Debugging Techniques for C Programs
Debugging Basics

• Will focus on the gcc/gdb combination.
• Will also talk about the ddd gui for gdb (lots of value added to gdb).
• First, debugging in the abstract:
  – Program state is a snapshot of all variables, PC, etc.
  – A statement in your program transforms one program state into another.
  – You should be able (at some level) to express what you expect the state of your program to be after every statement.
  – Often state predicates on program state; i.e., “if control is here, I expect the following to be true.”
• Map into a toy example.
Small Example: `ave.c`

```c
#include <stdio.h>

int sum=0, val, num=0;
double ave;

main()
{
    while (scanf("%d\n", &val) != EOF) {
        sum += val;
        num++;
    }

    if (num > 0) {
        ave = sum/num;
        printf("Average is %f\n", ave);
    }
}
```

- `sum` should be 0 and `num` should be 0.
- `sum` should be the total of the `num` input values processed.
- `sum` should be the total of the `num` input values and there is no more input.
- `ave` should be the floating point mean of the `num` input data values.
Small Example: `ave.c`

```
% a.out
1
Average is 1.000000
% a.out
1
2
3
Average is 2.000000
% a.out
1
2
3
4
Average is 2.000000
```

Experienced programmer can probably “eyeball debug” the program from this output
Using **gdb**

- Make sure to compile source with the `-g` switch asserted.
- In our case, `gcc -g ave.c`
- **Breakpoint**: line in source code at which debugger will pause execution. At breakpoint, can examine values of relevant components of program state. `break` command sets a breakpoint; `clear` removes the breakpoint.
- Diagnostic `printf()` crude, but effective way of getting a snapshot of program state at a given point.
- Once paused at a breakpoint, use `gdb print` or `display` to show variable or expression values. `display` will automatically print values when execution halts at breakpoint.
- From a breakpoint, may `step` or `next` to single step the program. `step` stops after next source line is executed. `next` similar, but executes functions without stopping.
Using `gdb`

- May find out where execution is, in terms of function call chain, with the `where` command; also shows function argument values.
- Apply some of this in context of bogus averaging program.
- To make things easier, put the problematic data set in a file named data.

```bash
% a.out < data
Average is 2.000000
```
Using gdb (ave.c)

% gdb a.out
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i586-suse-linux". Using host libthread_db library "/lib/tls/libthread_db.so.1".

(gdb)
Using `gdb` (`ave.c`)
Using **gdb** (ave.c)

```
(gdb) break 8
Breakpoint 1 at 0x80483dc: file ave.c, line 8.
(gdb) break 13
Breakpoint 2 at 0x8048414: file ave.c, line 13.
(gdb) display num
(gdb) display val
(gdb) display sum
(gdb) r < data
Starting program: /home/jmayo/courses.d ...
Breakpoint 1, main () at ave.c:8
8     while (scanf("%d\n",&val) != EOF) {
3: sum = 0
2: val = 0
1: num = 0
(gdb) c
Continuing.

Breakpoint 1, main () at ave.c:8
8     while (scanf("%d\n",&val) != EOF) {
3: sum = 1
2: val = 1
1: num = 1
```
Using `gdb` (ave.c)

```
(gdb) c
Continuing.

Breakpoint 1, main () at ave.c:8
8   while (scanf("%d\n", &val) != EOF) {
3: sum = 3
2: val = 2
1: num = 2
(gdb) c
Continuing.

Breakpoint 1, main () at ave.c:8
8   while (scanf("%d\n", &val) != EOF) {
3: sum = 6
2: val = 3
1: num = 3
```
Using *gdb* (*ave.c*)

```
(gdb) c
Continuing.

Breakpoint 1, main () at ave.c:8
8 while (scanf("%d\n", &val) != EOF) {
  3: sum = 10
  2: val = 4
  1: num = 4

(gdb) c
Continuing.

Breakpoint 2, main () at ave.c:13
13   ave = sum/num;
  3: sum = 10
  2: val = 4
  1: num = 4
```
Using `gdb (ave.c)`

```
(gdb) n
14    printf("Average is %f\n", ave);
3: sum = 10
2: val = 4
1: num = 4
(gdb) p ave
$1 = 2
(gdb) p (double)sum/(double)num
$2 = 2.5
(gdb) c
Continuing.
Average is 2.000000
```

Everything fine until `ave` is computed. Integer division the problem.

Evaluate expression inside `gdb` to validate our reasoning.

Program exited with code 024.
```
(gdb) q
%
A GUI for gdb: ddd

• The ddd program is just a GUI front-end for gdb.

• Value added three main ways:
  – Can mouse left on source line, then mouse left on `Break at()` to set a breakpoint. Or mouse right on a source line and set a breakpoint in the menu that pops up.
  – Can mouse left on variable, then mouse left on `Print()` or `Display()` to examine data values. Or get value displayed at bottom of ddd window by "mouse hovering" over a variable name.
  – Displayed values graphically displayed. Click on a pointer value, graphically display thing pointed to. Visualize complex linked data structures.

• Play with inorder tree traversal program.
Using \texttt{\texttt{ddd} (inorder.c)}

Introduce a pointer-related bug into the program by modifying the \texttt{inorder()} function:

\begin{verbatim}
void inorder(r)
struct node *r;
{
    inorder(r->left);
    printf("%c",r->data);
    inorder(r->right);
}
\end{verbatim}

Formerly:

\begin{verbatim}
if (r != NILNODE) {
    inorder(r->left);
    printf("%c",r->data);
    inorder(r->right);
}
\end{verbatim}
Quickie Post Mortem Debugging (inorder.c)

% a.out
Segmentation fault (core dumped)

% gdb a.out core
GNU gdb 6.1
..........
Core was generated by `./a.out core'.
Program terminated with signal 11, Segmentation fault.
..........
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0 0x080484d5 in inorder (r=0x0) at buggy_inorder.c:38
    inorder(r->left);
(gdb)

Function in which segfaulted.
Arguments to function.
Line of source where segfaulted.
Quickie Post Mortem Debugging (inorder.c)

(gdb) where
#0 0x080484d5 in inorder (r=0x0) at buggy_inorder.c:38
#1 0x080484dd in inorder (r=0x804a008) at buggy_inorder.c:38
#2 0x080484dd in inorder (r=0x804a028) at buggy_inorder.c:38
#3 0x080484dd in inorder (r=0x804a048) at buggy_inorder.c:38
#4 0x080484dd in inorder (r=0x804a068) at buggy_inorder.c:38
#5 0x08048479 in main () at buggy_inorder.c:21

The above listing walks back the call chain as it was at the moment of the segfault.

Clear that we dereferenced a null pointer in a call to inorder() at a leaf node of the binary tree.
```
#include <stdlib.h>
#define NILNODE (struct node *)0

struct node {
    char data;
    struct node *left, *right;
};

main()
{
    struct node *gimme(), *n1, *n2, *n3, *n4, *n5, *n6, *n7;
    void inorder();
    n1 = gimme('a', NILNODE, NILNODE);
    n2 = gimme('b', NILNODE, NILNODE);
    n3 = gimme('c', n1, n2);
    n4 = gimme('d', NILNODE, NILNODE);
    n5 = gimme('e', n3, n4);
    n6 = gimme('f', NILNODE, NILNODE);
    n7 = gimme('g', n5, n6);
    inorder(n7);
    printf("n");
}

struct node *gimme(val, l, r)
char val;
struct node *l, *r;
{
    struct node *tmp;
    tmp = (struct node *) malloc(sizeof(struct node));
    return tmp;
}
```
#include <stdlib.h>

#define NILNODE (struct node *)0

struct node {
    char data;
    struct node *left, *right;
};

main()
{
    struct node *gimme(), *n1, *n2, *n3, *n4, *n5, *n6, *n7;
    void inorder();

    n1 = gimme('a', NILNODE, NILNODE);
    n2 = gimme('b', NILNODE, NILNODE);
    n3 = gimme('c', n1, n2);
    n4 = gimme('d', NILNODE, NILNODE);
    n5 = gimme('e', n3, n4);
    n6 = gimme('f', NILNODE, NILNODE);
    n7 = gimme('g', n5, n6);

    inorder(n7);
    printf("n");

    struct node *gimme(val, l, r)
    char val;
    struct node *l, *r;

    struct node *tmp;
    tmp = (struct node *) malloc(sizeof(struct node));
}

Gdb) run
Starting program: /home/f85/kearns/courses/d/cs315-S03.d/slides.d/debug.d/a.out
Breakpoint 1, main () at buggy_inorder.c:21
(gdb)
1. tmp = (struct node *) malloc(sizeof(struct node));
   tmp->data = v; tmp->left = l; tmp->right = r;
   return(tmp);
}

void inorder(r)
    struct node *r;
{
    inorder(r->left);
    printf("\%c", r->data);
    inorder(r->right);
}

(gdb) graph display *(n->left->left) dependent on 3
(gdb) graph display *(n->left->right) dependent on 3
(gdb) graph display *(n->left->left->left) dependent on 5
(gdb) graph display *(n->left->left->right) dependent on 5
(gdb)
Debugging Tips

• Examine the most recent change
• Debug it now, not later
• Read before typing
• Make the bug reproducible
• Display output to localize your search
• Write a log file
• Use tools
• Keep records