

`gdb` for debugging (1)

- `gdb`: the Gnu DeBugger
 - <http://www.cs.caltech.edu/courses/cs11/material/c/mike/misc/gdb.html>
 - Use when program core dumps
 - or when want to walk through execution of program line-by-line
-

`gdb` for debugging (2)

- Before using `gdb`:
 - Must compile C code with additional flag:
`-g`
 - This puts all the source code into the binary executable
- Then can execute as: `gdb myprogram`
- Brings up an interpreted environment

`gdb` for debugging (3)

`gdb> run`

- Program runs...
 - If all is well, program exits successfully, returning you to prompt
 - If there is (*e.g.*) a core dump, `gdb` will tell you and abort the program
-

gdb – basic commands (1)

- Stack backtrace ("**where**")
 - Your program core dumps
 - Where was the last line in the program that was executed before the core dump?
 - That's what the **where** command tells you
-

gdb – basic commands (2)

gdb> where

last call

last call in your code

#0 0x4006cb26 in free () from /lib/libc.so.6

#1 0x4006ca0d in free () from /lib/libc.so.6

#2 0x8048951 in board_updater (array=0x8049bd0,
ncells=2) at 1dCA2.c:148

#3 0x80486be in main (argc=3, argv=0xbffff7b4) at
1dCA2.c:44

#4 0x40035a52 in __libc_start_main () from
/lib/libc.so.6

stack backtrace

gdb – basic commands (3)

- Look for topmost location in stack backtrace that corresponds to your code
 - Watch out for
 - freeing memory you didn't allocate
 - accessing arrays beyond their maximum elements
 - dereferencing pointers that don't point to part of a `malloc()` ed block
-

gdb – basic commands (4)

- **break, continue, next, step** commands
 - **break** causes execution to stop on a given line
`gdb> break foo.c: 100` (setting a breakpoint)
 - **continue** resumes execution from that point
 - **next** executes the next line, then stops
 - **step** executes the next statement
 - goes into functions if necessary (**next** doesn't)
-

gdb – basic commands (5)

- **print** and **display** commands
- **print** prints the value of any program expression

```
gdb> print i
```

```
$1 = 100
```

- **display** prints a particular value every time execution stops

```
gdb> display i
```

gdb – printing arrays (1)

- `print` will print arrays as well

```
int arr[] = { 1, 2, 3 };
```

```
gdb> print arr
```

```
$1 = {1, 2, 3}
```

- N.B. the `$1` is just a name for the result

```
print $1
```

```
$2 = {1, 2, 3}
```

gdb – printing arrays (2)

- `print` has problems with dynamically-allocated arrays

```
int *arr;
```

```
arr = (int *)malloc(3 * sizeof(int));
```

```
arr[0] = 1; arr[1] = 2; arr[2] = 3;
```

```
gdb> print arr
```

```
$1 = (int *) 0x8094610
```

- Not very useful...
-

gdb – printing arrays (3)

- Can print this array by using @ (gdb special syntax)

```
int *arr;
```

```
arr = (int *)malloc(3 * sizeof(int));
```

```
arr[0] = 1; arr[1] = 2; arr[2] = 3;
```

```
gdb> print *arr@3
```

```
$2 = {1, 2, 3}
```

gdb – abbreviations

- Common **gdb** commands have abbreviations

p (same as **print**)

c (same as **continue**)

n (same as **next**)

s (same as **step**)

- More convenient to use when interactively debugging
-