VSCode Configuration and Tips for Flang Development

Anthony Cabrera
Research Scientist, Architectures and Performance Group, CSMD
Oak Ridge National Laboratory
January 4, 2023

Overview

VSCode Configuration

Stepping Through Code

Case Study

Preliminary Stuff

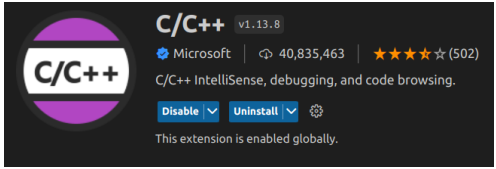
- You can find my slides here
- You can find my configuration files here

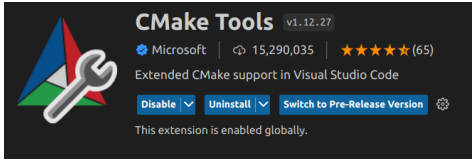
Current Topic

Current Topic

Extensions to Download

- C/C++
- CMake





Current Topic

VSCode Configuration

- VSCode Extensions
- Configuration Files
  - `CMakePresets.json` and `CMakeUserPresets.json`
  - `settings.json`
  - `launch.json`
  - `tasks.json`

Stepping Through Code

Case Study

`CMakePresets.json` and `CMakeUserPresets.json`

These files allow you to create multiple configurations for the configure, generate, and build steps of any CMake project.

`CMakePresets.json` and `CMakeUserPresets.json`

These files allow you to create multiple configurations for the configure, generate, and build steps of any CMake project.

- Place these files in the `llvm` directory

`CMakePresets.json` and `CMakeUserPresets.json`

These files allow you to create multiple configurations for the configure, generate, and build steps of any CMake project.

- Place these files in the `llvm` directory
- `CMakePresets.json` are for common configurations, and `CMakeUserPresets.json` are on a per-user basis

`CMakePresets.json` and `CMakeUserPresets.json`

These files allow you to create multiple configurations for the configure, generate, and build steps of any CMake project.

- Place these files in the `llvm` directory
- `CMakePresets.json` are for common configurations, and `CMakeUserPresets.json` are on a per-user basis
  - Specifically, in the Flang project, the `CMakePresets.json` config can contain the default build configuration from the Flang `README.md`, and the `CMakePresets.json` can contain your personal system-specific details for building on your machine

CMakePresets.json and CMakeUserPresets.json

These files allow you to create multiple configurations for the configure, generate, and build steps of any CMake project.

- Place these files in the `llvm` directory
- `CMakePresets.json` are for common configurations, and `CMakeUserPresets.json` are on a per-user basis
  - Specifically, in the Flang project, the `CMakePresets.json` config can contain the default build configuration from the Flang `README.md`, and the `CMakePresets.json` can contain your personal system-specific details for building on your machine

Use these config files over `cmake-kits.json` and `cmake-variants.json`

It's cross-plaform; you can share your `CMakePresets.json` and `CMakeUserPresets.json` files with someone who does not use VSCode as their IDE, since the presets files are a 'CMake' feature and not a VSCode feature. Also, this is now the recommended method from Microsoft developers.

`CMakePresets.json` Example

```
{
  "version": 5,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 23,
    "patch": 0
  },
  "include": [],
  "configurePresets": [
    {
      "name": "Flang Default Configure",
      "displayName": "Flang Default Configure",
      "description": "Flang Default configure recipe given from Flang docs",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/../build_flang_default",
      "cacheVariables": {
        "CMAKE_INSTALL_PREFIX": "${sourceDir}/../install_flang_default",
        "CMAKE_CXX_STANDARD": "17",
        "CMAKE_BUILD_TYPE": "Release",
        "CMAKE_EXPORT_COMPILE_COMMANDS": "ON",
        "CMAKE_CXX_LINK_FLAGS": "-Wl,-rpath,$LD_LIBRARY_PATH",
        "FLANG_ENABLE_WERROR": "ON",
        "LLVM_ENABLE_ASSERTIONS": "ON",
        "LLVM_TARGETS_TO_BUILD": "host",
        "LLVM_LIT_ARGS": "-v",
        "LLVM_ENABLE_PROJECTS": "clang;mlir;flang;openmp",
        "LLVM_ENABLE_RUNTIMES": "compiler-rt"
      }
    }
  ],
  "buildPresets": [ ⋯
  ],
  "testPresets": [ ⋯
  ]
}
```

CMakeUserPresets.json Example

llvm > {} CMakeUserPresets.json > [ ] testPresets

```
{
  "version": 5,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 23,
    "patch": 0
  },
  "include": [
  ],
  "configurePresets": [
    {…
    },
    {…
    },
    {
      "name": "default_local_relwithdebuginfo_with_gcc",
      "inherits": [
        "Flang Default Configure"
      ],
      "binaryDir": "${sourceDir}/../build_relwithdebuginfo_gcc",
      "displayName": "local Default relwithdebuginfo Config with gcc",
      "description": "local Default relwithdebuginfo build using Ninja generator",
      "environment": {
      },
      "cacheVariables": {
        "CMAKE_C_COMPILER": "gcc",
        "CMAKE_CXX_COMPILER": "g++",
        "CMAKE_BUILD_TYPE": "RelWithDebInfo"
      }
    }
  ],
  "buildPresets": [
    {…
    },
    {…
    },
    {
      "name": "build_from_user_presets_gcc_relwithdebuginfo",
      "configurePreset": "default_local_relwithdebuginfo_with_gcc",
      "displayName": "Build from user presets gcc relwithdebuginfo",
      "description": "local Default build using Ninja generator",
      "jobs": 128
    }
  ],
  "testPresets": [
  ]
}
```

settings.json

This config file configures your project in the current VSCode workspace.

`settings.json`

This config file configures your project in the current VSCode workspace.

- Place this file in the `.vscode` directory

`settings.json`

This config file configures your project in the current VSCode workspace.

- Place this file in the `.vscode` directory
- Specifically in LLVM and Flang, we need to associate `.inc` files `c++` files

settings.json

This config file configures your project in the current VSCode workspace.

- Place this file in the `.vscode` directory
- Specifically in LLVM and Flang, we need to associate `.inc` files `c++` files
- You can't step through auto-generated `.inc` files unless you make this association

settings.json

This config file configures your project in the current VSCode workspace.

- Place this file in the `.vscode` directory
- Specifically in LLVM and Flang, we need to associate `.inc` files `c++` files
- You can't step through auto-generated `.inc` files unless you make this association
- Example `settings.json` file:

.vscode > {} settings.json > ...

```
{
  "files.associations": {
    "*.inc": "cpp",
    "*.tcc": "cpp",
    "string": "cpp",
    "regex": "cpp"
  },
  "cmake.sourceDirectory": "${workspaceFolder}/llvm",
  "cmake.parallelJobs": 16,
  "git.ignoreLimitWarning": true,
}
```

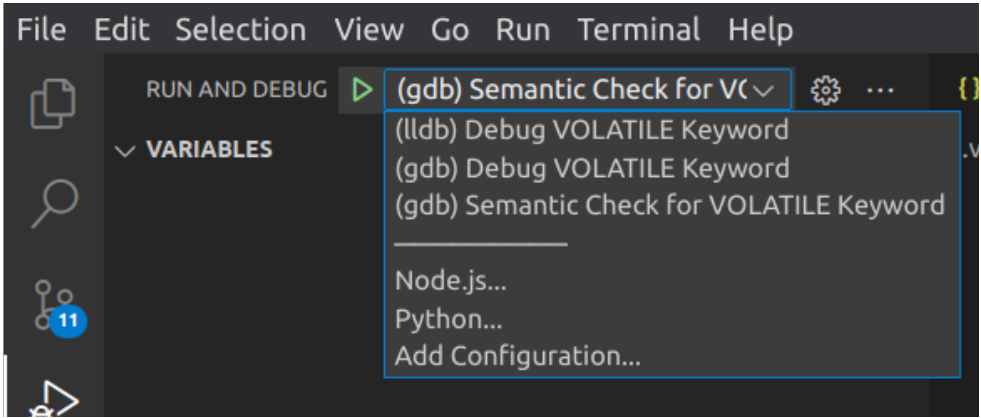
launch.json

A launch.json file is used to configure the debugger in VSCode

launch.json

A `launch.json` file is used to configure the debugger in VSCode

- This is useful for quickly selecting between debugging tasks



launch.json

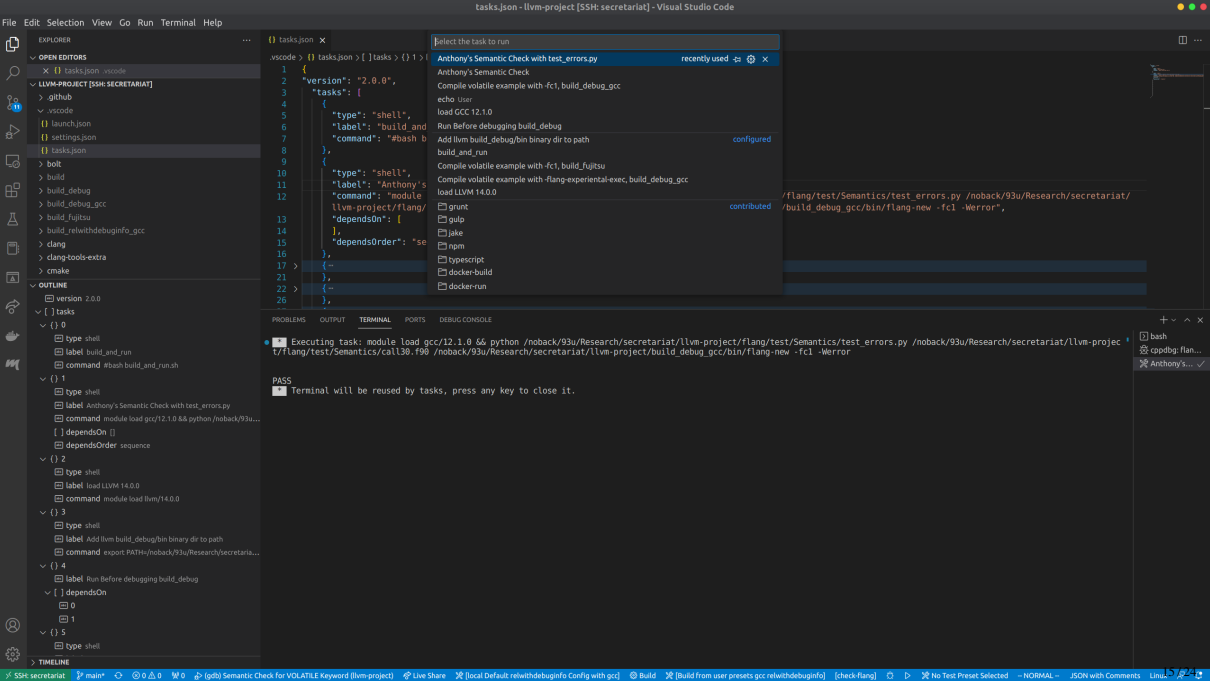
tasks.json

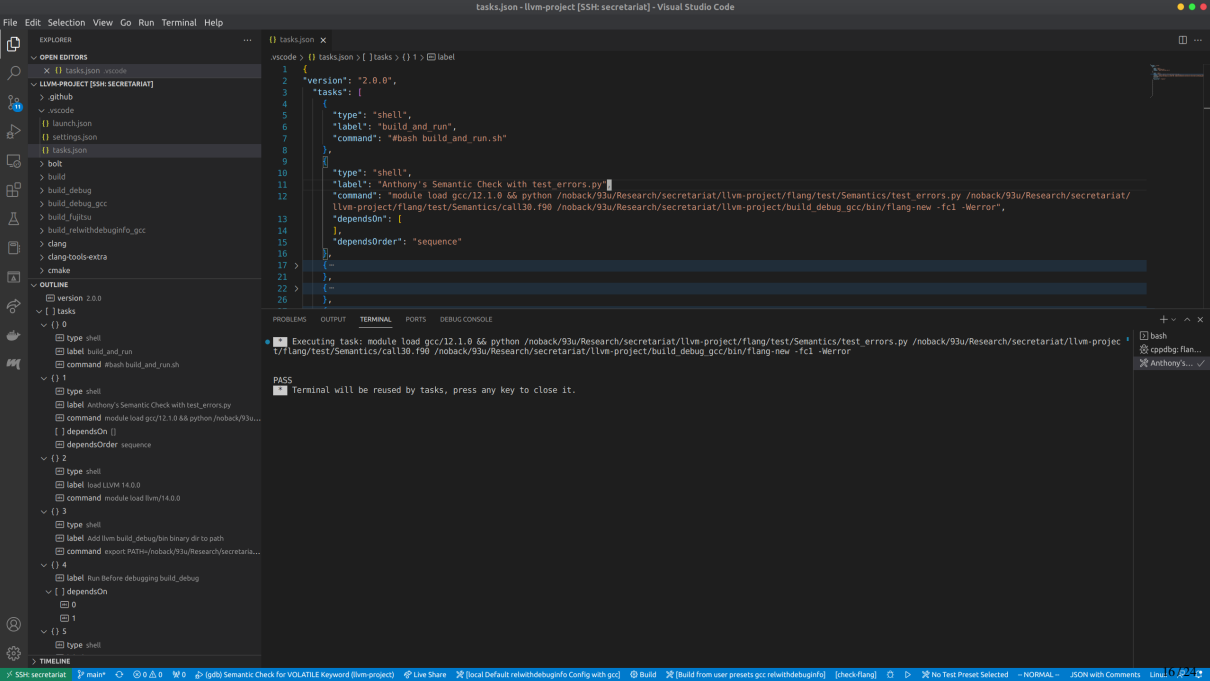
This config file allows you to create tasks with external tools, e.g., through shell commands, that are easily navigatable and can be executed within the VSCode environment

`tasks.json`

This config file allows you to create tasks with external tools, e.g., through shell commands, that are easily navigatable and can be executed within the VSCode environment

- For example, this is helpful for *quickly* compiling a given test to check if LLVMFlang behaved as expected or not


tasks.json - llvm-project [SSH: secretariat] - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
OPEN EDITORS
tasks.json .vscode
LLVM-PROJECT [SSH: SECRETARIAT]
.github
.vscode
launch.json
settings.json
tasks.json
bolt
build
build_debug
build_debug_gcc
build_fujitsu
build_relwithdebuginfo_gcc
clang
clang-tools-extra
cmake
OUTLINE
version 2.0.0
tasks
0
type shell
label build_and_run
command #bash build_and_run.sh
1
type shell
label Anthony's Semantic Check with test_errors.py
command module load gcc/12.1.0 && python /noback/93u...
dependsOn []
dependsOrder sequence
2
type shell
label load LLVM 14.0.0
command module load llvm/14.0.0
3
type shell
label Add llvm build_debug/bin binary dir to path
command export PATH=/noback/93u/Research/secretaria...
4
label Run Before debugging build_debug
dependsOn
0
1
5
type shell
TIMELINE
tasks.json
.vscode > tasks.json > [ ] tasks > { } 1 > [ ] dependsOn
Select the task to run
Anthony's Semantic Check with test_errors.py recently used
Anthony's Semantic Check
Compile volatile example with -fc1, build_debug_gcc
echo user
load GCC 12.1.0
Run Before debugging build_debug
Add llvm build_debug/bin binary dir to path configured
build_and_run
Compile volatile example with -fc1, build_fujitsu
Compile volatile example with -flang-experiental-exec, build_debug_gcc
load LLVM 14.0.0
grunt contributed
gulp
jake
npm
typescript
docker-build
docker-run
PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE
Executing task: module load gcc/12.1.0 && python /noback/93u/Research/secretariat/llvm-project/flang/test/Semantics/test_errors.py /noback/93u/Research/secretariat/llvm-project/flang/test/Semantics/call30.f90 /noback/93u/Research/secretariat/llvm-project/build_debug_gcc/bin/flang-new -fc1 -Werror
PASS
Terminal will be reused by tasks, press any key to close it.
bash
cppdbg: flan...
Anthony's...
SSH: secretariat main* 0 0 0 (gdb) Semantic Check for VOLATILE Keyword (llvm-project) Live Share [local Default relwithdebuginfo Config with gcc] Build [Build from user presets gcc relwithdebuginfo] [check-flang] No Test Preset Selected -- NORMAL -- JSON with Comments Ln 1, Col 1


tasks.json - llvm-project [SSH: secretariat] - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
OPEN EDITORS
tasks.json .vscode
LLVM-PROJECT [SSH: SECRETARIAT]
.github
.vscode
launch.json
settings.json
tasks.json
bolt
build
build_debug
build_debug_gcc
build_fujitsu
build_relwithdebuginfo_gcc
clang
clang-tools-extra
cmake
OUTLINE
version 2.0.0
[ ] tasks
{ } 0
type shell
label build_and_run
command #bash build_and_run.sh
{ } 1
type shell
label Anthony's Semantic Check with test_errors.py
command module load gcc/12.1.0 && python /noback/93u...
[ ] dependsOn []
dependsOrder sequence
{ } 2
type shell
label load LLVM 14.0.0
command module load llvm/14.0.0
{ } 3
type shell
label Add llvm build_debug/bin binary dir to path
command export PATH=/noback/93u/Research/secretaria...
{ } 4
label Run Before debugging build_debug
[ ] dependsOn
0
1
{ } 5
type shell
TIMELINE
tasks.json
.vscode > tasks.json > [ ] tasks > { } 1 > label
{
"version": "2.0.0",
"tasks": [
{
"type": "shell",
"label": "build_and_run",
"command": "#bash build_and_run.sh"
},
{
"type": "shell",
"label": "Anthony's Semantic Check with test_errors.py",
"command": "module load gcc/12.1.0 && python /noback/93u/Research/secretariat/llvm-project/flang/test/Semantics/test_errors.py /noback/93u/Research/secretariat/
llvm-project/flang/test/Semantics/call30.f90 /noback/93u/Research/secretariat/llvm-project/build_debug_gcc/bin/flang-new -fc1 -Werror",
"dependsOn": [
],
"dependsOrder": "sequence"
},
{ ⋯
},
{ ⋯
},
PROBLEMS
OUTPUT
TERMINAL
PORTS
DEBUG CONSOLE
Executing task: module load gcc/12.1.0 && python /noback/93u/Research/secretariat/llvm-project/flang/test/Semantics/test_errors.py /noback/93u/Research/secretariat/llvm-project/flang/test/Semantics/call30.f90 /noback/93u/Research/secretariat/llvm-project/build_debug_gcc/bin/flang-new -fc1 -Werror
PASS
Terminal will be reused by tasks, press any key to close it.
bash
cppdbg: flan...
Anthony's ...
SSH: secretariat
main*
0 0 0
(gdb) Semantic Check for VOLATILE Keyword (llvm-project)
Live Share
[local Default relwithdebuginfo Config with gcc]
Build
[Build from user presets gcc relwithdebuginfo]
[check-flang]
No Test Preset Selected
-- NORMAL --
JSON with Comments
Ln 17, Col 1

Current Topic

VSCode Configuration

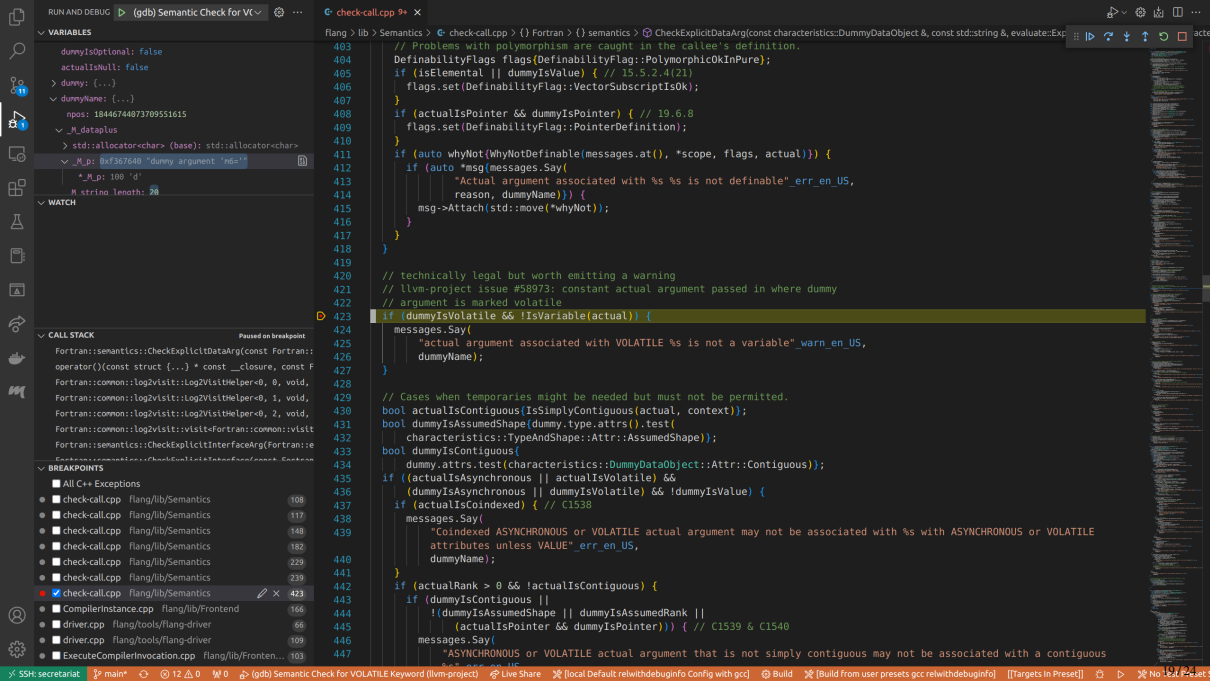
Stepping Through Code

Case Study

Using GDB to Step Through Code

Though it's somewhat slow to step through LLVM using a debugger, the tradeoff is that you can *interactively* see what the code is doing. This is particularly useful when you're still learning the codebase (like me!)

- You can examine the state of all variables
- You can jump through all currently active call stacks
- You can easily see where certain routines/variables are defined


RUN AND DEBUG
(gdb) Semantic Check for VC
VARIABLES
dummyIsOptional: false
actualIsNull: false
dummy: {...}
dunnyName: {...}
npos: 18446744073709551615
_M_dataplus
std::allocator<char> (base): std::allocator<char>
_M_p: 0xf367640 "dummy argument 'n6='"
*_M_p: 100 'd'
M_string_length: 20
WATCH
CALL STACK
Paused on breakpoint
Fortran::semantics::CheckExplicitDataArg(const Fortran::
operator()(const struct {...} * const __closure, const F
Fortran::common::log2visit::Log2VisitHelper<0, 0, void,
Fortran::common::log2visit::Log2VisitHelper<0, 1, void,
Fortran::common::log2visit::Log2VisitHelper<0, 2, void,
Fortran::common::log2visit::visit<Fortran::common::visit
Fortran::semantics::CheckExplicitInterfaceArg(Fortran::e
BREAKPOINTS
All C++ Exceptions
check-call.cpp flang/lib/Semantics 108
check-call.cpp flang/lib/Semantics 117
check-call.cpp flang/lib/Semantics 148
check-call.cpp flang/lib/Semantics 182
check-call.cpp flang/lib/Semantics 229
check-call.cpp flang/lib/Semantics 239
check-call.cpp flang/lib/Semantics 423
CompilerInstance.cpp flang/lib/Frontend 166
driver.cpp flang/tools/flang-driver 66
driver.cpp flang/tools/flang-driver 109
ExecuteCompilerInvocation.cpp flang/lib/Fronten... 103
check-call.cpp 9+
flang > lib > Semantics > check-call.cpp > { } Fortran > { } semantics > CheckExplicitDataArg(const characteristics::DummyDataObject &, const std::string &, evaluate::Exp
// Problems with polymorphism are caught in the callee's definition.
DefinabilityFlags flags{DefinabilityFlag::PolymorphicOkInPure};
if (isElemental || dummyIsValue) { // 15.5.2.4(21)
flags.set(DefinabilityFlag::VectorSubscriptIsOk);
}
if (actualIsPointer && dummyIsPointer) { // 19.6.8
flags.set(DefinabilityFlag::PointerDefinition);
}
if (auto whyNot{WhyNotDefinable(messages.at(), *scope, flags, actual)}) {
if (auto *msg{messages.Say(
"Actual argument associated with %s %s is not definable"_err_en_US,
reason, dummyName)}) {
msg->Attach(std::move(*whyNot));
}
}
}
// technically legal but worth emitting a warning
// llvm-project issue #58973: constant actual argument passed in where dummy
// argument is marked volatile
if (dummyIsVolatile && !IsVariable(actual)) {
messages.Say(
"actual argument associated with VOLATILE %s is not a variable"_warn_en_US,
dummyName);
}
// Cases when temporaries might be needed but must not be permitted.
bool actualIsContiguous{IsSimplyContiguous(actual, context)};
bool dummyIsAssumedShape{dummy.type.attrs().test(
characteristics::TypeAndShape::Attr::AssumedShape)};
bool dummyIsContiguous{
dummy.attrs.test(characteristics::DummyDataObject::Attr::Contiguous)};
if ((actualIsAsynchronous || actualIsVolatile) &&
(dummyIsAsynchronous || dummyIsVolatile) && !dummyIsValue) {
if (actualIsCoindexed) { // C1538
messages.Say(
"Coindexed ASYNCHRONOUS or VOLATILE actual argument may not be associated with %s with ASYNCHRONOUS or VOLATILE
attributes unless VALUE"_err_en_US,
dummyName);
}
if (actualRank > 0 && !actualIsContiguous) {
if (dummyIsContiguous ||
!(dummyIsAssumedShape || dummyIsAssumedRank ||
(actualIsPointer && dummyIsPointer))) { // C1539 & C1540
messages.Say(
"ASYNCHRONOUS or VOLATILE actual argument that is not simply contiguous may not be associated with a contiguous
SSH: secretariat
main*
12 0
0
(gdb) Semantic Check for VOLATILE Keyword (llvm-project)
Live Share
[local Default relwithdebinfo Config with gcc]
Build
[Build from user presets gcc relwithdebinfo]
[Targets In Preset]
No Test Preset s

Current Topic

VSCode Configuration

Stepping Through Code

Case Study

- `llvm-project` issue 58973

Current Topic

VSCode Configuration

Stepping Through Code

Case Study

- `llvm-project` issue 58973

`llvm-project` issue 58973

- Here is an issue that is raising an error when a warning is more appropriate
  - A little more specifically, an attribute of a dummy argument does not correspond to the actual argument being passed
- Besides knowing that I want to step through the code using `gdb`, I have no idea where to start, so I of course ask the Flang community for help
- After a tip, I find that `flang/lib/Semantics/check-call.cpp` might be where the fix should be applied
- Before going further, I start configuring VSCode + CMake
  - I set up `CMakeUserPresets.json` with the default Flang configuration, and set up `CMakeUserUserPresets.json` with details specific to my system as well as setting the `RelWithDebInfo` build option

`llvm-project` issue 58973 (cont.)

- I visually scan the `flang/lib/Semantics/check-call.cpp` and start setting breakpoints in places where I think might be useful
- I start implementing what I think the fix should be
- Once I have something I think works, it's time to build LLVM using the CMake extension integrated into our VSCode environment
- At this point, I can start editing the `launch.json` and `tasks.json` config files
  - The commands in `launch.json` and `tasks.json` will essentially be the same except `launch.json` contains config information for the debugger, and the `tasks.json` file will configure shell command that allows for a quick check for whether we see what we expect or not
- Once I'm happy with what I have so far, I start soliciting reviews on Phabricator
- After an iterative process, the fix was deemed acceptable but contingent on writing some tests in the `flang/test` directory

VSCode Configuration and Tips for Flang Development

Anthony Cabrera

Research Scientist, Architectures and Performance Group, CSMD
Oak Ridge National Laboratory
January 4, 2023