass
47
48 jvs::StringAnalyzer::Result jvs::StringAnalyzer::run(llvm::Module& m)
49 {
50 StringCount = 0;
51 Result results;
52 for (llvm::GlobalVariable& gv : llvm::make_filter_range(m.globals(), isCString))
53 {
54 ++StringCount;
55 auto stringVal =
56 llvm::cast<llvm::ConstantDataArray>(gv.getInitializer()->getAsCString());
57 PASS_DEBUG("Found string "
58 << gv.getNameOrAsOperand()
59 << " : \""
60 << GetEscapedString(stringVal)
61 << "\"\n");
62 results.insert({ gv, stringVal });
63 }
64
65 return results;
66 }
67
68 jvs::StringAnalyzer::Result jvs::StringAnalyzer::run(
69 llvm::Module& m, llvm::ModuleAnalysisManager& /#unused/)
70 {
71 return run(m);
72 }
73
74 void jvs::StringAnalyzer::registerAnalysis(llvm::ModuleAnalysisManager& am)
75 {
76 am.registerPass([] { return llvm::PassInstrumentationAnalysis{}; });
77 am.registerPass([] { return StringAnalyzer{}; });
78 }
79
80 // string-analyzer printing pass
81
82 jvs::StringAnalyzerPrinter::StringAnalyzerPrinter(llvm::raw_ostream& os)
83 : 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