Memory Management

CS449 Spring 2016

Lifetimes

- Lifetime: time from which a particular memory location is allocated until it is deallocated
- Three types of lifetimes
  - Automatic (within a scope)
  - Static (duration of program)
  - Manual (explicitly controlled by programmer)
- Manual control involves inserting into your code
  - Allocation calls
  - Deallocation calls

Manual Allocation Pros/Cons

- Pros (Flexibility)
  - Can create storage locations on demand without declaring them as variables
  - Useful for constructing dynamic data structures (e.g. linked lists, trees, graphs) whose size and shape can change depending on user input
- Cons (Complexity)
  - Programmer has to think about the behavior of program to determine the lifetimes of locations

C Standard Library Functions

- Takes care of the nitty-gritty details of memory management
  - System calls to request more memory from OS
  - Keeping track of areas of allocated memory
  - Keeping track of areas of free memory
  - Searching for suitable area to allocate memory
- Programmer only has to worry about *when* to allocate/deallocate and not *how*
- Just include functions declared in <stdlib.h>

Allocation Functions

- `void *malloc(size_t size)`
  - Allocates `size` bytes and returns a pointer to the allocated memory (NULL on failure)
  - Memory is not cleared
- `void *calloc(size_t nmemb, size_t size)`
  - Allocates `nmemb` * `size` bytes and returns a pointer to the allocated memory (NULL on failure)
  - Memory is set to 0
- Check return value for out-of-memory error

Deallocation Function

- `void free(void *ptr)`
  - Frees the memory space pointed to by `ptr`, which must have been previously allocated
  - Counterpart for both `malloc`() and `calloc`()
  - If `free`() has already been called on the same location, behavior is undefined

Re-Allocation Function

- `void *realloc(void *ptr, size_t size)`
  - Changes size of memory block pointed to by `ptr` to `size` bytes and returns that pointer (NULL on failure)
  - Newly allocated memory will be uninitialized
  - If `ptr` is NULL, same as `malloc`
  - If `size` is zero, and `ptr` is not NULL, same as `free`
- Use when you want to resize previously allocated memory without losing its contents

Malloc/Free Example

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int *p, i, length = atoi(argv[1]);
  p = (int *)malloc(length * sizeof(int));
  srand((unsigned int)time(NULL));
  for(i = 0; i < length; i++)
    p[i] = rand();
  for(i = 0; i < length; i++)
    printf("%d \n", p[i]);
  free(p);
  return 0;
}
```

```
>> ./a.out 5
729751416
264693780
884704288
90101471
690008936
```

Malloc/Free Example

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int *p, i, length = atoi(argv[1]);
  p = (int *)malloc(length * sizeof(int));
  srand((unsigned int)time(NULL));
  for(i = 0; i < length; i++)
    p[i] = rand();
  for(i = 0; i < length; i++)
    printf("%d \n", p[i]);
  free(p);
  return 0;
}
```

- Calculate the number of bytes of memory you want to allocate using `sizeof`
- Don't forget to `free` memory once you ware done using it
- The lifetime of the locations allocated using `malloc()` is independent of pointer `p`
  - Pointer `p` is deallocated at the end of scope `main()`
  - The malloced locations lifetime ends on `free()`

Memory Management Strategy

- Allocation is relatively easy (just like `new` in Java): allocate memory before using it
- Basic Rules for deallocation
  - Memory should be freed when it can no longer be reached by the program
    - That is, when the *last pointer* referencing it is updated
    - Freeing before: can lead to accessing freed memory
    - Freeing after: last reference is gone, so no way to free!
    - This is actually the rule used by the Java garbage collector
  - How do you know if you are the last pointer?
    - Reference counting (how many point to you) can be helpful

Common Memory Errors

- Memory Leak
  - Forgetting to free unused memory after pointer update
  - Result: Steady rise in memory consumption leading to degraded performance and eventual out-of-memory error
- Double Free
  - Freeing the same memory location twice
  - Result: Undefined. Depends on C stdlib implementation.
- Dangling Pointer
  - Accessing memory that has already been freed
  - Result: Potential memory corruption when that memory is allocated for something else
- Out-of-bounds Access
  - Accessing memory beyond the allocation boundary
  - Result: Potential memory corruption when memory beyond boundary is allocated for something else
- Memory errors are typically Heisenbugs (non-deterministic bugs)

Valgrind

- Diagnostic tool that detects common memory errors (among other things) at runtime
- Command: valgrind [options] <program>
- Not perfect.
  - Can miss errors (sometimes)
  - Can report errors when there are none (rarely)
- Not a replacement for sound programming

Valgrind Example

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  char *p = malloc(100);
  return 0;
}
```

```
>> gcc –g main.c
>> ./a.out
>> valgrind --leak-check=full --track-origins=yes ./a.out
==32563== HEAP SUMMARY:
==32563==     in use at exit: 100 bytes in 1 blocks
==32563==   total heap usage: 1 allocs, 0 frees, 100 bytes allocated
==32563==
==32563== 100 bytes in 1 blocks are definitely lost in loss record 1 of 1
==32563==    at 0x4A069EE: malloc (vg_replace_malloc.c:270)
==32563==    by 0x4004DC: main (main.c:5)
```

- The “-g” option given to GCC inserts debug symbols to binary, enabling valgrind to locate the errors more accurately

How does malloc interact with the OS?

- C library File I/O functions were built on top of 5 system calls: open, read, write, lseek, close
- C library memory management functions are built on top of 2 system calls: brk and mmap
- `int brk(void *addr)`
  - Changes program break (location of the end of the process's data segment)
  - Increasing program break ➔ allocates memory
  - Decreasing program break ➔ deallocates memory

Tracing System Calls for malloc

- strace ./a.out
  (a.out is the text file dumper run on a Hello World main.c file)

```
(116) thot $ strace ./a.out 2> /tmp/a.err && cat /tmp/a.err
open("main.c", O_RDONLY)                = 3
read(3, "#include <stdio.h>\nint main()\n{\n"..., 4096) = 75
write(1, "#include <stdio.h>\n", 19)    = 19
write(1, "int main()\n", 11)            = 11
write(1, "{\n", 2)                      = 2
write(1, "  printf(\"Hello world!\\n\");\n", 28) = 28
write(1, "  return 0;\n", 12)           = 12
write(1, "}\n", 2)                      = 2
write(1, "\n", 1)                       = 1
read(3, "", 4096)                       = 0
close(3)                                = 0
```

- Notice the difference in buffering for “read” and “write”

Tracing System Calls for malloc

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
  int i = 0;
  for(i = 0; i < 100; i++) {
    malloc(4096);
  }
  return 0;
}
```

```
>> strace ./a.out
[sic]
brk(0)                                  =
0x601000
brk(0x623000)                                =
0x623000
brk(0x644000)                                =
0x644000
brk(0x665000)                                =
0x665000
brk(0x686000)                                =
0x686000
[sic]
```

Practice Problem 1

- Write a program in C, which reads from standard input an integer ***N*** and generates an array of ***N*** random numbers;
- Write a function in C, which gets an array of random numbers as input parameter and returns the number of occurrences of the second biggest/smallest number in the array.
- Example: ***{-1, 0, 1, 2, 0, 0, 1, 1 ,1 ,1, -1}***; ***3*** for the second smallest and ***5*** for the second biggest.

Practice Problem 2

- Write a library in C, which implements rational numbers. The numbers should be represented as a ratio of numerator (integer) and denominator (integer). Write functions for the following operations of two rational numbers – addition, subtraction, multiplication, division, comparison, simplification.

Practice Problem 3

- Write a program in C, which reads from standard input an integer ***N***, generates an 2 arrays of ***N*** integer numbers – $A_1, A_2, \ldots, A_N$ and $B_1, B_2, \ldots, B_N$. The numbers in the 2 arrays should be read from the standard input. They are the catheti of ***N*** right-angled triangles.
- Write a function in C, which finds the hypotenuse of a triangle. Use this function to find the index of a triangle with the longest hypotenuse.
- You can use function: **double sqrt(double x)** from **math.h** library to calculate square root.

Practice Problem 4

- Write a function in C, which gets an integer ***N*** as input parameter and returns an array of 3-digit numbers for which the sum of the digits is equal to ***N***.
- Example: ***N=2; {110, 200}***
- *This problem can be solved using recursion*

Practice Problem 5

- Write a function in C, which gets an integer ***N*** as input parameter. The function should read from standard input numbers until their sum exceeds ***N***. Once ***N*** is exceeded, the function should print the numbers in sorted order.
- N.B. Negative numbers are valid input.