601.220 Intermediate Programming

Multidimensional arrays and GDB

Outline

- Multidimensional arrays
- Debugger tool: gdb

Declaring multi-dimensional arrays

- `<base_type> <name>[dim1_sz][dim2_sz];`
- `<base_type> <name>[dim1_sz][dim2_sz][dim3_sz];`
- `<base_type>`
  `<name>[dim1_sz][dim2_sz][dim3_sz][dim4_sz];`

Two-dimensional arrays

- `int a[row_size][col_size];  //declaration`
- `int table[2][4] = { {1,2,3,4},{5,6,7,8} };`
  - Physically in memory, laid out as one contiguous block of 2*4=8 int-sized locations
  - Array elements are stored in *row-major order*
    - all of row1 comes first, then all of row2

Two-dimensional arrays

- `int table[2][4] = { {1,2,3,4},{5,6,7,8} };`
  - `table` - holds address of first element of the 2D array
  - `table[i][j]` - refers to jth element in ith row of 2D array

Specifying multi-dimensional arrays as function parameters

- First array size need not be specified in formal parameter, but second and following dimensions must be given
  - `void sum_matrix(int list[][4], int numRows);`
  - Can you see why writing `int list[][]` isn't good enough?
  - Passing an array will always be decayed to a pointer, which we are going to talk about

Zoom poll!

Consider the follow array declaration:

```
float grid[2][3];
```

Which of the following pairs of elements are adjacent in memory?

A. `grid[0][2]` and `grid[1][2]`

B. `grid[0][2]` and `grid[1][0]`

C. `grid[0][1]` and `grid[1][0]`

D. More than one pair of elements are adjacent

E. None of the pairs of elements is adjacent

gdb

`gdb`: GNU debugger

`gdb` helps you run your program in a way that allows you to:

- flexibly *pause* and *resume*
- print out the values of variables mid-stream
- see where severe errors like `Segmentation Faults` happen

When using `gdb` (or `valgrind`) we compile with `-g`, which packages up the source code ("debug symbols") along with the executable

gdb

Buggy program:

```c
// str_rev.c:
#include <stdio.h>
#include <string.h>

void string_reverse(char *str) {
  const int len = strlen(str);
  for(int i = 0; i < len; i++) {
    str[i] = str[len-i-1]; // swap characters
    str[len-i-1] = str[i];
  }
}

int main() {
  char reverse_me[] = "AAABBB";
  string_reverse(reverse_me);
  printf("%s\n", reverse_me);
  return 0;
}
```

gdb

We are trying to reverse a string by starting at the left and right extremes, swapping the characters, then continuing inward, swapping as we go until we've reversed the whole thing.

```
$ gcc -o str_rev str_rev.c -std=c99 -pedantic -Wall -Wextra -g
$ ./str_rev
BBBBBB
```

Oops, I expected output to be `BBBAAA`

gdb

We'll use gdb to investigate

Since the problem would seem to be in the `string_reverse` function, I am going to start my program at the beginning and then take small steps forward until I get to the loop.

To do so, we compile with -g flag:

```
gcc -std=c99 -Wall -Wextra -pedantic str_rev.c -o str_rev -g
```

Then we launch debugger using gdb and name of executable:

```
gdb ./str_rev
```

gdb

```
(gdb) break main
Breakpoint 1 at 0x4005ad: file str_rev.c, line 13.
(gdb) run
Starting program: /app/str_rev
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.26-15.f

Breakpoint 1, main () at str_rev.c:13
13        char reverse_me[] = "AAABBB";
```

`break main` because I want to debugger to pause as soon I as get to the beginning of the program, i.e. the `main` function

`run` to start the program, which immediately pauses at top of `main`

After running a command, `gdb` prints out the next line of code in the program

gdb

```
(gdb) next
14      string_reverse(reverse_me);
(gdb) step
string_reverse (str=0x7fffffffe629 "AAABBB") at str_rev.c:5
5             const int len = strlen(str);
```

`next` executes the statement on the current line and moves onto the next. If the statement contains a function call, gdb executes it without pausing.

`step` begins to execute the statement on the current line. If the statement contains a function call, it *steps into* the function and pauses there. Otherwise, it behaves like `next`.

Now we're at the beginning of `string_reverse`

gdb

```
(gdb) n
6       for(int i = 0; i < len; i++) {
(gdb) print len
$2 = 6
```

`n` is short for `next`

`print` prints out the value of a variable. `len` is 6 – that's what we expected. So far so good.

We're about to enter the loop.

gdb

```
(gdb) n
7           str[i] = str[len-i-1]; // swap characters
(gdb) p i
$3 = 0
(gdb) p str[i]
$4 = 65 'A'
(gdb) p str[len-i-1]
$5 = 66 'B'
```

`p` is short for `print`

`i`'s initial value is 0, as expected

The elements we're swapping really are the first `A` and the last `B`, as expected

gdb

Let's execute the swap:

```
(gdb) n
8           str[len-i-1] = str[i];
(gdb) n
6         for(int i = 0; i < len; i++) {
(gdb) p i
$6 = 0
```

Just finished the first iteration; `i` still equals 0

gdb

Let's see if the swap was successful:

```
(gdb) p str[i]
$7 = 66 'B'
(gdb) p str[len-i-1]
$8 = 66 'B'
```

No – the swap fails because I overwrite `str[i]` with the value of `str[len-i-1]` *before* copying it into `str[len-i-1]`

This explains why the result is `BBBBBB`

I need to use a temporary variable to enact a `swap`

gdb

Fixed?:

```c
// str_rev2.c:
#include <stdio.h>
#include <string.h>

void string_reverse(char *str) {
  const int len = strlen(str);
  for(int i = 0; i < len; i++) {
    char temp = str[i];  // swap characters -- FIXED
    str[i] = str[len-i-1];
    str[len-i-1] = temp;
  }
}

int main() {
  char reverse_me[] = "AAABBB";
  string_reverse(reverse_me);
  printf("%s\n", reverse_me);
  return 0;
}
```

gdb

```
$ gcc -o str_rev2 str_rev2.c -std=c99 -pedantic -Wall -Wextra -g
$ ./str_rev2
AAABBB
```

- Still not working! I expected output to be `BBBAAA`

More on gdb

```c
// random_crash.c:
#include <stdio.h>
#include <stdlib.h>

void foo();
void foo() {
    int* crashing_it = NULL;
    if (rand() % 7 == 0) crashing_it[0] = 0;
    else foo();
}

int main(int argc, char* argv[]) {
  if (argc != 2) return -1;
  FILE *fp = fopen(argv[1], "r");
  if (!fp) return -2;
  fclose(fp);
  foo();
  return 0;
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c random_crash.c -g
$ gcc -o random_crash random_crash.o
$ touch input.txt
$ ./random_crash input.txt
Segmentation fault (core dumped)
```

More on gdb

How to locate where the segmentation fault is?

How to pass command line arguments in gdb?

More on gdb

Use `--args` to pass command line arguments

```
gdb --args random_crash input.txt
```

Now `run` as usual

```
(gdb) run
Starting program: /app/random_crash input.txt

Program received signal SIGSEGV, Segmentation fault.
0x00400616 in foo () at random_crash.c: 10
10      if (rand() % 7 == 0) crashing_it[0] = 0;
```

More on gdb

Backtrace the call stack

```
(gdb) backtrace
#0  0x00400616 in foo () at random_crash.c:10
#1  0x00400628 in foo () at random_crash.c:11
#2  0x00400628 in foo () at random_crash.c:11
#3  0x00400628 in foo () at random_crash.c:11
#4  0x00400628 in foo () at random_crash.c:11
#5  0x00400628 in foo () at random_crash.c:11
#6  0x00400628 in foo () at random_crash.c:11
#7  0x00400628 in foo () at random_crash.c:11
#8  0x00400628 in foo () at random_crash.c:11
#9  0x00400628 in foo () at random_crash.c:11
#10 0x00400628 in foo () at random_crash.c:11
#11 0x00400628 in foo () at random_crash.c:11
#12 0x00400628 in foo () at random_crash.c:11
#13 0x00400628 in foo () at random_crash.c:11
#14 0x00400628 in foo () at random_crash.c:11
#15 0x00400628 in foo () at random_crash.c:11
#16 0x004006bd in main (argc=2, argv=0xfffede88)
    at random_crash.c:21
```

More on gdb

Use up and down to navigate the stack

```
(gdb) up
#1  0x00400628 in foo () at random_crash.c:11
11          else foo();

(gdb) down
#0  0x00400616 in foo () at random_crash.c:10
10          if (rand() % 7 == 0) crashing_it[0] = 0;
```

More on gdb

Use `list` to show the current code block

```
(gdb) list
5
6       void foo();
7
8       void foo() {
9         int* crashing_it = NULL;
10        if (rand() % 7 == 0) crashing_it[0] = 0;
11        else foo();
12      }
13
14      int main(int argc, char* argv[]) {
```

gdb help

Type `help` at the `(gdb)` prompt for help

- `(gdb) help running` – for advancing thru program
- `(gdb) help show` – for printing commands

There are *many* gdb commands, so brief "cheat sheets" can help:

- darkdust.net/files/GDB%20Cheat%20Sheet.pdf