Debugging C++ Coroutines in GDB

ZARTAJ MAJEED

Cppcon
The C++ Conference
2025
September 13 - 19

How hard is it to debug C++ coroutines in GDB?

- Debugging is hard
- C++ coroutines are hard
- GDB is hard
- Debugging C++ coroutines in GDB is 3 times as hard

GCC Bug 99215 - Coroutines: debugging with gdb

Open and unassigned since 23 Feb 2021

"I am itching to get into C++20 coroutines (and very grateful for their implementation) but am somewhat put off by the apparent inability to inspect them from within a debugger currently.

While looking for existing related GCC specific issues, discussions or commits (none of which I found) the following paper [Debugging C++ coroutines] did come up: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2073r0.pdf

This seems to at least confirm the current state that I was seeing

I can not tell if support for this is missing in GCC or GDB or both but I figured I'd try finding out here first."

Is it still really that bad?

No.

Things have improved a lot since the survey in P2073

You can set breakpoints in coroutines, view local variables, view coroutine parameters and examine a coroutine's promise type

Some features of coroutines that are unusual for functions are still difficult to work with in GDB

These include examining suspended coroutines and figuring out async call stacks

A problem Knuth used to motivate coroutines

String Decoding Problem

- From The Art of Computer Programming by Donald Knuth
- A text parsing and transformation problem
- Use coroutines as loosely-coupled functions collaborating to solve a problem

String Decoding Problem
The Art of Computer Programming, Vol 1 & Fascicle 1
Donald Knuth

String Decoding Problem - Input

-- Fascicle 1, TAOCP, p.67

Suppose we want to write a program that translates one code into another. The input code to be translated is a sequence of 8-bit characters terminated by a period, such as

`a2b5e3426fg0zyw3210pq89r.` (1)

This code appears on the standard input file, interspersed with whitespace characters in an arbitrary fashion. For our purposes a "whitespace character" will be any byte whose value is less than or equal to **`#20`**, the ASCII code for ' '. All whitespace characters in the input are ignored

String Decoding Problem - Output

-- Fascicle 1, TAOCP, p.67

The other characters should be interpreted as follows, when they are read in sequence:

(1) If the next character is one of the decimal digits 0 or 1 or ... or 9, say n, it indicates (n + 1) repetitions of the following character, whether the following character is a digit or not.

(2) A nondigit simply denotes itself.

The output of our program is to consist of the resulting sequence separated into groups of three characters each, until a period appears; the last group may have fewer than three characters. For example,

`a2b5e3426fg0zyw3210pq89r.` (1)

should be translated into

`abb bee eee e44 446 66f gzy w22 220 0pq 999 999 999 r.` (2)

String Decoding Problem - Notes -- Fascicle 1, TAOCP, p.67

Notice that `3426f` does not mean 3427 repetitions of the letter `f`; it means 4 fours and 3 sixes followed by `f`.

If the input sequence is '`1.`', the output is simply '`.`', not '`..`', because the first period terminates the output.

The goal of our program is to produce a sequence of lines on the standard output file, with 16 three-character groups per line (except, of course, that the final line might be shorter).

The three-character groups should be separated by blank spaces, and each line should end as usual with the ASCII newline character `#a`.

String Decoding Problem
Original TAOCP Solution with Coroutines

What is a coroutine to Knuth? -- Fascicle 1, TAOCP, pp.66-67

Subroutines are special cases of more general program components, called coroutines. In contrast to the unsymmetric relationship between a main routine and a subroutine, there is complete symmetry between coroutines, which call on each other.

A subroutine is always initiated *at its beginning*, which is usually a fixed place; a coroutine is always initiated *at the place following* where it last terminated.

Such coroutine linkage is easy to achieve with MMIX if we set aside two global registers, `a` and `b`. In coroutine `A`, the instruction `GO a,b,0` is used to activate coroutine `B`; in coroutine `B`, the instruction `GO b,a,0` is used to activate coroutine `A`

String decoding - A solution using coroutines

Split problem into three functions along input, output and processing of single character items

- `NextChar`, to manage the raw input buffer
- `In`, to parse input and provide a character item for further processing
- `Out`, to format a character item and print output

`In` and `Out` are coroutines working in tandem

`In` is a generator that yields one character

`Out` fills a group of three characters awaiting each in sequence from `In`

`NextChar` is a subroutine called by `In`

String Decoding Problem
C++ Coroutines Port of TAOCP MMIX Solution

In() coroutine

```
InRO In() {

  char inchr;
  int count;

  for(;;) {
    inchr = NextChar();
    if(inchr > '9') {
      co_yield inchr;
      continue;
    }

    count = inchr - '0';
    if(count < 0) {
      co_yield inchr;
      continue;
    }

    inchr = NextChar();
    co_yield inchr;
    for(--count; count >= 0; --count) {
      co_yield inchr;
    }
  }
}
```

NextChar function

```
char NextChar() {
  static char InBuf[1000];
  static char* inptr = InBuf;
  char inchar;

  for(;;) {
    if(*inptr == '\0') {
      inptr = InBuf;
      if(fgets(InBuf, sizeof InBuf,
               stdin) == NULL)
        *inptr = '.';
    }
    inchar = *inptr;
    ++inptr;
    if(!isspace(inchar) &&
       !iscntrl(inchar))
      break;
  }
  return inchar;
}
```

Out() coroutine, 1/3

```
OutRO Out() {

  char OutBuf[] = {
    0, 0, 0, ' ',
    /* group repeated 14 times */,
    0, 0, 0, '\n',
    0
  };
  char* outptr = OutBuf;

  InRO&& in = In();
```

```
for(;;) {

  char outchr = co_await in;
  outptr[0] = outchr;

  if(outchr == '.') {
    finishLine(OutBuf,
      (uint64_t)(outptr + 3 - OutBuf));
    break;
  }
```

Out() coroutine, 2/3

```
outchr = co_await in;
outptr[1] = outchr;

if(outchr == '.') {
  ++outptr;
  finishLine(OutBuf,
    (uint64_t)(outptr + 3 - OutBuf));
  break;
}
```

```
outchr = co_await in;
outptr[2] = outchr;

if(outchr == '.') {
  outptr += 2;
  finishLine(OutBuf,
    (uint64_t)(outptr + 3 - OutBuf));
  break;
}
```

Out() coroutine, 3/3

```
    outptr += 4;

    if(outptr != OutBuf + sizeof(OutBuf) - 1) {
      continue;
    }

    fputs(OutBuf, stdout);
    outptr = OutBuf;
  }
}
```

```
void finishLine(char* line, uint64_t len) {
  line[len - 2] = '\n';
  line[len - 1] = '\0';
  fputs(line, stdout);
}
```

First among equals?

- Unlike subroutines, coroutines don't have hierarchical caller-callee relationship
- Which coroutine should run first and what should its initial state be?
- Better to start with `Out` for decoding since it needs to await a character from `In`
- Run `Out` right away to suspend at first `co_await`
- Start `In` suspended and resume when `Out` needs a value

Program start - Decoding problem

- Call **`Out`** coroutine to kick off processing
- Program does not exit till **`Out`** coroutine suspends or returns and control is transferred to **`main`**

```
int main() {
  Out();
}
```

String Decoding Problem
Compiler Machinery for C++ Coroutines Solution

In coroutine - Return object - Compiler machinery

- **`InRO`** is return type of **`In`** coroutine
- Forward declare nested **`promise_type`**
- Has **`coroutine_handle`** to **`In`** coroutine frame
- Constructor runs when compiler calls **`get_return_object`** method of **`promise_type`**
- Destructor destroys coroutine frame

```
struct InRO {

  struct promise_type;
  coroutine_handle<promise_type> coro;

  InRO(coroutine_handle<promise_type> h)
    : coro(h) {}

  ~InRO() {
    coro.destroy();
  }
// omitted ...
```

In coroutine - Promise type - Compiler machinery

- No need for **`coroutine_traits`** if promise type is inside return object
- Compiler calls **`get_return_object`** to construct return object instance using coroutine handle
- **`return_void`** required for coroutine that does not **`co_return`** a value
- **`unhandled_exception`** also required
    - ours simply terminates program

```
// struct InRO {
struct promise_type {
  InRO get_return_object() {
    return InRO{
      coroutine_handle<promise_type>::
        from_promise(*this)};
  }

  void return_void() {}
  void unhandled_exception() noexcept {
    terminate();
  }
// omitted ...
};
```

In coroutine - Initial/Final suspend - Compiler machinery

- Previously decided to suspend **`In`** coroutine at start
- **`In`** never actually breaks out of its main loop
- So **`final_suspend`** is never called and it can return any **`awaitable`**
- We arbitrarily return **`suspend_always`**

```
// struct promise_type {

auto initial_suspend() noexcept {
  return suspend_always{};
}

auto final_suspend() noexcept {
  return suspend_always{};
}

// };
```

In coroutine - co_yield handler - Compiler machinery

- `yield_value` is called with `co_yield` argument
- Allows `In` promise to update `val` member that can be retrieved later through promise

- Returns **`awaitable`** because `co_yield` is just a form of `co_await`
- We suspend `In` coroutine to return control to where `In` was resumed

```
// struct promise_type {

char val;

auto yield_value(char x) {
  val = x;
  return suspend_always{};
}
// };
```

In coroutine - Make InRO awaitable - Compiler machinery

- The return object of `In()` is made `awaitable` by adding `await_ready`, `await_suspend` and `await_resume` methods
- `In()` `await_ready` returns `true` because `In()` is always ready to return next character
- Simplest `void` form of `await_suspend` is used since it's never called

```
// struct InRO {

bool await_ready() const noexcept {
  return true;
}

void await_suspend(coroutine_handle<>) {}

// };
```

In coroutine - Orchestrate awaitable - Compiler machinery

- `InRO` `await_resume` resumes its own `In()` coroutine
- This causes `In()` to run till it yields
  - Recall `In()` promise `yield_value` sets `val` member of promise then suspends
- Then `await_resume` can return promise `val` member to awaiting coroutine

```
// struct InRO {

char await_resume() const noexcept {
  coro.resume();
  return coro.promise().val;
}

// };
```

Out coroutine - Return object - Compiler machinery

- `OutRO` is return type of `Out()` coroutine
- Forward declare nested `promise_type`
- Has `coroutine_handle` to `Out()` coroutine frame
- Constructor runs when compiler calls `get_return_object` method of `promise_type`
- Destructor destroys coroutine frame

```
struct OutRO {
  struct promise_type;
  coroutine_handle<promise_type> coro;

  OutRO(coroutine_handle<promise_type> h)
    : coro(h) {}

  ~OutRO() {
    coro.destroy();
  }
// omitted ...
```

Out coroutine - Promise type - Compiler machinery

- Easier to define promise type inside return object than using **`coroutine_traits`**
- Compiler calls **`get_return_object`** to construct return object instance with coroutine handle
- **`return_void`** required for coroutine that does not **`co_return`** a value
- **`unhandled_exception`** also required
  - ours simply terminates program

```
// struct OutRO {
struct promise_type {
  OutRO get_return_object() {
    return OutRO{
      coroutine_handle<promise_type>::
        from_promise(*this)};
  }

  void return_void() {}
  void unhandled_exception() noexcept {
    terminate();
  }
// omitted ...
};
```

Out coroutine - Initial/Final suspend - Compiler machinery

- Previously decided to immediately run `Out()` coroutine at start
- Want **`final_suspend`** to suspend `Out()` to let its return object destructor destroy coroutine frame

```
// struct promise_type {

auto initial_suspend() noexcept {
  return suspend_never{};
}

auto final_suspend() noexcept {
  return suspend_always{};
}

// };
```

Demo