

# CME 212: Tutorial 2

January 25, 2012

- GDB & Valgrind tools for pointers, heap and stack.

# Linux (Mac) memory & debugging utilities

- Memory inspection (shell) utilities in linux
  - `free -m` list memory usage in megabytes
  - `ps aux` lists cpu/memory info on running processes
  - `top` interactive process inspector (h for help)
- Debugging tools:
  - `printf` inside C code:  

```
printf("%p", &i);
```
  - GDB - executes code line by line allowing inspection of variables, memory, code, etc ...
  - Valgrind - simulates an executable detecting memory errors

## Computer memory: 3 segments

- Code/Data segment
  - Keeps all machine instructions for load programs
  - Located near lower address regions i.e. 0x8048424  
`printf("code segment around %p", &main)`
- Stack segment
  - Keeps and maintains function arguments & local variables.
  - Local variables are allocated on stack  
`int a[] = {1,2,3}; // do not return a!`
  - Located in high address regions i.e. 0xbffee6c  
`printf("code stack around %p", &var)`
- Heap segment
  - Located in between the stack and code segments
  - Largest area of memory
  - Memory on heap allocated with `malloc( ... )`
  - Other heap memory management functions are  
`free, calloc, memcpy, memset, realloc ...`

## Printing out memory regions

```
1 void func(char q) {
2     char str[256] = "where_am_i?";
3     printf("memory_contains:_%s_\n", &q + 4);
4 }
5
6 void main(int argc, char** argv) {
7     char q = *argv[1]; // should check argc!
8     func(q);
9 }
```

```
$ gcc memory.c
```

```
$ ./a.out Q
```

```
memory contains: where am i?
```

## Using GDB to inspect code

```
1 void func(char q) {  
2     char str[256] = "where_am_i?";  
3     printf("memory_contains: %s\n", &q + 4);  
4 }
```

```
$ gcc -g memory.c
```

```
$ gdb a.out
```

```
(gdb) set args Q
```

```
(gdb) break memory.c:3
```

```
Breakpoint 1 at 0x8048423: file memory.c, line 3.
```

```
(gdb) run
```

```
Breakpoint 1, func (q=81 'Q') at memory.c:3
```

```
3 printf("memory contains: %s\n", &q + 4);
```

```
(gdb) p &q
```

```
$1 = 0xbfffed3c "Q\367\377\267where am i?"
```

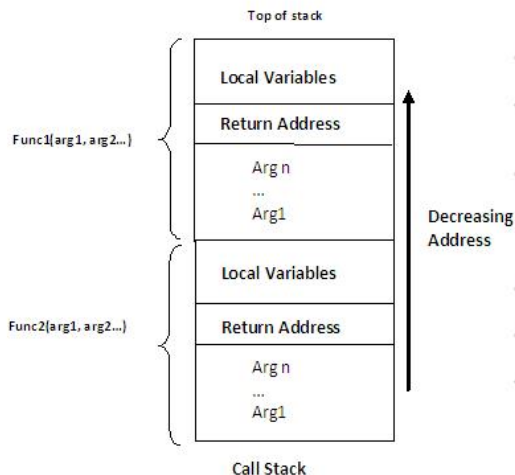
# GDB Cheat sheet

- To compile use the `-g` flag i.e. `gcc -g file.c -o exec`
- Run GDB with `gdb exec`
  - use `set args` followed by arguments (if any) to `main()`
  - to run the program type `run`
  - `list` prints out loaded code
  - to set breakpoint type `break filename.c:4`
  - to quit GDB type `quit`
- The following commands are useful
  - `p var` prints value of variable named `var`
  - `n` goes to execute the next line of code
  - `c` continues execution till next breakpoint
  - `s` steps inside a function call to execute its code
  - `b` lists all break points
  - `d` deletes all break points
  - `d 3` deleted the third break point

## GDB and the stack

- GDB has commands to inspect the stack  
`bt n` prints `n` stack frames (omit `n` for full stack)
- To print address of local (stack) variable  
`p &i`
- You can set variables during execution via  
`set var i = 5`
- You can print the stack (top) pointer with  
`p $sp`
- Moving up and down stack by frame is done with  
`up` and `down`

# Function execution on stack



- **Call Stack:** is a stack that contains a collection of stack frames.
- A **stack frame**, also called as **activation record**, contains information about function call.
- When the program runs, the call stack contains at least one stack frame which is for the main function. A stack frame is created whenever a function is called.
- Arguments are inserted into stack frame from *right to left*.
- *The stack grows from higher addresses to lower addresses.*
- A **stack pointer register** (register 1) is used to mark the current "top" of the stack.

- Question: how would you determine size of stack?

## Crashing the stack

```
1 int crash_stack(int i) {
2     int bytes_in_kb = 1024;
3     int padding = sizeof(void*) + 4*sizeof(int);
4     int numints = (bytes_in_kb - padding)/sizeof(int);
5
6     // allocate kilobyte on heap accounting for
7     // return address, local variables and args
8     int a[ numints ];
9     printf("pushed_stack_down_by_%d_kb_to_%p\n", i, &i);
10
11     return crash_stack(i + 1);
12 }
```

- Call to `crash_stack(1)` segfaults after pushing stack down by 7477 kilobytes which is roughly only 7MB

## Crashing the heap

```
1 int main(int argc, char** argv)
2 {
3     unsigned int bytes_in_kilobyte = 1024;
4
5     void* x = (void *) 1;
6     unsigned int i = 0;
7
8     // when will the loop terminate?
9     while ( x )
10    {
11        x = (void *) malloc( bytes_in_kilobyte );
12        printf("allocated_%d_kilobytes_\n", ++i);
13    }
14    return 0;
15 }
```

- Allocates lots of memory in kilobyte chunks. Run `top` from shell and type `shift+M` to track programs memory usage.

# Valgrind

- Valgrind is a free memory debugging tool for Linux and Mac
- Simulates an executable tracking memory errors like
  - Uninitialized variables,
  - Memory leaks,
  - Buffer overflows, etc
- Does not detect static (stack) memory errors
- Will only detect an error if it occurs during runtime

## Valgrind: uninitialized variables

```
1 int main(int argc, char** argv) {
2     int i;
3     double L1;
4     double tuple[] = {1, -1};
5
6     for ( i = 0; i < 2; ++i )
7         L1 += abs( tuple[i] );
8
9     return (L1 > 0);
10 }
```

```
$ gcc uninitialized.c
```

```
$ valgrind ./a.out
```

```
==9388== Syscall param exit_group(status) contains  
uninitialised byte(s)
```

```
==9388==      at 0x455F7364: _Exit (in /lib/libc-2.14.90.so)
```

```
==9388== Use --track-origins=yes to see where uninitialised  
values come from
```

## Valgrind: buffer overflow

```
1 int main(int argc, char** argv){
2     int i, n = 4;
3     int* ary = malloc(sizeof(int)*n);
4
5     for (i = 0; i <= n; ++i)
6         ary[i] = 1;
7
8     free(ary);
9     return 0;
10 }
```

```
$ gcc overflow.c
```

```
$ valgrind ./a.out
```

```
==9736== Invalid write of size 4
```

```
==9736==      at 0x8048435: main
```

```
==9736==    total heap usage: 1 allocs, 1 frees, 16 bytes
```

```
==9736== All heap blocks were freed -- no leaks are possible
```

```
==9736== ERROR SUMMARY: 1 errors from 1 contexts
```

## Valgrind: memory leak

```
1 void main(int argc, char** argv) {
2     int i, j;
3     int** ary;
4     // initialize array of integer pointers
5     ary = (int**) malloc( 4*sizeof(int*) );
6
7     // initialize rows of all zeros
8     for(i = 0; i < 4; ++i)
9         ary[i] = (int*) calloc(4, sizeof(int) );
10
11     // leaky free ... what is the proper way?
12     free(ary);
13 }
```

```
$ gcc overflow.c
$ valgrind ./a.out
```

```
==9892== LEAK SUMMARY:
==9892==      definitely lost: 64 bytes in 4 blocks
==9892== Rerun with --leak-check=full to see details
```