

Debugging

2024.03.14

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

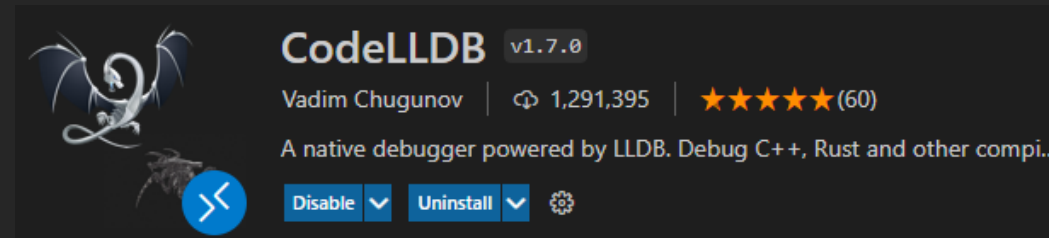
Using the Debugger

- 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]

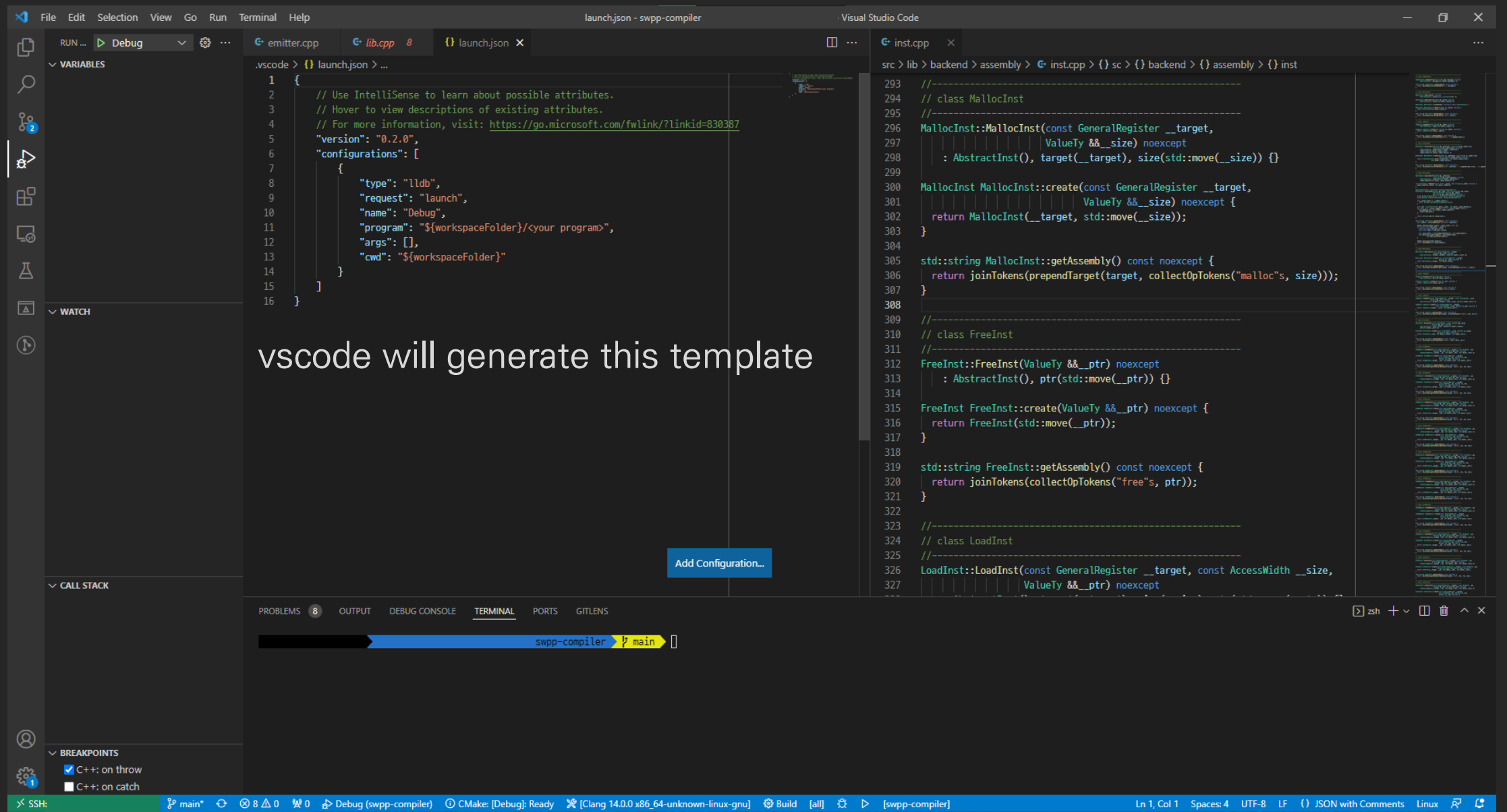
You, 2 months ago Ln 308, Col 1 Spaces: 2 UTF-8 LF C++ Linux

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]", "[swpp-compiler]", and "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 overlay indicates: "Fill in the **program** and **args**. Then run the debugger".

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.

Buttons for "Add Configuration..." and "Run" are visible.

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

- Local
 - obj: 0x00000000242f90
 - tinfo: <invalid address>
 - dest: (libSCBackend.so`anony...
 - globals: <variable not availa...
 - header: <variable not availab...
- Static
- Global
- Registers

WATCH

- eh_throw.cc:61: __cxa_rethrow_exception *header

CALL STACK

- eh_throw.cc:61: __cxa_rethrow_exception *header

BREAKPOINTS

- eh_throw.cc:61: __cxa_rethrow_exception *header

PROBLEMS

- eh_throw.cc:61: __cxa_rethrow_exception *header

OUTPUT

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

DEBUG CONSOLE

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

TERMINAL

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

PORTS

- swpp-compiler

GIT LENS

- eh_throw.cc:61: __cxa_rethrow_exception *header

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 a function `__cxxabiv1::__cxa_throw`. A yellow box highlights the "Execute until next breakpoint" button (a blue play icon with a right-pointing arrow) in the top toolbar.

The left sidebar shows the "VARIABLES" pane with local variables like `obj`, `tinfo`, `dest`, `globals`, and `header`. The "WATCH" pane is empty. The "CALL STACK" pane shows the current function `__cxxabiv1::__cxa_throw` and its caller `std::__cxx11::basic_string<char, ...>`.

The bottom status bar indicates the current state: "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".

A yellow box highlights the "BREAKPOINTS" pane in the bottom left, showing two breakpoints: "C++: on throw" (checked) and "C++: on catch" (unchecked).

Overlaid on the center of the image is the text "Execute until next breakpoint" in a large, white, sans-serif font.

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 local variables: `obj` (0x000000000242f90), `tinfo` (invalid address), `dest` (libSCBackend.so), `globals` (variable not available), and `header` (variable not available). The "WATCH" panel shows the expression `__cxxabiv1: __cxa_throw(void *, std::type_info *, void(GLIBCXX_CTOR_CALLABI *dest) (void *))`. The "CALL STACK" panel shows the current function `__cxa_throw` and its caller `__cxa_throw`. The "BREAKPOINTS" panel 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 project is using "CMake: [Debug]: Ready" and "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 "Run and Debug" icon in the top toolbar. A large text overlay reads: "Step in (Get inside the function at current line & execute the first line of that function)". The "WATCH" panel shows the current state of variables, including "obj", "tinfo", "dest", "globals", and "header". The "CALL STACK" panel shows the current function call stack, with "eh_throw.cc" at the top. The "DEBUG CONSOLE" panel shows the output of the program, including the launch command and the process ID.

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. A large text overlay reads: "Step in (Get inside the function at current line & execute the first line of that function)". The "WATCH" panel shows the current state of variables, including "obj", "tinfo", "dest", "globals", and "header". The "CALL STACK" panel shows the current function call stack, with "eh_throw.cc" at the top. The "DEBUG CONSOLE" panel shows the output of the program, including the launch command and the process ID.

Visual Studio Code interface showing a C++ development environment. The main editor displays the source code of `eh_throw.cc`. A yellow box highlights the debug toolbar icons: `Step Out` (a blue arrow pointing right), `Continue` (a blue arrow pointing right), `Breakpoint` (a blue square), `Step Into` (a blue arrow pointing down), `Step Over` (a blue arrow pointing right), `Restart` (a green circular arrow), and `Stop` (a red square).

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** 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.

The bottom status bar shows the current file is `eh_throw.cc` at line 77, column 1, in the `Debug (swpp-compiler)` session. The status bar also indicates the current build system is `CMake` and the compiler is `Clang 14.0.0`.



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 with 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, in the `swpp-compiler` workspace.

Visual Studio Code interface showing a C++ file named `inst.cpp` in the `src/lib/backend/assembly` directory. The file is being edited, and the current line (21) is highlighted, showing a breakpoint set on the line:

```
21 joined_string.append(D token);
```

The text "Paused on breakpoint" is overlaid on the code editor.

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

- 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 function 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

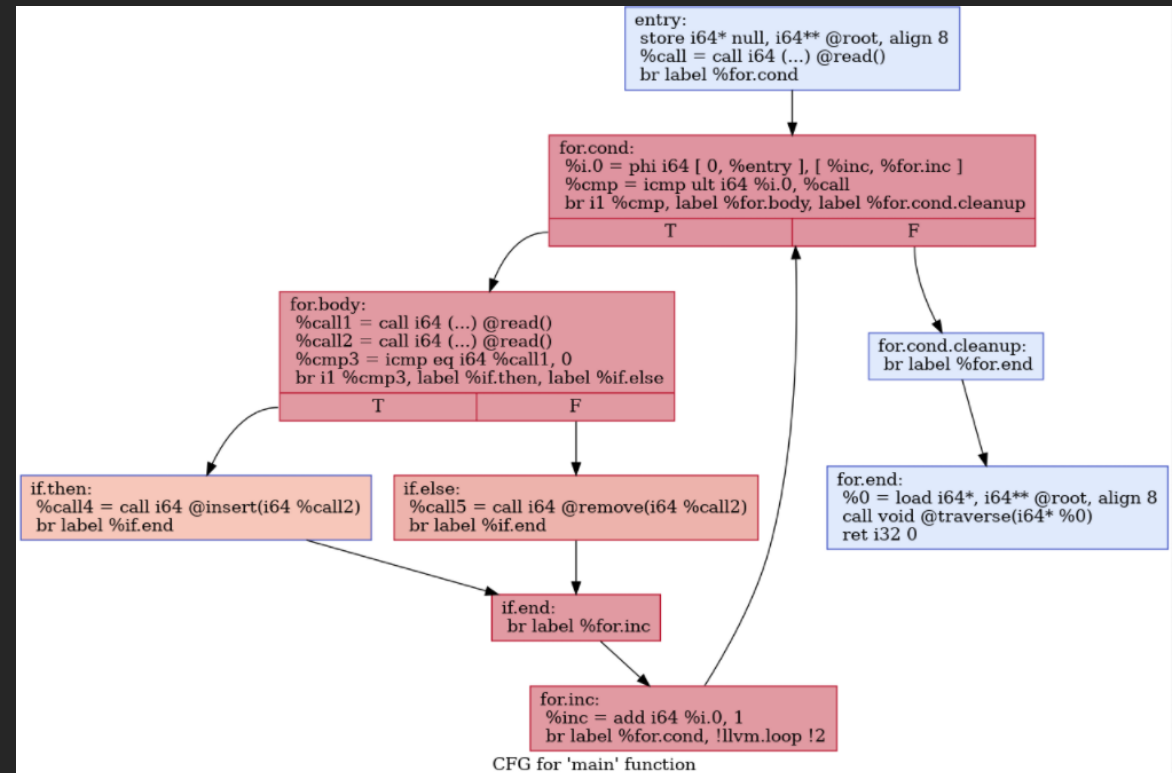
The bottom status bar shows the current file is `inst.cpp` in the `src/lib/backend/assembly` directory, with the following settings:

- main*
- 0 0 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 21, Col 26 Spaces: 2 UTF-8 LF C++ Linux

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