Onboarding - Recap 04 - Debugging

Debuggear es costoso. Lleva tiempo entender el código y más aún la causa del error.

En todo momento hay que invertir en técnicas de programación, metodologías y herramientas que nos eviten cometer errores y asi reducir la necesidad de debugging.

Aun asi, eventualmente necesitaremos debugguear y para ello saber usar un debugger es fundamental.

Poner prints, compilar y ver que pasa es **totalmente ineficiente**.

El debugging es ya difícil, no lo compliques aún más. **Usa un debugger**.

En este *recap* veras algunas características de GDB. Todo lo que veas en este tutorial aplica a cualquier otro debugger.

Es importantísimo que uses un debugger con el que te sientas comodo/comoda.

El debugging es ya difícil, no lo compliques aún más!!

GDB es tosco y ciertamente feo. Buscate uno de tu agrado y cuando termines este recap, *hacelo de nuevo* pero con tu debugger de preferencia.

Lo que aprendas aquí lo podrás usar en cualquier otro lenguaje y te sera de increible valor.

¡Tu respuesta es exitosa! Tu calificación es 100.0%. [Tarea #66d2329d6096f1682c45ad8d] ×

Información

Autor (res)	Martin Di Paola
Fecha de entrega	03/09/2024 18:00:00
Estado	Succeeded
Calificación	100%
Promedio ponderado	1.0
Número de intentos	13
Tiempo límite de envío	Sin límite de envío

Enviado como

- Santiago Jorda - 102924
- Grupo: Default classroom

Para evaluación

- Tu mejor envío es
- 30/08/2024 17:59:06 - 100.0%

Bitácora de envíos

- 30/08/2024 17:59:06 - 100.0%
- 30/08/2024 17:43:25 - 95.45%

Pregunta 1:

Instalate byexample con `pip3 install byexample`

Clonate el repositorio `https://github.com/Taller-de-Programacion/hands-on-gdb.git` con `git clone https://github.com/Taller-de-Programacion/hands-on-gdb.git`

Verifica que podes compilarlo con `make` y que podes correrle los tests con `make tests`.

Revisa el Makefile.

Con que flags se esta compilando el programa? Hay más de uno, **marcarlos todos!**

☐ `--O3`
☑ `--O0`
☑ `--std=c++17`
☐ `--std=c++98`
☑ `--ggdb`

Pregunta 2:

El markdown `doctests/intset.md` funciona tanto como documentación como tests automáticos.

Segun la documentación (y sin espiar en el código!), cual de las siguientes afirmaciones son correctas?

☑ `IntSet` es un `set`.
☐ `IntSet::add` y `IntSet::remove` agregan/remueven un `int` del `set` respectivamente.
☐ `IntSet` es implementado con un árbol binario balanceado (Red-Black).
☑ `IntSet` es implementado con un árbol binario no-balanceado.
☐ `IntSet` es implementado con un árbol binario balanceado (AVL).
☐ `IntSet` es parte de la STL, la librería estándar de C++ (C++17).

Pregunta 3:

El markdown `doctests/intset.md` funciona tanto como documentación como tests automáticos.

Corre los tests con `make tests` o bien con `byexample -l shell --timeout 8 doctests/*.md`.

Cuántos tests pasaron (`pass`) y cuántos fallaron (`fail`)?

○ Pass: 0 Fail: 7
○ Pass: 5 Fail: 5
● Pass: 2 Fail: 5
○ Pass: 5 Fail: 2

Pregunta 4:

Entender que es lo que debería hacer una parte del programa que falla es tener la mitad de la batalla ganada.

En el ejemplo de la línea 40. Qué es lo que dice la documentación que deberia suceder (lo esperado)? Y que es lo que realmente sucede (lo obtenido)?

Hay más de un opción correcta, **marcarlas todas!**

☑ El test falla por que la documentación dice `is 34 in the set? false` (lo esperado) pero el programa imprime `is 34 in the set? true` (lo obtenido).
☑ La documentación dice que se agrega el número 21 y este **no** estaba en el set anteriormente: `21 added, was already in the set? false`
☑ El input del test es `"a 21 h 21 h 34"`.
☐ La documentación dice que se agrega el número 34 y este **no** estaba en el set anteriormente: `is 34 in the set? true`
☐ La documentación dice que se agrega el número 34 y este **si** estaba en el set anteriormente: `is 34 in the set? true`
☐ El test falla por que la documentación dice `is 34 in the set? true` (lo esperado) pero el programa imprime `is 34 in the set? false` (lo obtenido).
☐ El input del test es `"has 21 has 34"`.

Pregunta 5:

It's debugging time!

Nos enfocaremos en el test de la línea 40.

Ejecutá el `echo` y redireccioná su output a un archivo, digamos `input.txt`. (ya hiciste el recap `Onboarding - Recap 01 - Proceso de Building y Testing`, no?)

Corré `gdb bin/intset` o `gdb --tui bin/intset`. En ese momento GDB cargara el binario y el código fuente pero el programa aún no habra iniciado.

Para arrancarlo ingresá `start < input.txt`. Esto le dice a GDB no solo que arranque el programa sino que tome el archivo `input.txt` y lo redireccione.

La idea es que GDB (como cualquier debugger) te permitira ejecutar tu código línea a línea y explorar las variables.

Luego del `start`, en que archivo y línea el debugger se detuvo?

○ Archivo `bin/intset`, línea 1.
● Archivo `intset/main.cpp`, línea 5.

Pregunta 6:

Con `n` GDB se moverá a la siguiente línea de código (`next`) mientras con `s` GDB se meterá adentro de las funciones (`step`).

Ingresando `n` y `s`, llevá al debugger al método `IntSetController::process`.

A partir de ahí con `n` y `s` llevá al debugger al método `IntSet::has`.

Tendras que tipear `n` y `s` un par de veces hasta llegar (no me odies). TIP: si presionas enter GDB ejecuta el último comando.

Una vez dentro de `IntSet::has`, en que archivo y línea el debugger se detuvo?

○ Archivo `intset/intset.h`, línea 81.
○ Archivo `intset/intset.cpp`, línea 31.
○ Archivo `intset/intset.cpp`, línea 29.
○ Archivo `intset/intset.h`, línea 74.
● Archivo `intset/intset.cpp`, línea 30.

Pregunta 7:

GDB te permite ver el contenido de una variable con el comando `p`.

Movete hasta la línea de `return` y ahí escribí `p n` para ver el valor del puntero `n` y `p *n` para ver a que apunta. Escribí tambien `p val` para ver por que entero estas preguntando.

Fijate que usar `p` es **mucho más rápido** que estar metiendo prints y compilando tu programa!!

Imagina todo **el tiempo q te ahorras usando un debugger**!!

● El puntero `n` **no** es null y apunta a algo que tampoco es null. El valor de `val` es 21.
○ El valor de `val` es basura.
○ El puntero `n` **no** es null **pero** apunta a algo que **si** es null. El valor de `val` es 21.

Pregunta 8:

Ya te diste cuenta porque `IntSet::has` esta retornando el valor incorrecto?

Modificá el código con el fix, compilá y volvé a correr las pruebas. Deberiás ver que el test pasa ahora.

Nota: modificá sólo la línea que necesites, no agregues ni remuevas ninguna línea de más sino no podras seguir con este recap!

Una vez que tengas el fix corre `git diff` y verifica que solo estas modificando una línea sin agregar ni remover otras. Recien ahí podes commitear el fix.

Cual era el bug?

● Se compara por igualdad (`== nullptr`) cuando debería ser por desigualdad (`!= nullptr`).
○ El puntero `n` **no** es null pero debería serlo.
○ El parámetro `val` es basura.

Pregunta 9:

Ahora nos enfocarmos en el test de la línea 66.

Por que esta fallando?

○ El listado esta fuera de orden.
○ El número 21 no aparece en el listado.
● El número 33 no aparece en el listado.
○ Hay repetidos.

Pregunta 10:

Ejecuta el `echo` del test y redireccionalo a `input.txt`. Lanza GDB y arranca el debugging con `start < input.txt >/dev/null`.

Podrias hacer uso de los comandos `n` (`next`) y `s` (`step`) para navegar por el código y llegar a `IntSet::as_list` como lo hiciste en el ejercicio anterior pero sería agotador, no te parece?

Para esto tenemos los **breakpoints**.

Un breakpoint es una marca en alguna parte del programa de nuestro interes. Cuando el programa ejecuta dicha parte *"salta"* el breakpoint y GDB detiene la ejecución justo ahí.

Pone un breakpoint en `IntSet::as_list` escribiendo `b IntSet::as_list`.

Podes ver todos los breakpoints instalados corriendo `info breakpoints`.

Ahora, decile a GDB que continue con la ejecucion del programa con `c` (`continue`).

En que archivo y línea el debugger se detuvo?

● Archivo `intset/intset.cpp`, línea 59.
○ Archivo `intset/intset.cpp`, línea 57.
○ Archivo `intset/intset.cpp`, línea 60.

Pregunta 11:

Los breakpoints son formas de viajar rápido en el programa.

Estando dentro de `IntSet::as_list`, pone un breakpoint en la línea 73 (en el `while`) ejecutando `b 73`.

Notaras que GDB es inteligente y que entiende que `73` es un número de línea del archivo actual.

Dale continuar (`c`).

En este momento podríamos imprimir las variables locales, como para tener una idea que hay. Podes usar `p` como lo viste antes pero hay un shortcut.

Si escribis `info locals` te imprime todas las variables locales de una.

Y no te olvides q con `p` podes desreferenciar. Por ejemplo podrías ver `p *current` para ver el nodo actual o `p stack.top()->value` para ver el valor guardado en el nodo que está en el top del stack.

Que valores se obtuvieron? Hay más de un opción correcta, **marcarlas todas!**

Nota: el comando `p` de GDB es muy flexible pero también es tedioso de usar. Acá es donde **realmente vale la pena** que uses un debugger con una buena interfaz gráfica. GDB es solo para valientes y nerds de consola! Para este tutorial segui con GDB pero **no dudes** en usar otro para el resto de la materia.

☐ El `stack` tiene 2 elementos (con valores 15 y 21).
☐ El entero guardado en el `top` del `stack` es 15.
☑ El entero guardado en el `top` del `stack` es 21.
☑ El `stack` tiene un solo elemento.
☑ La lista `result` esta vacía.
☑ El nodo de la derecha de `current` es nulo.
☑ El nodo de la izquierda de `current` es nulo.
☐ El nodo `current` es nulo.
☑ El entero guardado en `current` es 15.
☐ El `stack` tiene 3 elementos (con valores 15, 21 y 33).
☐ El nodo `current` apunta a algo que es nulo.
☐ El `stack` esta vacío.

Pregunta 12:

He aqui ahora la parte dura del debugging.

`IntSet::as_list` tiene un bug pero no sera tan simple de encontrar como lo fue con `IntSet::has`.

Como dato te cuento que el árbol tiene esta forma:

```
       21
      /  \
(left)    (right)
   |         |
  15        33
```

La mejor estrategia es tomar papel y lapiz e ir dibujando el árbol binario e ir viendo como este es recorrido usando el debugger.

Ir imprimiendo el valor de las variables con `p` o con `info locals` puede ser engorroso.

GDB te permite guardar expresiones que son automaticamente mostradas cada vez que el debugger se detenga.

Por ejemplo, con el comando `display *current` le estaras diciendo a GDB que imprima el valor de `*current` en cada paso.

Podes agregar tantas expresiones como quieras. Si queres borrar alguna usá `undisplay` y para ver que expresiones están activas usá `info display`.

Una vez que encuentres el bug implementá el fix (no agregues ni saques líneas, con modificar una sola línea deberías fixear el bug).

Con `git diff` verificá q solo estas modificando una línea y con `git add` y `git commit` commitea el fix.

TIP: El test mostraba que debería imprimirse `15, 21, 33` pero que se imprimía solo `15, 21`. Hay algo que falta recorrer!

● La condición del `while` debería preguntar no solo por `stack.empty()` sino también por `current != nullptr`
○ En la línea 90 el código hace `current = current->right;` cuando debería hacer `current = current->left;`
○ El orden de recorrido del árbol debería ser pre-orden pero el código hace incorrectamente en post-orden.
○ En la línea 80 el código hace `current = current->left;` cuando debería hacer `current = current->right;`
○ La condición del `while` debería preguntar no solo por `stack.empty()` sino también por `result.empty()`.
○ En la línea 73 el código hace `node_t *current = root->left;` cuando debería hacer `node_t *current = root->right;`
○ El recorrido del árbol usa incorrectamente un `stack` cuando debería usar una `queue`.

Pregunta 13:

Ahora nos enfocaremos en el test de la línea 79.

Por que esta fallando?

○ El método `IntSet::as_list` falla con un `abort`.
○ El método `IntSet::clear` lanza una excepción de C++.
● El método `IntSet::clear` falla con un `abort`.
○ El método `IntSet::as_list` crashea.

Pregunta 14:

Ejecutá el `echo` del test de la línea 79 y redireccionalo a `input.txt`. Corre el debugger y ejecutá `start < input.txt >/dev/null`.

Esta vez no pondremos ningun breakpoint. Como el programa crashea (recibe una señal del sistema operativo llamada `SIGABRT` para ser específicos), GDB es capaz de detectarla y frenar el programa justo momentos luego.

Dale a GDB `c` (`continue`) y deja que el programa crashee. Veras que GDB se detiene automáticamente aunque lo hará en una función interna de la libc (o std lib de C++).

Con el comando `bt` o `backtrace` podes ver el call stack, o sea la cadena de funciones y métodos que se fueron llamando hasta llegar a donde estas parado en ese momento.

Que funciones/métodos ves?

○ `main -> IntSet::as_list -> ...`
○ `main -> IntSet::~IntSet -> IntSet::clear -> ...`
● `main -> IntSetController::~IntSetController -> IntSet::~IntSet -> IntSet::clear -> ...`

Pregunta 15:

Con `bt` podes ver el call stack. Cada llamada a una función/método es lo que se llama un frame.

Con GDB podes moverte de un frame a otro con el comando `f n` (donde `n` es el número de frame al cual queres ir).

Moverte de un frame a otro no cambia en nada el estado de tu programa, este sigue frenado en el mismo lugar. Moverte a un frame te permite explorar las variables locales de ese frame.

Movete al frame del método `IntSet::clear` y mira el valor de la variable local `current`.

Que valor tiene?

● `current` es un puntero nulo
○ `current` es un puntero a algo no-nulo pero su `value` es nulo

Pregunta 16:

Cual es la causa del bug entonces?

Fixea el bug y comitealo. En este caso podes agregar las líneas que necesites.

TIP: imprimite no solo las variables locales de `IntSet::clear` sino también `p *this` para ver al objeto.

● `root` es un puntero nulo
○ `to_remove` esta vacio
○ `current` es un puntero nulo

Pregunta 17:

Cual es el problema real detras de la falla del test de la línea 88?

Fixealo y commitea. (Si luego del fix corres el test de nuevo veras que falla por otra cosa, eso lo arreglaras despues).

TIP: el programa imprime un error no? Una forma rápida de buscar la causa del bug es poner un breakpoint en la línea que imprime el mensaje de error y explorar las variables locales.

● El comando `c` no existe, 100% seguro que la documentación es correcta y que el código está mal (el bug está en el código)
○ El error es un falso positivo (realmente no hay un bug)
○ El comando `c` no existe, 100% seguro que el código es correcto y que la documentación está mal (el bug está en la documentación)

Pregunta 18:

Okay, ya arrelgaste uno de los problemas del test de la línea 88 pero hay algo más.

Fixealo y commitea.

La **moraleja** de este ejercicio es que un bug visible puede tener múltiples errores. Y que los errores no necesariamente estan solamente en el código: los tests también pueden tener errores!

○ El comando `l` no existe, 100% seguro que la documentación es correcta y que el código está mal (el bug está en el código)
○ El error es un falso positivo (realmente no hay un bug)
● El comando `l` no existe, 100% seguro que el código es correcto y que la documentación está mal (el bug está en la documentación)

Pregunta 19:

Vamos por el último!

El test de la línea 116 falla con los números listados en el orden incorrecto.

Pone un breakpoint en `IntSet::as_list` y debuggea el método.

Cual es el bug en `IntSet::as_list`?

○ El recorrido del árbol usa incorrectamente un `stack` cuando debería usar una `queue`.
○ El orden de recorrido del árbol debería ser pre-orden pero el código hace incorrectamente en post-orden.
○ El orden de recorrido del árbol debería ser pre-orden pero el código hace incorrectamente en in-orden.
● El bug no está en `IntSet::as_list`.

Pregunta 20:

En debugging siempre hay 3 pasos: saber que hay un problema, saber donde está y saber exactamente que es.

Con los tests automáticos **detectamos los bugs** (aunque algunos eran falsos positivos). Con ellos supiste si había un problema o no.

Hasta el momento los bugs siempre estuvieron "ahí nomás", siempre supiste donde empezar a debuggear aunque obviamente no sabías exactamente cual era el bug.

Para el test 116 la cosa cambió. El método `IntSet::as_list` está imprimiendo mal las cosas pero no parece estar ahí el problema.

Empezar a debuggear paso a paso todo el programa no es una estrategia eficiente. Llevaría mucho tiempo!

Aca tenemos que pensar y **descartar** grandes bloques de código y **reducir el área de búsqueda** lo más posible para asi debuggear solo una pequeña parte.

Que otros métodos podrían estar generando el bug?

● `IntSet::add`
○ `IntSet::count`
○ `IntSet::clear`
○ `IntSet::as_list`
○ `IntSet::has`

Pregunta 21:

Okay, supongamos que el problema está en `IntSet::add` cuando se agrega un número negativo.

Si pones un breakpoint ahí el programa se frenará una y otra vez por que justamente se agregan muchos números.

Podes apretar `c` (`continue`) varias veces e ir imprimiendo con `p val` el valor (o `display val`) y solo debugguear cuando veas un número negativo.

Pero es tedioso.

En vez de poner un breakpoint normal podes poner un **breakpoint condicional** que solo frene el programa si se cumple una condición.

En GDB podes hacer `b IntSet::add if val < 0` donde lo que viene luego del `if` es la condición.

Que trucazo!

Armado con todo lo que viste, cual es el error? Arreglalo y commitea.

● El casteo `unsigned` no debería estar.
○ La condición `value < val` deberia estar al revez.
○ El casteo `const_cast` no debería estar.

Pregunta 22:

En este *hands-on* viste las estrategias y herramientas básicas del debugging.

Siempre que tengas un bug corre herramientas automáticas como `cppcheck` o `valgrind` y solo cuando sigas con el problema anda al debugger.

Reducí tanto como te sea posible el área de búsqueda **descartando rápido**.

Debuggear lleva mucho tiempo, apuntá bien los cañones!

Si pensas que el bug esta en una función `x` pero no tenes 100% evidencia de ello, **no te encierres**. Usa la **lógica** y si la **evidencia** apunta a otro lado andá ahí.

Creeme, he encontrado bugs del mismísimo kernel de linux una vez que descarte (con evidencia) que mi código no era el del problema! (write up)

Una vez que ya sabes por donde está el problema **usá un debugger**.

Usar prints es tentador pero ineficiente y lento.

Debuggear es costoso, no lo hagas más costoso!

Pone breakpoints estratégicos (normales o condicionales) y usá `p`, `display` o `info locals` para verificar los valores y refutar o no tu hipótesis.

No especules.

En este **hands-on** viste GDB pero todo lo visto aplica a cualquier debugger (no solo para C/C++ sino para cualquier otro lenguaje).

Y GDB no es el debugger más bonito. Hay muchos más ahí afuera que te pueden servir para C/C++. Buscate uno y usalo! No te cases con GDB si te molesta. (y te recomiendo entonces que rehagas este recap con ese debugger para que sepas como usarlo, no querras perder tiempo mientras haces el TP)

Debuggear es difícil, no lo hagas más difícil!

● Si, usare un debugger de mi preferencia.
○ No, vengo usando prints desde Algo 1 y morire haciendo prints, total, tengo tiempo.