

# Debugging

2022.04.25

SWPP Practice Session

Seunghyeon Nam (with lots of derived works)

# 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 'causes' 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

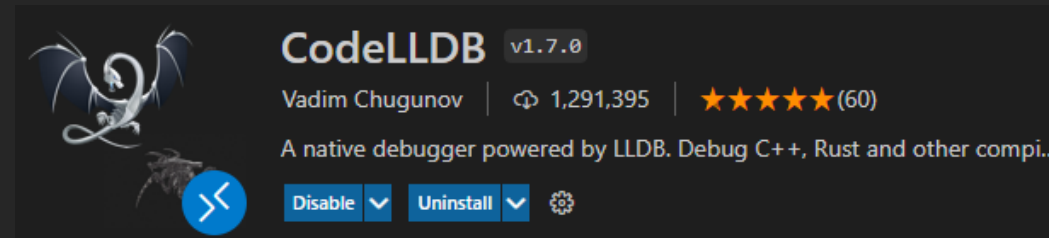
- LLDB: LLVM debugger
  - You have to enable LLDB project when building the LLVM
  - Already included if you built the LLVM using class repo script
- You must build your program **with clang**
- You must build your program **in debug mode**



# 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) noexcept {

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 source files: `lib.cpp` and `inst.cpp`.

The `lib.cpp` file contains the following code:

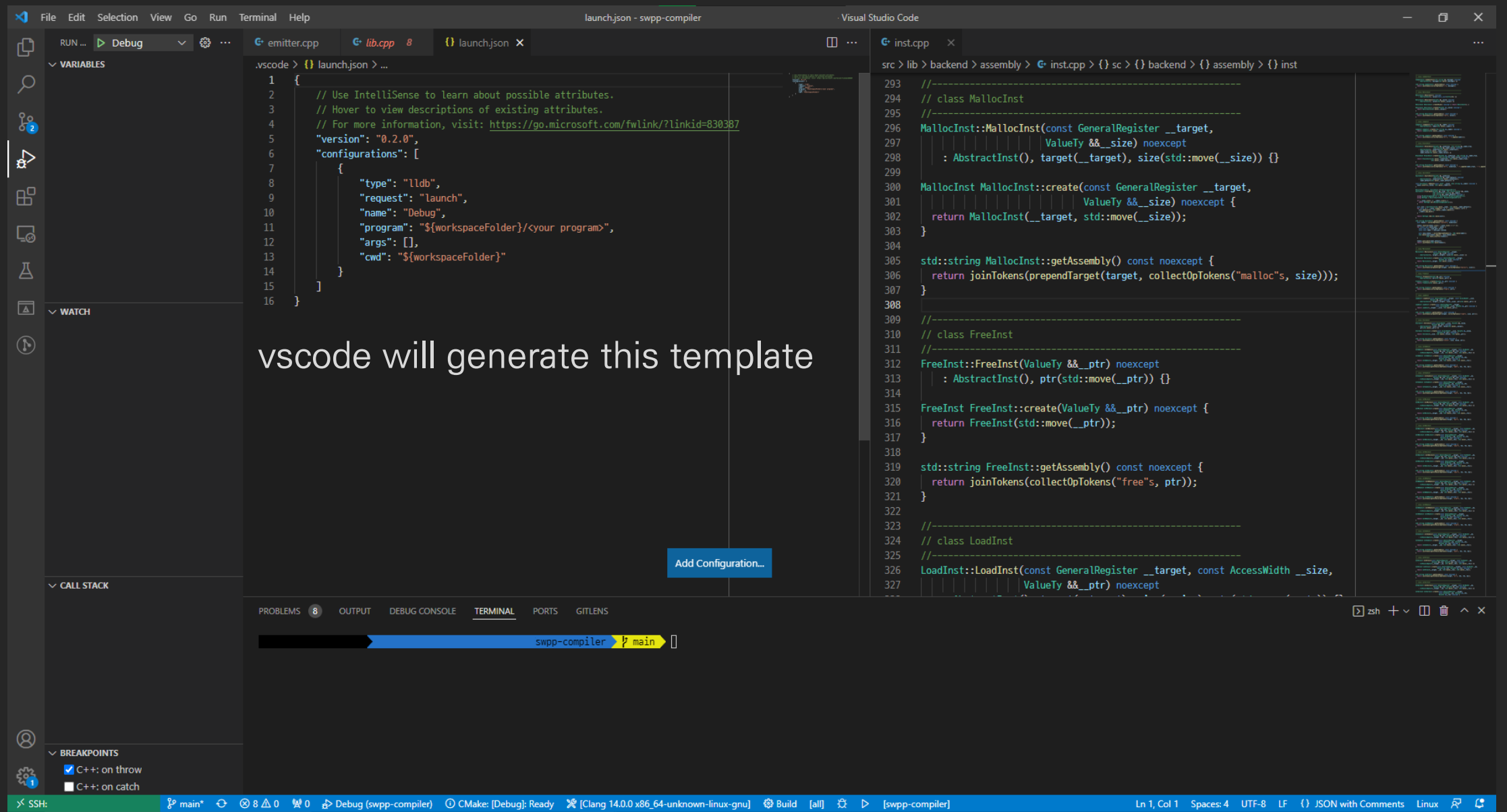
```
src > lib.cpp > {} 'anonymous-namespace' > InputFileError > InputFileError(sc::parser::ParserError &&)\n19 std::string message;\n20\n21 public:\n22 InputFileError(sc::parser::ParserError &&__err) noexcept {\n23     using namespace std::string_literals;\n24     message = \"invalid input file\\n\"s.append(__err.what());\n25     You, 2 months ago • Init ...\n26\n27 InputFileError(fs::FilesystemError &&__err) noexcept {\n28     using namespace std::string_literals;\n29     message = \"invalid input file\\n\"s.append(__err.what());\n30 }\n31\n32 const char *what() const noexcept { return message.c_str(); }\n33\n34\n35 You, last month | 1 author (You)\n36\n37 class OutputFileError : public Error<OutputFileError> {\n38 private:\n39     std::string message;\n40\n41 public:\n42 OutputFileError(const std::string_view __message)\n43     using namespace std::string_literals;\n44     message = \"invalid output file\\n\"s;\n45     message.append(__message);\n46\n47     const char *what() const noexcept { return message.c_str(); }\n48\n49 Result<std::string, InputFileError>\n50 readFile(const std::string_view __filename) {\n51     auto read_result = fs::readFile(__filename);\n52     return decltype(read_result)::mapErr<InputFileError>(\n53         std::move(read_result),\n54         [](auto &&err) { return InputFileError(std::move(err)); });\n55 }
```

The `inst.cpp` file contains the following code:

```
src > lib > backend > assembly > inst.cpp > {} sc > {} backend > {} assembly > {} inst\n293 //-----\n294 // class MallocInst\n295 //-----\n296 MallocInst::MallocInst(const GeneralRegister __target,\n297     ValueTy &&__size) noexcept\n298     : AbstractInst(), target(__target), size(std::move(__size)) {}\n299\n300 MallocInst MallocInst::create(const GeneralRegister __target,\n301     ValueTy &&__size) noexcept {\n302     return MallocInst(__target, std::move(__size));\n303 }\n304\n305 std::string MallocInst::getAssembly() const noexcept {\n306     return joinTokens(prependTarget(target, collectOpTokens(\"malloc\"s, size))); \n307 }\n308\n309 //-----\n310 // class FreeInst\n311\n312 FreeInst::FreeInst(ValueTy &&__ptr) noexcept\n313     : AbstractInst(), ptr(std::move(__ptr)) {}\n314\n315 FreeInst FreeInst::create(ValueTy &&__ptr) noexcept {\n316     return FreeInst(std::move(__ptr));\n317 }\n318\n319 std::string FreeInst::getAssembly() const noexcept {\n320     return joinTokens(collectOpTokens(\"free\"s, ptr));\n321 }\n322\n323 //-----\n324 // class LoadInst\n325 //-----\n326 LoadInst::LoadInst(const GeneralRegister __target, const AccessWidth __size,\n327     ValueTy &&__ptr) noexcept
```

A 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 an "OK" button.

The terminal at the bottom shows the command `swpp-compiler` being executed, with the output `main`.



Visual Studio Code interface showing the configuration of a debugger for a C++ project named "swpp-compiler".

The **launch.json** file is open, showing the configuration for a **Debug** target. 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 **inst.cpp** file is also open, showing the implementation of the **MallocInst** and **FreeInst** classes.

The **Terminal** panel at the bottom shows the output of the debugger, indicating that the program is running successfully.

**Fill in the program and args.**  
**Then run the debugger**

**Add Configuration...**

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. The code defines a custom exception handler and a primary exception class.

```
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 }
```

The left sidebar shows the "VARIABLES" pane with the following information:

- obj:** 0x000000000242f90
- tinfo:** <invalid address>
- dest:** (libSCBackend.so` (anonymous namespace):`\_\_cxa\_throw` (void \*, std::type\_info \*, void \*)@0x000000000242f90)
- globals:** <variable not available>
- header:** <variable not available>

The "WATCH" pane is empty.

The "CALL STACK" pane shows the current call stack:

- \_\_cxxabiv1::\_\_cxa\_throw (void \*, std::type\_info \*, void \*)@0x000000000242f90

The "PROBLEMS" pane shows the following message:

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

The "DEBUG CONSOLE" pane shows the following output:

```
swpp-compiler/build/swpp-compiler benchmarks/anagram/src/anagram.ll test.s
Launched process 2701418
```

The "BREAKPOINTS" pane shows the following breakpoints:

- ☒ C++: on throw
- ☐ C++: on catch

The "MODULES" pane shows the following modules:

- main\*
- 0 0 0 0 2
- 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 main editor displays the source code of "eh\_throw.cc", which includes exception handling logic and assembly generation. The code is written in C++ and uses the `std::type_info` and `std::move` functions. A yellow box highlights the "VARIABLES" panel on the left, showing the current stack entry for the function `__cxxabiv1::__cxa_throw`. The variables listed are `obj` (address `0x000000000242f90`), `tinfo` (invalid address), `dest` (libSCBackend.so), `globals` (variable not available), and `header` (variable not available). Another yellow box highlights the "CALL STACK" panel, showing the call stack with the current frame being `__cxxabiv1::__cxa_throw`. The "WATCH" panel is also visible, showing the value of `__cxa_eh_globals` as `__cxa_get_globals()`. The "DEBUG CONSOLE" at the bottom shows the output of the program, including the launch command and the process ID `2701418`. The status bar at the bottom indicates the current configuration: `SSH: main* | 0 0 0 0 2 | 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`.

## Variables in current stack entry

**VARIABLES**

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

## Navigate call stack

**CALL STACK** PAUSED ON BREAKPOINT

- `__cxxabiv1::__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`

**DEBUG 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.

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. 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 with a breakpoint set in `eh_throw.cc`. The code is being debugged, and the call stack shows the current function and its caller.

**Step out**  
(Execute until the end of current function & execute the next line of the caller)

The image shows the Visual Studio Code editor with a C++ project. The main editor window displays `eh_throw.cc` with a breakpoint set at line 77. The call stack on the left shows the current function and its caller. The debug console at the bottom shows the output of the program, including the launch command and the process ID.

**Call Stack:**

- `std::__cxx11::basic_string<char, std::backends::emitter::AssemblyE...`
- `llvm::InstVisitor<sc::backends::e...`
- `void llvm::InstVisitor<sc::backends::e...`
- `sc::backends::emitAssembly[abi:c...`
- `(anonymous namespace)::compile(s...`
- `result::Result<std::__cxx11::bas...`

**Debug 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 being debugged. The main editor displays the source code of `eh_throw.cc` and `inst.cpp`. The left sidebar shows the **VARIABLES** and **WATCH** panels. The bottom status bar indicates the current state: **SSH: main**, **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**.

The **DEBUG CONSOLE** at the bottom shows the following output:

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

The **DEBUG CONSOLE** also shows the following output:

```
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 **DEBUG CONSOLE** also shows the following output:

```
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
```

# 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 defines a `joinTokens` function that concatenates tokens from a vector into a single string, removing trailing whitespace.

```
1 #include "inst.h"
2
3 #include "../result.h"
4
5 #include <algorithm>
6 #include <numeric>
7
8 using namespace std::string_literals;
9
10 namespace {
11     std::string joinTokens(std::vector<std::string> &&_tokens) noexcept {
12         const auto collection_len = std::accumulate(
13             __tokens.cbegin(), __tokens.cend(), static_cast<size_t>(0),
14             [](const size_t len, const auto &line) -> size_t {
15                 return len + line.size() + 1; // trailing whitespace or colon
16             });
17
18         std::string joined_string;
19         joined_string.reserve(collection_len);
20         for (const auto &token : _tokens) {
21             joined_string.append(token);
22             joined_string.append(" ");
23         }
24         // remove trailing whitespace
25         joined_string.pop_back();
26         return joined_string;
27     }
28 }
29
30 using namespace sc::backend::assembly::inst;
31
32 std::string getToken(const IcmpCondition __cond) noexcept {
33     switch (__cond) {
34         case IcmpCondition::E0:
```

The left sidebar shows the **BREAKPOINTS** panel, which is highlighted with a yellow box. It contains the following settings:

- ☒ C++: on throw
- ☐ C++: on catch
- ☒ inst.cpp src/lib/backend/asse... (21)

The bottom status bar indicates the current file is `inst.cpp` at line 21, column 33, with a UTF-8 encoding and LF line endings. The terminal shows the `swpp-compiler` process running.



Visual Studio Code interface showing a C++ file named `inst.cpp` in the `src/lib/backend/assembly` directory. The code defines a `joinTokens` function that concatenates tokens from a vector into a single string, handling trailing whitespace and colons. A breakpoint is set at line 21, where `joined_string.append(D token);` is executed. The text "Paused on breakpoint" is overlaid on the code.

The left sidebar shows the **VARIABLES** pane with the following data:

- Local**
  - `__tokens`: size=3
  - `collection_len`: 14
  - `joined_string`: error: summary\_
  - `token`: {\_M\_string\_length:5}
- Static**
- Global**
- Registers**

The **WATCH** pane is empty.

The **CALL STACK** pane shows the current call stack, with the top frame being `(anonymous namespace)::joinToker`.

The **BREAKPOINTS** pane shows the following breakpoints:

- ☒ C++: on throw
- ☐ C++: on catch
- ☒ inst.cpp src/lib/backend/asse... 21

The **TERMINAL** pane shows the output of the `zsh` shell.

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 active editor is `inst.cpp` in the `src/lib/backend/assembly` directory.

# 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

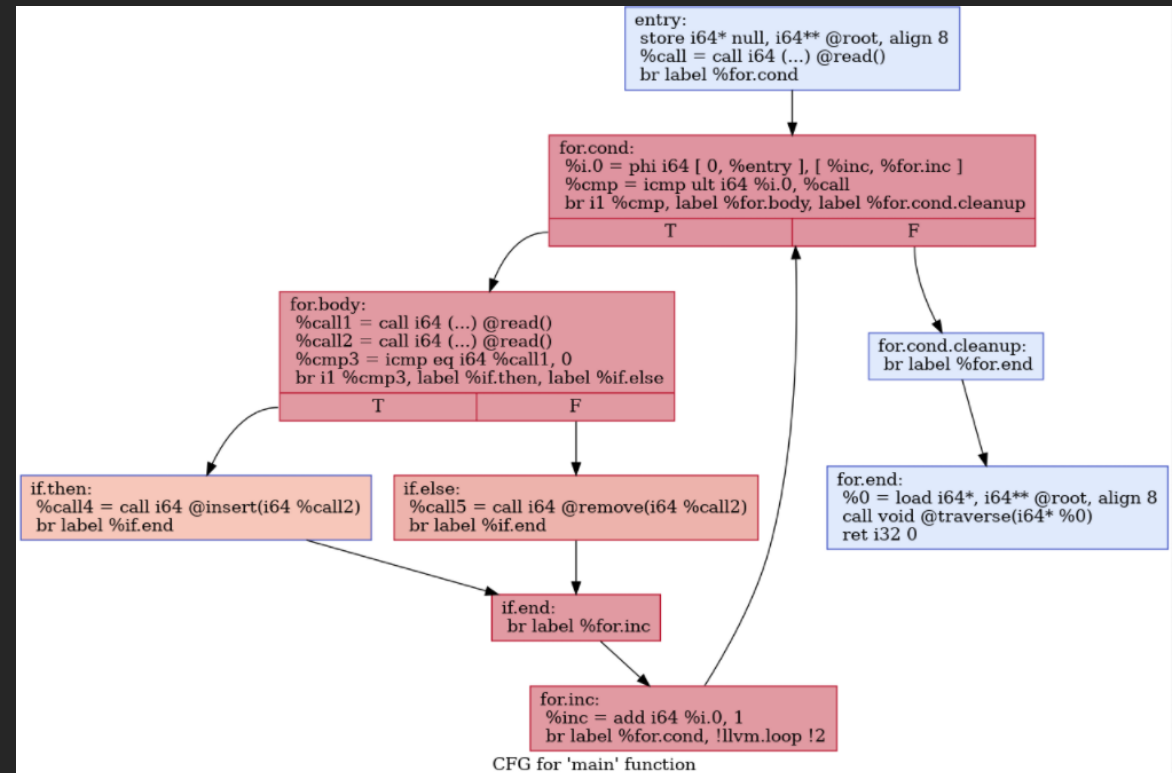
if.else:                                  ; preds = %for.body
  %call5 = call i64 @remove(i64 %call2)
  br label %if.end

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>`