Dynamic memory allocation in C

(Reek, Ch. 11)
Overview of memory management

- Stack-allocated memory
  - When a function is called, memory is allocated for all of its parameters and local variables.
  - Each active function call has memory on the stack (with the current function call on top)
  - When a function call terminates, the memory is deallocated (“freed up”)

- Ex: `main()` calls `f()`,
  - `f()` calls `g()`
  - `g()` recursively calls `g()`
Overview of memory management

- Heap-allocated memory
  - This is used for *persistent* data, that must survive beyond the lifetime of a function call
    - global variables
    - dynamically allocated memory – C statements can create new heap data (similar to new in Java/C++)
  - Heap memory is allocated in a more complex way than stack memory
  - Like stack-allocated memory, the underlying system determines where to get more memory – the programmer doesn’t have to search for free memory space!
Allocating new heap memory

void *malloc(size_t size);

- Allocate a block of size bytes, return a pointer to the block (NULL if unable to allocate block)

void *calloc(size_t num_elements, size_t element_size);

- Allocate a block of num_elements * element_size bytes, initialize every byte to zero, return pointer to the block (NULL if unable to allocate block)

Note: void * denotes a generic pointer type
Allocating new heap memory

void *realloc(void *ptr, size_t new_size);

- Given a previously allocated block starting at ptr,
  - change the block size to new_size,
  - return pointer to resized block
    - If block size is increased, contents of old block may be copied to a completely different region
    - In this case, the pointer returned will be different from the ptr argument, and ptr will no longer point to a valid memory region
- If ptr is NULL, realloc is identical to malloc

Note: may need to cast return value of malloc/calloc/realloc:
char *p = (char *) malloc(BUFFER_SIZE);
Deallocating heap memory

```c
void free(void *pointer);
```

- Given a pointer to previously allocated memory,
  - put the region back in the heap of unallocated memory

- Note: easy to forget to free memory when no longer needed...
  - especially if you’re used to a language with “garbage collection” like Java
  - This is the source of the notorious “memory leak” problem
  - Difficult to trace – the program will run fine for some time, until suddenly there is no more memory!
Checking for successful allocation

- Call to malloc might fail to allocate memory, if there’s not enough available
- Easy to forget this check, annoying to have to do it every time malloc is called...

Reek’s solution:
#define malloc DON’T CALL malloc DIRECTLY!
#define MALLOC(num,type) (type *)alloc((num)*sizeof(type))
extern void *alloc(size_t size);

Garbage inserted into source code if programmer uses malloc

Use MALLOC instead...
Scales memory region appropriately
(Note use of parameters in #define)
Also, calls “safe” alloc function
Checking for successful allocation

- implementation of alloc:
  
  ```c
  #undef malloc

  void *alloc(size_t size) {
    void *new_mem;
    new_mem = malloc(size);
    if (new_mem == NULL) exit(1);
    return new_mem;
  }
  
  Nice solution – as long as “terminate the program” is always the right response
  ```
Memory errors

- Using memory that you have not initialized
- Using memory that you do not own
- Using more memory than you have allocated
- Using faulty heap memory management
Using memory that you have not initialized

- Uninitialized memory read
- Uninitialized memory copy
  - not necessarily critical – unless a memory read follows

```c
void foo(int *pi) {
    int j;
    *pi = j;
    /* UMC: j is uninitialized, copied into *pi */
}
void bar() {
    int i=10;
    foo(&i);
    printf("i = %d\n", i);
    /* UMR: Using i, which is now junk value */
}
```
Using memory that you don’t own

- Null pointer read/write
- Zero page read/write

```c
typedef struct node {
    struct node* next;
    int val;
} Node;

int findLastNodeValue(Node* head) {
    while (head->next != NULL) { /* Expect NPR */
        head = head->next;
    }
    return head->val; /* Expect ZPR */
}
```

What if head is NULL?
Using memory that you don’t own

- Invalid pointer read/write
  - Pointer to memory that hasn’t been allocated to program

```c
void genIPR() {
    int *ipr = (int *) malloc(4 * sizeof(int));
    int i, j;
    i = *(ipr - 1000); j = *(ipr + 1000); /* Expect IPR */
    free(ipr);
}

void genIPW() {
    int *ipw = (int *) malloc(5 * sizeof(int));
    *(ipw - 1000) = 0; *(ipw + 1000) = 0; /* Expect IPW */
    free(ipw);
}
```
Using memory that you don’t own

- Common error in 64-bit applications:
  - Ints are 4 bytes but pointers are 8 bytes
  - If prototype of `malloc()` not provided, return value will be cast to a 4-byte `int`

/*Forgot to `#include <malloc.h>, <stdlib.h>` in a 64-bit application*/
void illegalPointer() {
    int *pi = (int*) malloc(4 * sizeof(int));
    pi[0] = 10; /* Expect IPW */
    printf("Array value = %d\n", pi[0]); /* Expect IPR */
}
Using memory that you don’t own

- Free memory read/write
  - Access of memory that has been freed earlier

```c
int* init_array(int *ptr, int new_size) {
    ptr = (int*) realloc(ptr, new_size*sizeof(int));
    memset(ptr, 0, new_size*sizeof(int));
    return ptr;
}
```

```c
int* fill_fibonacci(int *fib, int size) {
    int i;
    /* oops, forgot: fib = */ init_array(fib, size);
    /* fib[0] = 0; */ fib[1] = 1;
    for (i=2; i<size; i++)
        fib[i] = fib[i-1] + fib[i-2];
    return fib;
}
```

Remember: `realloc` may move entire block

What if array is moved to new location?
Using memory that you don’t own

- Beyond stack read/write

```c
char *append(const char* s1, const char *s2) {
    const int MAXSIZE = 128;
    char result[MAXSIZE];
    int i=0, j=0;
    for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++, j++) {
        result[i] = s1[j];
    }
    for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++, j++) {
        result[i] = s2[j];
    }
    result[++i] = '\0';
    return result;
}
```

- `result` is a local array name – stack memory allocated
- Function returns pointer to stack memory – won’t be valid after function returns
Using memory that you haven’t allocated

- Array bound read/write

```c
void genABRandABW() {
    const char *name = "Safety Critical";
    char *str = (char*) malloc(10);
    strncpy(str, name, 10);
    str[11] = '\0'; /* Expect ABW */
    printf("%s\n", str); /* Expect ABR */
}
```
Faulty heap management

- Memory leak

```c
int *pi;
void foo() {
    pi = (int*) malloc(8*sizeof(int));
    /* Allocate memory for pi */
    /* Oops, leaked the old memory pointed to by pi */
    ...
    free(pi); /* foo() is done with pi, so free it */
}
void main() {
    pi = (int*) malloc(4*sizeof(int));
    /* Expect MLK: foo leaks it */
    foo();
}
```
Faulty heap management

- Potential memory leak
  - no pointer to the beginning of a block
  - not necessarily critical – block beginning may still be reachable via pointer arithmetic

```c
int *plk = NULL;
void genPLK() {
    plk = (int *) malloc(2 * sizeof(int));
    /* Expect PLK as pointer variable is incremented past beginning of block */
    plk++;
}
```
Faulty heap management

- Freeing non-heap memory
- Freeing unallocated memory

```c
void genFNH() {
    int fnh = 0;
    free(&fnh); /* Expect FNH: freeing stack memory */
}

void genFUM() {
    int *fum = (int *) malloc(4 * sizeof(int));
    free(fum+1); /* Expect FUM: fum+1 points to middle of a block */
    free(fum);
    free(fum); /* Expect FUM: freeing already freed memory */
}
```
Tools for analyzing memory management

- Purify: runtime analysis for finding memory errors
  - dynamic analysis tool: collects information on memory management while program runs
  - contrast with static analysis tool like lint, which analyzes source code without compiling, executing it
Reference