Lab #6
Using debugging software

In This Week's Lab…

- Passing pointers to functions
- Using **valgrind** to find memory leaks
- How to start **gdb** and control how a program runs

Pointers & Functions

- Since C doesn’t allow you to return multiple datatypes, you can pass pointers to functions to modify multiple variables

- But be careful, C doesn’t have *true* pass-by-reference!

```
void
func1(int n, int a)
{
    a = n;
    printf("func1(a)=%02d\n", a);
    fflush(stdout);
}


int
main(int argc, char **argv)
{
    int x = 0;
    int *z = NULL;

    func1(1,x);
    printf("main(a)=%02d\n", x);
    fflush(stdout);

    return 0;
}
```

What would be the output in main be from this code?

a) main(a) = 01
b) main(a) = 1
c) main(a) = 00
d) main(a) = 0

```
void
func2(int n, int* a)
{
    *a = n;
    printf("func2(a)=%02d\n", *a);
    fflush(stdout);
}


int
main(int argc, char **argv)
{
    int x = 0;
    int *z = NULL;

    z = &x;
    func2(2,z);
    printf("main(a)=%02d\n", x);
    fflush(stdout);

    return 0;
}
```

What would be the output from this code?

a) main(a) = 00
b) main(a) = 02
c) main(a) = unknown
d) Segmentation fault.

```
int*
func3(int n)
{
    int *a = (int*)malloc(sizeof(int));
    if(a == NULL) {
        fprintf(stderr, "oopsie");
        exit(1);
    }
    memset(a,n,1);
    printf("func3(a)=%02d\n", *a);
    fflush(stdout);

        return a;
}

int
main(int argc, char **argv)
{
    int x = 0;
    int *z = NULL;

    z = func3(3);
    printf("main(a)=%02d\n", x);
    fflush(stdout);
    return 0;
}
```

What would be the output from this code?

a) main(a) = 00
b) main(a) = 03
c) main(a) = unknown
d) Segmentation fault.

```
int*
func3(int n)
{
    int *a = (int*)malloc(sizeof(int));
    if(a == NULL) {
        fprintf(stderr, "oopsie");
        exit(1);
    }
    memset(a,n,1);
    printf("func3(a)=%02d\n", *a);
    fflush(stdout);

        return a;
}

int
main(int argc, char **argv)
{
    int x = 0;
    int *z = NULL;

    z = func3(3);
     x = *z;
    printf("main(a)=%02d\n", x);
    fflush(stdout);
     return 0;
}
```

What would be the output from this code?

a) main(a) = 00
b) main(a) = 03
c) main(a) = unknown
d) Segmentation fault.

```
void
func4(int n, int *a)
{
    a = (int*)malloc(sizeof(int));
    if(a == NULL) {
        fprintf(stderr, "oopsie");
        exit(1);
    }
    memset(a,n,1);
    printf("func4(a)=%02d\n", *a);
    fflush(stdout);
}

int
main(int argc, char **argv)
{
    int x = 0;
    int *z = NULL;

    func4(4,z);
    x = *z;
    printf("main(a)=%02d\n", x);
    fflush(stdout);
    return 0;
}
```

What would be the output from this code?

a) main(a) = 00
b) main(a) = 04
c) main(a) = unknown
d) Segmentation fault.

```
void
func5(int n, int *a)
{
    memset(a,n,1); //or, *a = n;
    printf("func5(a)=%02d\n", *a);
    fflush(stdout);
}


int
main(int argc, char **argv)
{
    int x = 0;
    int *z = NULL;

    z = (int*)malloc(sizeof(int));
    func5(5,z);
    x = *z;
    printf("main(a)=%02d\n", x);
    fflush(stdout);

    return 0;
}
```

What would be the output from this code?

a) main(a) = 00
b) main(a) = 05
c) main(a) = unkown
d) Segmentation fault.

```c
void
func6(int n, int_t* a)
{
    a = (int_t*)malloc(sizeof(int_t));
    if(a == NULL) {
        fprintf(stderr, "oopsie");
        exit(1);
    }
    a->data = n;

    printf("func6(a)=%02d\n", a->data);
    fflush(stdout);
}

int
main(int argc, char **argv)
{
    int x = 0;
    int_t *z = NULL;

    func6(6,z);
    x = z->data;
    printf("main(a)=%02d\n\n", x);
    fflush(stdout);
    return 0;
}
```

What would be the output from this code?

a) main(a) = 00
b) main(a) = 06
c) main(a) = unknown
d) Segmentation fault.

```
void
func6(int n, int_t* a)
{
    a = (int_t*)malloc(sizeof(int_t));
    if(a == NULL) {
        fprintf(stderr, "oopsie");
        exit(1);
    }
    a->data = n;

    printf("func6(a)=%02d\n", a->data);
    fflush(stdout);
}

int
main(int argc, char **argv)
{
    int x = 0;
    int_t *z = NULL;

       z = 
  (int_t*)malloc(sizeof(int_t));
    func6(6,z);
    printf("main(a)=%02d\n", z);
    fflush(stdout);
    return 0;
}
```

What would be the output from this code?

a) main(a) = 00
b) main(a) = 06
c) main(a) = unknown
d) Segmentation fault.

```
void
func7(int n, int_t** a)
{
  (*a)->data = n;

  printf("func7(a)=%02d\n”,(*a)->data);
  fflush(stdout);
}

int
main(int argc, char **argv)
{
    int x = 0;
    int_t *z = NULL;

       z =
  (int_t*)malloc(sizeof(int_t));
    func7(7,z);
    x = z->data;
    printf("main(a)=%02d\n", x);
    fflush(stdout);
    return 0;
}
```

What would be the output from this code?

a) main(a) = 00
b) main(a) = 07
c) main(a) = unknown
d) Segmentation fault.

Valgrind

One weird trick that will save you from memory errors

- Valgrind is a special debugger program that checks your program for memory allocation errors. Your program needs to be compiled with **-g** first.

- To use it:

valgrind <*whatever you'd normally put to run your program*>

- Your program will be a bit slower, but afterwards, Valgrind will print out a report on any memory leaks or allocation errors.

- If your program stops with a segfault, Valgrind will tell you where it is.

Valgrind Output

"This report means WHAT again?"

- If a program has problems, Valgrind will print out a report where it complains about numerous issues. This is what one section of such a report looks like:

```
Invalid read of size 1
  at 0x7BF6: strlen (in …)
  by 0x1E3B8A: __strcat_chk (in …)
  by 0x100000D79: main (busted_program.c:52)
Address 0x1000121e0 is 0 bytes after a block of size 16 alloc'd
  at 0x6E70: realloc (in …)
  by 0x100000C78: main (busted_program.c:44)
```

- The first bolded line shows where the error is. The second one shows where the associated block of memory was last allocated, reallocated, or freed. Your problem is probably in one of these two places.

More Valgrind Output

"This report means WHAT again?"

- Common Valgrind errors include:
- `Invalid read/write of size n`

  You're trying to read from or write to memory you didn't allocate. Usually this indicates you're trying to read/write past the end of a block you allocated, or you got the block size wrong when resizing it with realloc().
- `Invalid free() / delete / delete[] / realloc()`

  You're trying to free() or realloc() a block of memory that you don't own. This can happen if you try to realloc() or free() a block of memory you already released with free() once.
- `Conditional jump depends on uninitialised value(s)`

  You're trying to use a variable that Valgrind thinks hasn't been given a value yet (and could contain junk data that might mess up your program).

Even More Valgrind Output

“This report means WHAT again?”

- At the end of the report, Valgrind will print something like this:

```
LEAK SUMMARY:
   definitely lost: 3 bytes in 1 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 4,096 bytes in 1 blocks
        suppressed: 25,031 bytes in 373 blocks
Rerun with --leak-check=full to see details of leaked memory
```

- If the ‘definitely lost’ line isn’t 0, you have a memory leak.
  You can rerun Valgrind with the ‘--leak-check=full’ option, as stated above, to get more details.

Practice: Memory Mistakes

Cap’n Ctrlaltdel Returns!

- Cap’n Ctrlaltdel’s back *again!* This time, he’s trying to learn dynamic memory.
- Unsurprisingly, he messed up, and made about every error in the book.
- “pirate_password.c” is in Connex. It works, but it’s got a bunch of memory errors. Can you fix it?
- Tip: The program may have memory-allocation errors, but still run. Run Valgrind to check once it seems like everything’s working.

gdb

Debugger time!

- **gdb** is a debugger that lets you see inside your program as it runs.
- You can control how your program runs, and examine the value of any variable.
- To use **gdb** properly, your program needs to have been compiled with **-g**.

Starting gdb

Are you ready to debug?

- Unlike Valgrind, you'll need to jump through a few hoops to run your program in gdb.
- You can start gdb with just the name of your program to run, but you can't put any arguments after it.

***gdb** ./my_program_that_I_made*

Running Programs in gdb

- When you open GDB, there'll be a gdb command prompt, and nothing else.
- Your program is currently waiting to start.

(gdb>) **run** *[or]* (gdb>) **r**

- The command '**run**', or '**r**' for short, will start your program running at normal speed. If your program has run once already, it will restart it instead.
- If you have a program that's stopped and you want it to pick up from where you left off, use:

(gdb>) **continue** *[or]* (gdb>) **c**

Arguments

Ways to have arguments

- If you want to set or change the arguments your program runs with, you have two ways to do it.
- You can either use **set args** to give your program arguments, or put your arguments after **run** to change the arguments *and* start/restart the program.

(gdb>) **set args** *argument_1 argument_2 ...*

(gdb>) **run** *argument_1 argument_2 ...*

- If you do this again after you previously set arguments, the new ones will overwrite the old ones.

Stopping Programs in gdb

Time to take a ‘break’

- gdb will stop if your program experiences a crash. Instead of exiting, the debugger will stop where the crash occurs so you can investigate.
- If you want to stop at some other point, you can set a breakpoint at a line or function, which will pause your program whenever it hits the breakpoint.

(gdb>) **break** *<line_to_stop_at>*

- If you say ‘break 53’, your program will stop whenever it hits the code on line 53 of the source file. If your program has multiple source files, you might need to say, “mysource.c:53” instead.

More on Breakpoints

When you’ve had enough stopping

- You can remove breakpoints when you’re done using them.
- To remove a breakpoint, use the **clear** command, which takes the line number (or function name) of the breakpoint to remove.

  (gdb>) **clear** *<line_you_put_your_breakpoint_at_before>*

- Basically, whatever you put for the location of the breakpoint when using **break** is the same thing you should put when using **clear**.

Investigating While Paused

Like freeze-framing a DVD

- Once your program is paused, either due to a crash or a breakpoint, you can investigate what's going at the present time.
- If you want to see the value of a variable, you can use the **print** command (**p** for short).

(gdb>) **print** *<variable_to_print>*

(gdb>) **p** *<variable_to_print>*

- You can only print variables that are active in the current stack frame (scope), or global variables.
  (How to switch scopes is outlined later.)

Investigating While Paused

“*I’m* not lost. My house is lost.”

- You can use the **list** command to look at the program’s source code.
- Running **list** with no arguments makes gdb display the source around your current position.
  If you give **list** a line number or function name, it’ll display the source code around that line or function instead.

(gdb>) **list**

(gdb>) **list** *<line_number>*

(gdb>) **list** *<func_name>*

Investigating While Paused

Stack frames and scope

- By default, you'll be in the stack frame for the current line(s) of code. (A stack frame is the context - local variables and everything - for a function. Each time a function is called, a new stack frame is created. When that function returns, that stack frame vanishes.)
- Running **where** or **bt** will show you the current set of stack frames.

  (gdb>) **where** *[or]* (gdb>) **bt**

- **print** only shows the values of variables that are in the current stack frame.
  - You can also use **watch** *var* to to automatically print out the contents of a variable when it changes

Advanced Running

Controlling how your program runs

- Sometimes, you might not want to let a program just continue unchecked after it’s been paused.
- You can use **step**, **next** and **finish** to only run a little at a time.
- **step** runs until the next line of code, and that’s it.
- **next** is like **step**, but gdb won’t enter functions if you have a function call (it’ll run until the next line of code *in the current function*.)

Advanced Running

Controlling how your program runs

- If you want even more control over your program's running, you can use **until.**
- **until** runs the program until it reaches the source line you tell it to stop at.
- Like **break**, if your program has multiple source files, you might need to tell **until** the file name: "my_file.c:22" instead of just "22".

```
(gdb>) until <line_number>
```

gdb cheat sheet

command	what it does	examples
run	Starts the program. If already running, restarts it. If you put text after **run**, runs with the text as arguments.	(gdb>) **run** (gdb>) **run** infile.txt outfile.txt
set args	Sets the program's arguments.	(gdb>) **set args** infile.txt outfile.txt
show args	Shows the program's current arguments.	(gdb>) **show args**
start	Starts the program, then stops at the start of main(). If the program is already running, restarts it. Takes arguments just like **run** or **set args.**	(gdb>) **start** (gdb>) **start** infile.txt outfile.txt
continue	Continues the program when stopped.	(gdb>) **continue**
quit	Quits gdb!	(gdb>) **quit**

gdb cheat sheet

command	what it does	examples
step	Runs the program until the next line of the source.	(gdb>) **step**
next	Runs the program until the next line of the source that's in the *same* function (i.e. it doesn't follow function calls).	(gdb>) **next**
finish	Runs the program until the current function returns.	(gdb>) **finish**
until	Runs the program until the specified line of source, or function definition.	(gdb>) **until** 53 (gdb>) **until** myprog.c:53 (gdb>) **until** myfunc
break	Sets a breakpoint to stop the program at the specified line of source, or function definition.	(gdb>) **break** 53 (gdb>) **break** myprog.c:53 (gdb>) **break** myfunc
clear	Clears a breakpoint you set.	(gdb>) **clear** 53 (gdb>) **clear** myprog.c:53 (gdb>) **clear** myfunc

gdb cheat sheet

command	what it does	examples
print	Prints the value of a variable. Only local variables (current stack frame) or global variables can be printed.	(gdb>) **print** num_iterations
list	Shows you the source code where you are now, or around a source line or function you specify.	(gdb>) **list** (gdb>) **list** 53 (gdb>) **list** myprog.c:53 (gdb>) **list** myfunc
where	Shows the stack frames (list of functions that are currently running) and where the program is in them.	(gdb>) **where**
up	Moves up a stack frame, into the function that called the current function.	(gdb>) **up**
down	Moves down a stack frame, into the function that was called from the current function.	(gdb>) **down**

gdb cheat sheet

command	what it does	examples
call	Call a function in the program immediately! Print any results.	(gdb>) **call** myfunc(arg1,arg2)
return	Force the current function (the stack frame you're in) to return immediately. You can give a return value if you want.	(gdb>) **return** (gdb>) **return** "foo" (gdb>) **return** 213
watch	Sets a 'watchpoint'. A watchpoint stops the program if the variable or expression you give it changes.	(gdb>) **watch** num_iterations (gdb>) **watch** foo+bar
kill	Exits the program being debugged entirely - gdb stays running. If you just want to stop the program you're debugging *right now*, press Ctrl-C. This won't exit the program, but it will stop it.	(gdb>) **kill**