CSIS1123/COMP2123 Programming Technologies and Tools

Compensated material: Debugging with GDB

Objectives

At the end of this self-learning lab, you should be able to:

- Understand the use of basic debugging skills.
- Understand the use of gdb to debug a program.

Section 1. Install GDB

GDB, the GNU Project debugger, allows you to see what is going on `inside' another program while it executes -- or what another program was doing at the moment it crashed. GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened, when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another. Those programs might be executing on the same machine as GDB (native), on another machine (remote), or on a simulator. GDB can run on most popular UNIX and Microsoft Windows variants, as well as on Mac OS X. In this lab, we focus on native debugging in Linux. If you are interested, please read the online materials for remote debugging.

To install GDB in Fedora in your own virtual machine, run the following command:

```
$ sudo apt-get install gdb
```

GDB is already installed in X2Go, and you will not need to install it.

Section 2. First glance at GDB

In this section, we will use gdb to find out why a program throw exceptions (e.g. segmentation fault).

Run the sample program without GDB. This program takes a sequence of numbers as studentID and then print it.

```
$ cd lab-gdb
$ g++ -o sample sample.cpp
```

At this time, if we enter a number, the program throws a Segmentation fault.

```
$ ./sample
1
```

```
Segmentation fault (core dumped)
```

A segmentation fault is when your program attempts to access memory it has either not been assigned by the operating system, or is otherwise not allowed to access. Therefore, we need to find out the reason.

Running with GDB

In order to debug the program, we will need to run the program with GDB. First, you need to compile the program using '-g' flag in gcc:

```
$ cd lab-gdb
$ g++ -o sample sample.cpp -g
```

Run gdb with the sample program:

```
$ gdb ./sample
```

You will see the following information while it is running:

```
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./sample...done.
(gdb)
```

Start running the program and input 1:

```
(gdb) run
Starting program: /home/hkucs/comp2123-materials/gdb-lab/sample
1
```

The program stops at error:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400a05 in built_nodes () at sample.cpp:21
```

```
21              current_read->next = tmp;
(gdb)
```

This means that GDB stop at line 21 at sample.cpp file because it has a segmentation fault. Therefore, this line attempts to access a invalid memory. We can fix the program by manually inspect the program or use more advance GDB skills after.

This is the basic usage of GDB, it will tells you where the unrecoverable error is throw.

Get the current state in GDB

It is essential to get the state when a program stops. This state includes variable values, function call stack (i.e., to answer how the program gets here) and current register values.

Current Call Stack

You can use `up` and `down` to find out the current call stacks. `up` returns to the caller frame while `down` returns to the callee frame. For example:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400a05 in built_nodes () at sample.cpp:21
21              current_read->next = tmp;
(gdb) up
#1  0x0000000000400a43 in main (argc=1, argv=0x7fffffffe568) at sample.cpp:30
30          head = built_nodes();
(gdb) down
#0  0x0000000000400a05 in built_nodes () at sample.cpp:21
21              current_read->next = tmp;
```

At this example, the functions stops at the function `built_nodes()`, line 21 at sample.cpp. When we use up, we can know that the function `built_nodes()` is called at line 30 at sample.cpp. The up and down are useful to find out how the program stops at a specific line.

Variable Value

You can use `print` (or just a single character `p`) to get the value of a variable. For example:

```
(gdb) p current_read
$1 = (StudentNode *) 0x400adb <_GLOBAL__sub_I__Z11built_nodesv()+19>
```

Here, we get the value of current_read as 0x400adb, and it actually points to a function instead of memory storing a StudentNode. As memory storing a function is not writable, the assignment of `current_read->next = tmp` is illegal. After we inspect the code, we find out that the current_read is not initialized. We can fix this by initialized it when the `head` is `NULL`.

The p command just allows you to print out value of variables in the current execution frame. Therefore, you need to use up/down to switch frames to print local variables in caller or callee. For this example, if you want to print variables in the main function (e.g., current), no variables will be found. You need to use up to returns to the caller frame and print it.

```
#0  0x0000000000400a0d in built_nodes () at sample.cpp:21
21              current_read->next = tmp;
(gdb) p current
No symbol "current" in current context.
(gdb) up
#1  0x0000000000400a4b in main (argc=1, argv=0x7fffffffe568) at sample.cpp:30
30          head = built_nodes();
(gdb) p current
$2 = (StudentNode *) 0x7fffffffe560
```

Show the source file

GDB allows users to see the source file in debugging, which is useful for one-step debugging. Press ctrl+x+1 pops up a windows showing the source file and the current running line.

```
   ┌──sample.cpp──────────────────────────────────────────┐
   │13          int studentIdTmp;                         │
   │14          while (std::cin >> studentIdTmp) {        │
   │15              tmp = new StudentNode();              │
   │16              tmp->studentId = studentIdTmp;        │
   │17              tmp->next = NULL;                     │
   │18              if (head == NULL) {                   │
   │19                  current_read = head = tmp;        │
   │20              }                                     │
 > │21              current_read->next = tmp;             │
   │22              current_read = tmp;                   │
   │23          }                                         │
   │24          return head;                              │
   │25      }                                             │
   │26                                                    │
   │27      int main(int argc, char const *argv[]) {      │
   │28          StudentNode *head, *current;              │
   │29                                                    │
   │30          head = built_nodes();                     │
   │31          current = head;                           │
   └──────────────────────────────────────────────────────┘
native process 61990 In: built_nodes      L21   PC: 0x400a0d
(gdb)
```

This is the new popped-up window, and the current running line, and > specifies the line that the program currently stops. Users can use up/down to switch frames.

More information

GDB can also shows the assembly or register value. Students who are interested can see the official documentations for more information.

Use breakpoint to help debugging

Breakpoint is a debugging primitive that breaks a program running when specific conditions are satisfied.

Setting breakpoint at functions / lines

In GDB, users can use break to set breakpoint before or running the program. Run the sample program and break it at the start of built_nodes function

```
$ gdb ./sample
(gdb) break built_nodes
Breakpoint 1 at 0x40097e: file sample.cpp, line 11.
(gdb) run
Breakpoint 1, built_nodes () at sample.cpp:11
11      StudentNode* built_nodes() {
```

The program now stops at the beginning of built_nodes function.

To break at a specific line, use break filename:lineno

```
$ gdb ./sample
(gdb) break sample.cpp:14
Breakpoint 1 at 0x40098d: file sample.cpp, line 13.
(gdb) run
Starting program: /home/hkucs/comp2123-materials/gdb-lab/sample
Breakpoint 1, built_nodes () at sample.cpp:14
14          while (std::cin >> studentIdTmp) {
```

Conditional breaking

Single-step debugging

When you stop at a breakpoint, you can single-stepping the program to find out how the program execute. There are two simple instructions step and next to execute one source code line and then pause.

- both step and next execute one source code line and then pause the program
- next executes the whole function call and pause after the call (this process is usually called as step out)
- step stops at the first source line in the called function.

Debugging a simple program using single stepping

The following example sort a 10-element array {1, 3, 2, 10, 4, 50, 44, 86, 25, 11} using BST and prints the result. When we run the program, it shows only one element 1.

```
$ cd gdb-lab
$ g++ -o bst bst.cpp
$ ./bst
$ 1
```

We can run the program using gdb and single step from line 33 of bst.cpp.

```
$ gdb ./bst
(gdb) break bst.cpp:33
(gdb) run
Starting program: /home/hkucs/comp2123-materials/gdb-lab/bst
Breakpoint 1, main () at bst.cpp:33
33          for (int i = 0;i < 10;i++)
(gdb) s
34              root = insert_node(root, arr[i]);
(gdb) s
insert_node (current=0x0, val=1) at bst.cpp:14
14          if (current == NULL)
(gdb) s
15              return new Node(val);
(gdb) n
21      }
(gdb) s
main () at bst.cpp:33
33          for (int i = 0;i < 10;i++)
(gdb) s
34              root = insert_node(root, arr[i]);
(gdb) ??? (how can we see the value of root?)
$1 = ???
(gdb) ??? (how can we see the value of root->val?)
$2 = 1
(gdb) s
insert_node (current=0x613c20, val=3) at bst.cpp:14
14          if (current == NULL)
(gdb) s
16          if (current->val < val)
(gdb) s
17              insert_node(current->right, val);
(gdb) n
20          return current;
(gdb) s
21      }
(gdb) n
main () at bst.cpp:33
33          for (int i = 0;i < 10;i++)
(gdb) p root->right
```

```
$3 = ???
(gdb) p root->right->val
??? (what will happen?)
```

Can you fill in the ??? value using GDB? Why the program cannot print all the sorted elements?

From line 34, we know that the first element is inserted into the tree. From line 33, we know that the second element is not inserted into the tree. From the its value, we can know that the root->right value is not set. When we inspect the code, we find out that there are no code that assign the current->right, and we find out that we should change the insert_bst as follows:

```
// the correct insert_bst function in bst.cpp
Node* insert_node(Node* current, int val) {
    if (current == NULL)
        return new Node(val);
    if (current->val < val)
        current->right = insert_node(current->right, val);
    else
        current->left = insert_node(current->left, val);
    return current;
}
```