

# Debugging

2023.03.16

SWPP Practice Session

Seunghyeon Nam

# Typical Bugfix Process

- Notice an error
- **Narrow down** the line that causes the error
  - If the program crashes, look for the assertion or invalid pointer
  - If the program yields wrong output, look for the output variable
- **Traceback** to the line that started to go wrong

# Narrowing Down the Line

- How do you locate the exact line that crashes?
  - **Guess** the location
  - Insert some `std::cout` or `std::cerr` all over the code
  - **Rebuild**
  - Look for the last printed message
  - **Repeat** until you actually pinpoint the location

# Narrowing Down the Line

- This is horribly inefficient!
  - Taking a wild guess in a large codebase purely depends on luck
  - Rebuilding a large codebase may take minutes or even hours
  - And you have to repeat it until you actually find the bug
- Locating a single bug **may already take hours or days**

# Traceback

- Most of the errors cannot be fixed locally
  - It is likely that the code that 'triggers' the error is not a bug
  - The code that 'leads to' the error is the real verdict
  - But these two are usually far away from each other...
- You have to locate the code that **first** went wrong
  - Narrow down, take a step back, narrow down, again and again

# Debugger to the Rescue

- Debugger can control the execution of your program
  - Line by line
  - In and out of function
  - Pause on assertion, throw, catch, breakpoint

# Debugger to the Rescue

- Debugger can expose the execution context of your program
  - Call stack
  - Local/global variables and values

# Using the Debugger

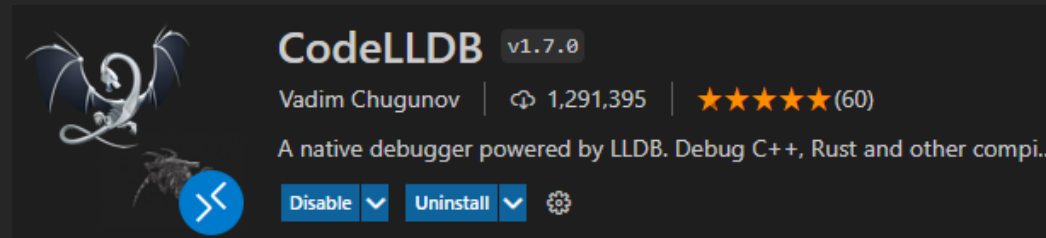
- LLDB: LLVM debugger
  - You have to enable LLDB project when building the LLVM
  - Already included if you built the LLVM using class repo script
- You must build your program **with clang**
- You must build your program **in debug mode**



# Using the Debugger

- We'll use vscode extension for convenience

- CodeLLDB



- LLDB directory should be added to your PATH
  - What is PATH?

File Edit Selection View Go Run Terminal Help

inst.cpp - swpp-compiler [S] /visual Studio Code

RUN AND DEBUG

RUN

Run and Debug

To customize Run and Debug create a launch.json file.

Show all automatic debug configurations.

To learn more about launch.json, see Configuring C/C++ debugging.

src > lib.cpp > {} 'anonymous-namespace' > Inputf

std::string message;

public:

InputFileError(sc::parser::ParserError & err) {

using namespace std::string\_literals;

message = "invalid input file\n"s.append(\_\_err.what());

}

InputFileError(fs::FilesystemError &&\_err) noexcept {

using namespace std::string\_literals;

message = "invalid input file\n"s.append(\_\_err.what());

}

const char \*what() const noexcept { return message.c\_str(); }

};

You, last month | 1 author (You)

class OutputFileError : public Error<OutputFileError> {

private:

std::string message;

public:

OutputFileError(const std::string\_view \_\_message) {

using namespace std::string\_literals;

message = "invalid output file\n"s;

message.append(\_\_message);

}

const char \*what() const noexcept { return message.c\_str(); }

};

Result<std::string, InputFileError>

readFile(const std::string\_view \_\_filename) {

auto read\_result = fs::readFile(\_\_filename);

return decltype(read\_result)::mapErr<InputFileError>(<

std::move(read\_result),

[auto &&err] { return InputFileError(std::move(err)); });

}

select environment

C++ (GDB/LLDB)

C++ (Windows)

LLDB

Install an extension for C++...

assembly > inst.cpp > {} sc > {} backend > {} assembly > {} inst

locInst

MallocInst(const GeneralRegister \_\_target,

ValueTy &&\_\_size) noexcept

: AbstractInst(), target(\_\_target), size(std::move(\_\_size)) {}

MallocInst MallocInst::create(const GeneralRegister \_\_target,

ValueTy &&\_\_size) noexcept {

return MallocInst(\_\_target, std::move(\_\_size));

}

std::string MallocInst::getAssembly() const noexcept {

return joinTokens(prependTarget(target, collectOpTokens("malloc"s, size)));

}

You, 2 months ago • Init ...

//-----

// class FreeInst

//-----

FreeInst::FreeInst(ValueTy &&\_\_ptr) noexcept

: AbstractInst(), ptr(std::move(\_\_ptr)) {}

FreeInst FreeInst::create(ValueTy &&\_\_ptr) noexcept {

return FreeInst(std::move(\_\_ptr));

}

std::string FreeInst::getAssembly() const noexcept {

return joinTokens(collectOpTokens("free"s, ptr));

}

//-----

// class LoadInst

//-----

LoadInst::LoadInst(const GeneralRegister \_\_target, const AccessWidth \_\_size,

ValueTy &&\_\_ptr) noexcept

: AbstractInst(), target(\_\_target), size(\_\_size), ptr(std::move(\_\_ptr)) {}

PROBLEMS 8 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

swpp-compiler main

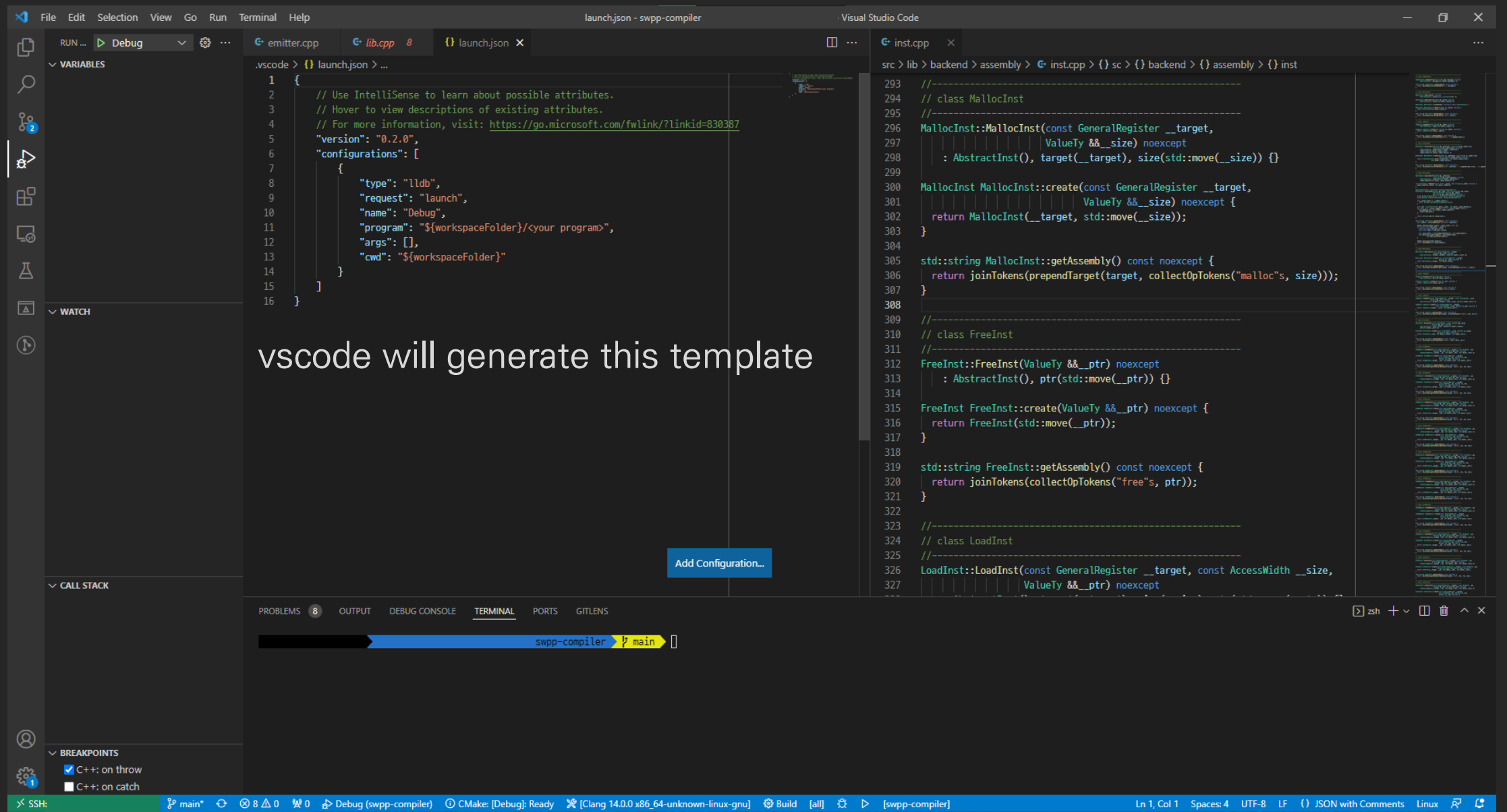
SSH: main\* 8 0 0 CMake: [Debug]: Ready [Clang 14.0.0 x86\_64-unknown-linux-gnu] Build [all] [swpp-compiler] You, 2 months ago Ln 308, Col 1 Spaces: 2 UTF-8 LF C++ Linux

Select LLDB from the options

Visual Studio Code interface showing a C++ project named "swpp-compiler". The editor displays two files: `lib.cpp` and `inst.cpp`. The `lib.cpp` file contains C++ code for error handling and file reading. The `inst.cpp` file contains assembly code for memory allocation and deallocation.

A modal dialog box is displayed in the center of the screen with the text: "You'll get this error on the first run". The dialog box has a red "X" icon and the text: "Cannot start debugging because no launch configuration has been provided." with an "OK" button.

The bottom status bar shows the current configuration: "SSH: main", "8 0", "CMake: [Debug]: Ready", "Clang 14.0.0 x86\_64-unknown-linux-gnu", "Build [all]", and "swpp-compiler". The bottom right corner shows the file path: "You, 2 months ago Ln 25, Col 4 Spaces: 2 UTF-8 LF C++ Linux".



Visual Studio Code interface showing the configuration of a debug target in `launch.json`.

The `launch.json` file is open, showing the configuration for a debug target named "Debug". The configuration is as follows:

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "lldb",
      "request": "launch",
      "name": "Debug",
      "program": "${workspaceFolder}/build/swpp-compiler",
      "args": ["benchmarks/anagram/src/anagram.ll", "test.s"],
      "cwd": "${workspaceFolder}"
    }
  ]
}
```

The text "Fill in the program and args. Then run the debugger" is overlaid on the configuration.

The `inst.cpp` file is also open, showing the implementation of the `MallocInst` and `FreeInst` classes.

The terminal shows the command `swpp-compiler main` being executed.

The status bar at the bottom indicates the current configuration is `Debug (swpp-compiler)` and the compiler is `Clang 14.0.0 x86_64-unknown-linux-gnu`.

Visual Studio Code interface showing a C++ project named "eh\_throw.cc - swpp-compiler". The main editor displays the source code of "eh\_throw.cc", which is paused at a breakpoint at line 77. The code defines a custom exception handler and a primary exception class. The left sidebar shows the "VARIABLES" pane with local variables like "obj", "tinfo", and "dest". The "WATCH" pane is empty. The "CALL STACK" pane shows the current function call stack. The bottom status bar indicates the current state of the debugger and the active file.

### Debugger paused at throw

```
60 {
61     _cxa_refcounted_exception *header
62     = __get_refcounted_exception_header_from_obj (obj);
63     header->referenceCount = 0;
64     header->exc.exceptionType = tinfo;
65     header->exc.exceptionDestructor = dest;
66     header->exc.unexpectedHandler = std::get_unexpected ();
67     header->exc.terminateHandler = std::get_terminate ();
68     __GXX_INIT_PRIMARY_EXCEPTION_CLASS(header->exc.unwindHeader.exception_class);
69     header->exc.unwindHeader.exception_cleanup = __gxx_exception_cleanup;
70
71     return header;
72 }
73
74 extern "C" void
75 __cxxabiv1::__cxa_throw (void *obj, std::type_info *tinfo,
76                          void (_GLIBCXX_CDTOR_CALLABI *dest) (void *))
77 {
78     PROBE2 (throw, obj, tinfo);
79     __cxa_throw(obj, tinfo, dest);
80     globals->uncaughtExceptions += 1;
81     // Definitely a primary.
82     _cxa_refcounted_exception *header =
83     __cxa_init_primary_exception(obj, tinfo, dest);
84     header->referenceCount = 1;
85
86     #ifdef __USING_SJLJ_EXCEPTIONS__
87     _Unwind_SjLj_RaiseException (&header->exc.unwindHeader);
88     #else
89     _Unwind_RaiseException (&header->exc.unwindHeader);
90     #endif
91
92     // Some sort of unwinding error. Note that terminate is a handler.
93     __cxa_begin_catch (&header->exc.unwindHeader);
94 }
```

Console is in 'commands' mode, prefix expressions with '?'.  
Launching: swpp-compiler/build/swpp-compiler benchmarks/anagram/src/anagram.ll test.s  
Launched process 2701418

**VARIABLES in current stack entry**

**CALL STACK**

**Navigate call stack**

**eh\_throw.cc**

```
60 libstdc++.v3 > libsupc++ > eh_throw.cc > _cxa_throw(void *, std::type_info *, void(_GLIBCXX_CDTOR
61   __cxa_refcounted_exception *header
62   = __get_refcounted_exception_header_from_obj (obj);
63   header->referenceCount = 0;
64   header->exc.exceptionType = tinfo;
65   header->exc.exceptionDestructor = dest;
66   header->exc.unexpectedHandler = std::get_unexpected ();
67   header->exc.terminateHandler = std::get_terminate ();
68   __GXX_INIT_PRIMARY_EXCEPTION_CLASS(header->exc.unwindHeader.exception_class);
69   header->exc.unwindHeader.exception_cleanup = __gxx_exception_cleanup;
70
71   return header;
72 }
73
74 extern "C" void
75 __cxa_biv1::__cxa_throw (void *obj, std::type_info *tinfo,
76   void (_GLIBCXX_CDTOR_CALLABI *dest) (void *))
77 {
78   PROBE2 (throw, obj, tinfo);
79
80   __cxa_eh_globals *globals = __cxa_get_globals ();
81   globals->uncaughtExceptions += 1;
82   // Definitely a primary.
83   __cxa_refcounted_exception *header =
84     __cxa_init_primary_exception(obj, tinfo, dest);
85   header->referenceCount = 1;
86
87   #ifdef __USING_SJLJ_EXCEPTIONS__
88     __Unwind_SjLj_RaiseException (&header->exc.unwindHeader);
89   #else
90     __Unwind_RaiseException (&header->exc.unwindHeader);
91   #endif
92
93   // So, sort of unwinding error. Note that terminate is a handler.
94   // So, sort of unwinding error. Note that terminate is a handler.
```

**inst.cpp**

```
293 //-----
294 // class MallocInst
295 //-----
296 MallocInst::MallocInst(const GeneralRegister __target,
297   ValueTy &&__size) noexcept
298   : AbstractInst(), target(__target), size(std::move(__size)) {}
299
300 MallocInst MallocInst::create(const GeneralRegister __target,
301   ValueTy &&__size) noexcept {
302   return MallocInst(__target, std::move(__size));
303 }
304
305 std::string MallocInst::getAssembly() const noexcept {
306   return joinTokens(prependTarget(target, collectOpTokens("malloc"s, size)));
307 }
308
309 //-----
310 // class FreeInst
311 //-----
312 FreeInst::FreeInst(ValueTy &&__ptr) noexcept
313   : AbstractInst(), ptr(std::move(__ptr)) {}
314
315 FreeInst FreeInst::create(ValueTy &&__ptr) noexcept {
316   return FreeInst(std::move(__ptr));
317 }
318
319 std::string FreeInst::getAssembly() const noexcept {
320   return joinTokens(collectOpTokens("free"s, ptr));
321 }
322
323 //-----
324 // class LoadInst
325 //-----
326 LoadInst::LoadInst(const GeneralRegister __target, const AccessWidth __size,
327   ValueTy &&__ptr) noexcept
```

**Console**

```
Console is in 'commands' mode, prefix expressions with '?'.
Launching: swpp-compiler/build/swpp-compiler benchmarks/anagram/src/anagram.ll test.s
Launched process 2701418
```







Visual Studio Code interface showing a C++ project named "eh\_throw.cc - swpp-compiler". The editor displays the source code of "eh\_throw.cc" and "inst.cpp". A yellow box highlights the "Execute next line" button (a blue square with a white circular arrow) in the top toolbar. Another yellow box highlights the same button in the bottom toolbar. The text "Execute next line" is overlaid on the editor area.

The left sidebar shows the "VARIABLES" panel with the following content:

```
VARIABLES
  Local
    > obj: 0x000000000242f90
    > tinfo: <invalid address>
    > dest: (libSCBackend.so` (anony-
    > globals: <variable not availa-
    > header: <variable not availab-
```

The "WATCH" panel shows the following content:

```
WATCH
  D 77 {
    PROBE2 (throw, obj, tinfo);
    _cxa_eh_globals *globals = _cxa_get_globals ();
    globals->uncaughtExceptions += 1;
    // Definitely a primary.
    _cxa_refcounted_exception *header =
    _cxa_init_primary_exception(obj, tinfo, dest);
    header->referenceCount = 1;
    #ifdef __USING_SJLJ_EXCEPTIONS__
    _Unwind_SjLj_RaiseException (&header->exc.unwindHeader);
    #else
    _Unwind_RaiseException (&header->exc.unwindHeader);
    #endif
    // Some sort of unwinding error. Note that terminate is a handler.
    _cxa_begin_catch (&header->exc.unwindHeader);
```

The "CALL STACK" panel shows the following content:

```
CALL STACK
  PAUSED ON BREAKPOINT
    _cxa_throw(void *,
    std::__cxx11::basic_string<char,
    sc::backend::emitter::AssemblyE
    llvm::InstVisitor<sc::backend::e
    void llvm::InstVisitor<sc::backe
    llvm::InstVisitor<sc::backend::e
    sc::backend::emitAssembly[abi:c
    (anonymous namespace)::compile(s
    result::Result<std::__cxx11::bas
```

The "BREAKPOINTS" panel shows the following content:

```
BREAKPOINTS
  [x] C++: on throw
  [ ] C++: on catch
```

The "MODULES" panel shows the following content:

```
MODULES
  >
```

The "DEBUG CONSOLE" panel shows the following content:

```
DEBUG CONSOLE
  Console is in 'commands' mode, prefix expressions with '?'.
  Launching:
  Launched process 2701418
```

The status bar at the bottom shows the following information:

```
SSH: main* 0 0 0 0 2 0 0 Debug (swpp-compiler) CMake: [Debug]: Ready [Clang 14.0.0 x86_64-unknown-linux-gnu] Build [all] [swpp-compiler] Format: auto Disasm: auto Deref: on Console: cmd Ln 77, Col 1 Spaces: 2 UTF-8 LF C++ Linux
```

Visual Studio Code interface showing a C++ project named "eh\_throw.cc - swpp-compiler". The editor displays the source code of "eh\_throw.cc" and "inst.cpp". A yellow box highlights the "Run and Debug" icon in the top toolbar, and another yellow box highlights the "Step Into" icon (a blue arrow pointing down) in the "Run and Debug" toolbar. A large text overlay reads: "Step in (Get inside the function at current line & execute the first line of that function)".

The left sidebar shows the "VARIABLES" pane with local variables like `obj`, `tinfo`, `dest`, `globals`, and `header`. The "WATCH" pane shows `__cxxabiv1::__cxa_throw(void *, ...)`. The "CALL STACK" pane shows the current function call: `__cxxabiv1::__cxa_throw(void *, ...)`. The "BREAKPOINTS" pane shows a breakpoint set for "C++: on throw".

The bottom status bar shows the current file is `eh_throw.cc`, line 77, column 1. The status bar also indicates the current build system is "CMake: [Debug]: Ready" and the compiler is "Clang 14.0.0 x86\_64-unknown-linux-gnu".

Visual Studio Code interface showing a C++ project named "eh\_throw.cc - swpp-compiler". The editor displays the source code of "eh\_throw.cc" and "inst.cpp". A yellow box highlights the "Step Out" button (represented by a blue arrow pointing up) in the top toolbar. A large text overlay reads: "Step out (Execute until the end of current function & execute the next line of the caller)".

The left sidebar shows the "VARIABLES" panel with local variables like "obj", "tinfo", "dest", "globals", and "header". The "WATCH" panel shows the expression "std::move(obj)". The "CALL STACK" panel shows the current function "std::move(obj)" and its caller "std::move(obj)". The "BREAKPOINTS" panel shows a breakpoint set at line 77.

The bottom status bar shows the current file is "eh\_throw.cc" at line 77, column 1. The status bar also indicates the project is "Debug (swpp-compiler)" and the compiler is "Clang 14.0.0 x86\_64-unknown-linux-gnu".



# Narrowing Down with Debugger

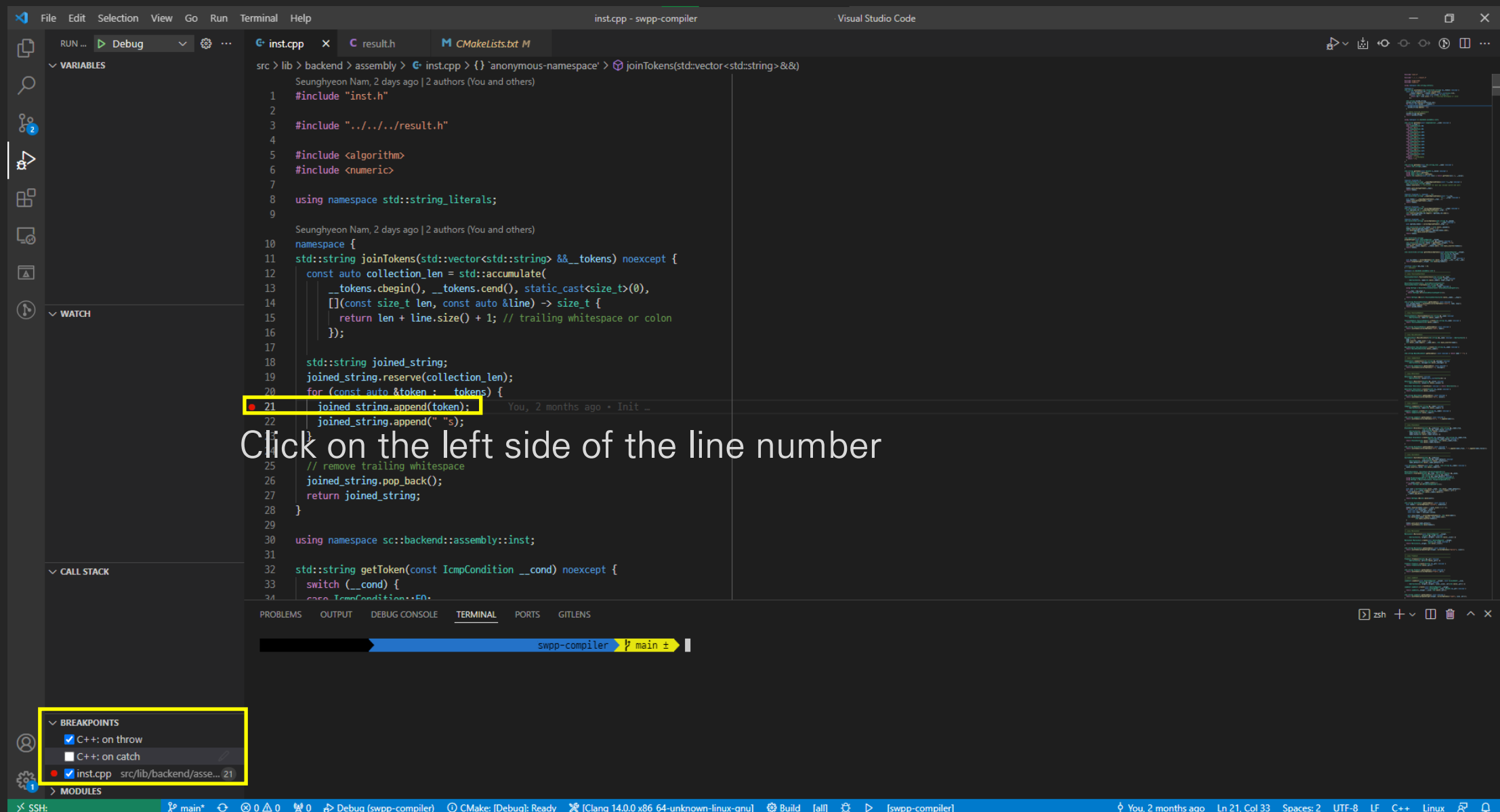
- How do you locate the exact line that crashes?
  - Debugger pinpoints (automatically pauses on) the crash site
  - No need to insert new code, rebuild, etc.

# Traceback with Debugger

- You have to locate the code that **first** went wrong
  - **Debugger shows you the call stack at the moment of the crash**
  - Clicking is all you need to navigate through the call stacks
  - Find the first call stack with unexpected value or control flow

# Traceback with Debugger

- Sometimes you have to **monitor the change** of values
  - If your code does not throw or crash, debugger won't pause
  - You can use **breakpoint** to pause execution at a certain point





Visual Studio Code interface showing a C++ file named `inst.cpp` in the `src/lib/backend/assembly` directory. The code is paused on a breakpoint at line 21, which is highlighted in yellow. The breakpoint is located at the line `joined_string.append(D token);`.

The code in `inst.cpp` includes headers `inst.h` and `result.h`, and uses `std::string` and `std::vector`. It defines a function `joinTokens` that takes a vector of strings and returns a single string. The function uses `std::accumulate` to calculate the total length of the concatenated strings, including trailing whitespace or colons. It then reserves space for the result string and appends each token from the input vector.

The `CALL STACK` pane shows the current function `(anonymous namespace)::joinToken` and its caller `sc::backend::assembly::inst::Fur`. The `BREAKPOINTS` pane shows the active breakpoint at `inst.cpp` line 21.

The `DEBUG CONSOLE` pane shows the output of the program, including the text `Seunghyeon Nam, 2 days ago | 2 authors (You and others)` and the output of the `joinTokens` function.

The `TERMINAL` pane shows the command prompt with the text `zsh`.

The status bar at the bottom indicates the current file is `inst.cpp` at line 21, column 26, with a UTF-8 encoding and LF line endings. The editor is running on a Linux system with the C++ compiler.

# IR Visualization

- Visualizing the control flow of your IR program can be helpful

```
define dso_local i32 @main() #0 {
entry:
  store i64* null, i64** @root, align 8
  %call = call i64 (...) @read()
  br label %for.cond

for.cond:                                ; preds = %for.inc, %entry
  %i.0 = phi i64 [ 0, %entry ], [ %inc, %for.inc ]
  %cmp = icmp ult i64 %i.0, %call
  br i1 %cmp, label %for.body, label %for.cond.cleanup

for.cond.cleanup:                        ; preds = %for.cond
  br label %for.end

for.body:                                ; preds = %for.cond
  %call1 = call i64 (...) @read()
  %call2 = call i64 (...) @read()
  %cmp3 = icmp eq i64 %call1, 0
  br i1 %cmp3, label %if.then, label %if.else

if.then:                                 ; preds = %for.body
  %call4 = call i64 @insert(i64 %call2)
  br label %if.end

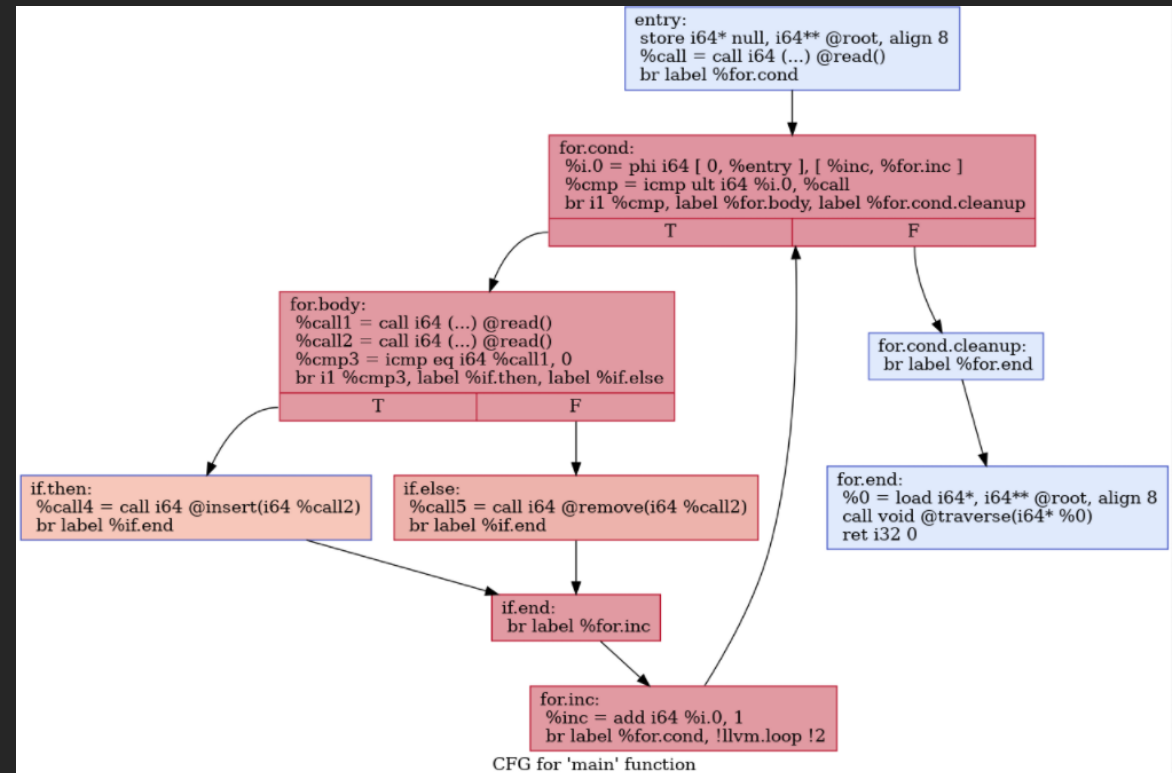
if.else:                                  ; preds = %for.body
  %call5 = call i64 @remove(i64 %call2)
  br label %if.end

if.end:                                   ; preds = %if.else, %if.then
  br label %for.inc

for.inc:                                  ; preds = %for.end, %if.end
  %inc = add i64 %i.0, 1
  br label %for.cond, !llvm.loop !2

for.cond.cleanup:
  br label %for.end

for.end:
  %0 = load i64*, i64** @root, align 8
  call void @traverse(i64* %0)
  ret i32 0
```



# IR Visualization

- Install GraphViz
  - Use the package manager to handle dependencies for you
- Run `<llvm-dir>/opt --dot-cfg <IR-program.ll>`
  - You'll get a .dot file for each function in the program
- Run `dot <dot-file.dot> -Tpng -o <image-name.png>`