CSE320 System Fundamentals II

Dynamic Memory Allocation I

YOUNGMIN KWON / TONY MIONE

Announcements

Reading: Text 9.9-9.11

Exam 1 :

- Thur, 10/17/2024 – During class period
- Covers through the next lecture!
- Closed book but A4 note sheet written in your own hand (both sides) is allowed during test.

Acknowledgements

Some slides provided by Dr Yoon Seok Yang

Dynamic Memory Allocation

Why dynamic memory allocation

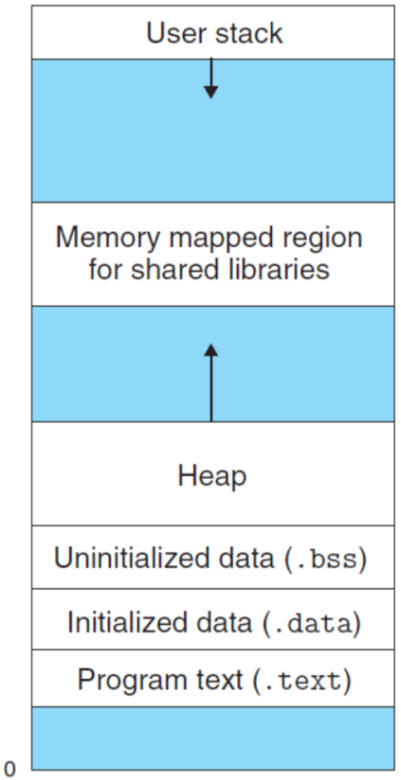
- Suppose that you are writing a program that sorts as many words as users provide.
- How much memory should we prepare in advance when we are writing the program.
  - Max number of words * Max length of each word?
  - It can be a waste of space.
  - How much allocation is enough?

Dynamic Memory Allocation

Solution
- Allocate memory as necessary
- Free memory when not using it

Declared in `stdlib.h`, are
- **`malloc`** that allocates a memory space in the heap area
- **`free`** that deallocates the memory

```
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
```



Heap memory, often referred to simply as the "heap," is a region of a computer's memory that is dynamically allocated for use by programs during runtime. It is distinct from the stack, another area of memory used by programs, and is managed by the operating system's memory manager or allocator

Dynamic Memory Allocation

Additional C runtime memory allocation functions

- **calloc** that allocates a memory space in the heap area and initializes it to all zero values
- **realloc** increases the size of an allocated block.
  - If it can be done at the same block location, the returned address matches the pointer provided.
  - Otherwise, a new address is given for the allocated space and the contents are copied to the new area.

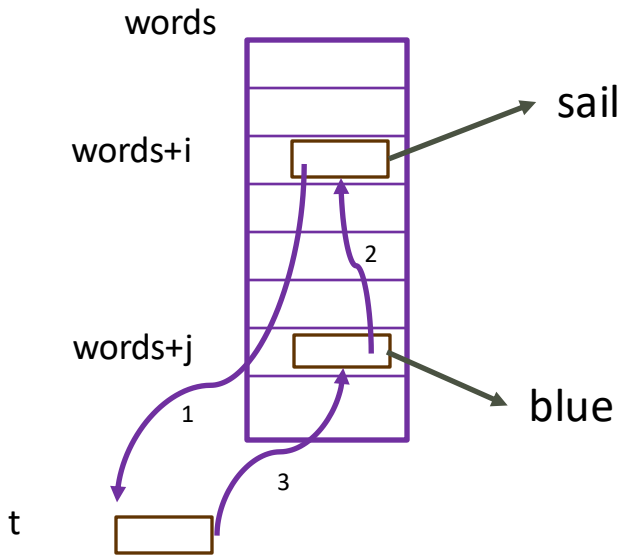
```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

Example 1: read words and sort them

```c
// sort.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void ReadWords(char ***pwords, int *pn) {
    int i, n;
    char **words;
    char word[100];

    printf("Number of words: ");
    scanf("%d", &n);

    words = (char**)malloc(n * sizeof(char*)); //dynamic mem allocation
    for(i = 0; i < n; i++) {
        printf("Word %d of %d: ", i+1, n);
        scanf("%99s", word);
        words[i] = strdup(word); //malloc + strcpy
    }
    *pwords = words; //unlike stack vars, heap memory outlives the function
    *pn = n;
}
```

In C, the strdup function is used to create a dynamically allocated copy of a null-terminated string. It allocates memory for the new string and copies the contents of the input string into the newly allocated memory.

```c
void Swap(char **p, char **q) {       //swap two strings pointers
    char *t;
    t = *p, *p = *q, *q = t;
}

void SortWords(char **words, int n) {
    int i, j;
    for(i = 0; i < n; i++)
        for(j = i + 1; j < n; j++)
            if(strcmp(words[i], words[j]) > 0)
                Swap(words+i, words+j);
}

void PrintWords(char **words, int n) {
    int i;
    printf("Words\n");
    for(i = 0; i < n; i++)
        printf("%2d: %s\n", i, words[i]);
}
```



```c
void FreeWords(char **words, int n) {
    int i;
    for(i = 0; i < n; i++)
        free(words[i]);     //free the mem alloc'd by strdup
    free(words);   //free the mem alloc'd by malloc
}

int main() {
    int n;
    char **words;

    ReadWords(&words, &n);
    PrintWords(words, n);
    SortWords(words, n);
    PrintWords(words, n);
    FreeWords(words, n);
}
```

Example 2: Linked List [list.h]

```
// list.h
#ifndef __LIST__
#define __LIST__
#define offsetof(st, m)         ((size_t) &(((st *)0)->m))
#define containerof(ptr, st, m) ((st *) (((char*)(ptr)) - offsetof(st, m)))
struct List {
    struct List *prev, *next;
};
void list_init_head(struct List *head);
int list_is_empty(struct List *head);
int list_size(struct List *head);
void list_add_to_prev(struct List *pos, struct List *list);
void list_add_to_next(struct List *pos, struct List *list);
struct List* list_remove(struct List *list);
void list_add_to_last(struct List *head, struct List *list);
void list_add_to_first(struct List *head, struct List *list);
struct List* list_remove_last(struct List *head);
struct List* list_remove_first(struct List *head);
struct List* list_find(struct List *head, void *data,
                       int (*comp)(struct List *list, void *data));
#endif
```

[list.c]

```c
// list.c
#include <stdio.h>
#include <stdlib.h>
#include "list.h"
void list_init_head(struct List *head) {
    head->next = head->prev = head;
}
int list_is_empty(struct List *head) {
    return head->next == head;
}
int list_size(struct List *head) {
    int count = 0;
    struct List *list;
    for(list = head->next; list != head; list = list->next)
        count++;
    return count;
}
```

[list.c]

```
void list_add_to_prev(struct List *pos, struct List *list) {
    list->next = pos;
    list->prev = pos->prev;
    pos->prev->next = list;
    pos->prev = list;
}
void list_add_to_next(struct List *pos, struct List *list) {
    list_add_to_prev(pos->next, list);
}
struct List* list_remove(struct List *list) {
    list->prev->next = list->next;
    list->next->prev = list->prev;
    list->next = list->prev = NULL;
    return list;
}
```

[list.c]

```
void list_add_to_last(struct List *head, struct List *list) {
    list_add_to_prev(head, list);
}
void list_add_to_first(struct List *head, struct List *list) {
    list_add_to_next(head, list);
}
struct List* list_remove_last(struct List *head) {
    return list_remove(head->prev);
}
struct List* list_remove_first(struct List *head) {
    return list_remove(head->next);
}
struct List* list_find(struct List *head, void *data,
                       int (*comp)(struct List *list, void *data)) {
    struct List *pos;
    for(pos = head->next; pos != head; pos = pos->next)
        if(comp(pos, data))
            return pos;
    return NULL;
}
```

[sort_list.c]

```c
//sort_list.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "list.h"
#define ORDERED_INSERT  0

typedef struct Person {
    char *name;
    long id;
    struct List list;
} Person;
```

[sort_list.c]

```
Person* NewPerson(char *name, long id) {
    Person *p = //TODO 1: allocate mem for p of size sizeof(Person)
    p->name = strdup(name);
    p->id = id;
    return p;
}

void FreePerson(Person *p) {
    //TODO 1: free p->name
    //TODO 1: free p
}

int CompareName(struct List *list, void *name) {
    return strcmp(  containerof(list, struct Person, list)->name,
                    (char*)name) > 0;
}
```

[sort_list.c]

```
void ReadNames(struct List *head) {
    long id = 0;
    printf("Enter names or q to stop.\n");
    while(1) {
        char name[100];
        scanf("%99s", name);
        if(strcmp(name, "q") == 0)
            break;

        Person *p = NewPerson(name, id++);
#if ORDERED_INSERT
        struct List *pos = list_find(head, name, CompareName);
        if(pos != NULL) //TODO 2:
        else            //TODO 2:
#else
        list_add_to_last(head, &p->list);
#endif
    }
}
```

[sort_list.c]

```
void Swap(char **p, char **q) {
    char *t;
    t = *p, *p = *q, *q = t;
}

void SortList(struct List *head) {
    struct List *i, *j;
    for(i = head->next; i != head; i = i->next) {
        char **name_i = &containerof(i, struct Person, list)->name;
        for(j = i->next; j != head; j = j->next) {
            char **name_j = &containerof(j, struct Person, list)->name;
            if(strcmp(*name_i, *name_j) > 0)
                Swap(name_i, name_j);
        }
    }
}
```

[sort_list.c]

```
void PrintList(struct List *head) {
    struct List *pos;
    printf("Person list:\n");
    for(pos = head->next; pos != head; pos = pos->next) {
        Person *person = containerof(pos, struct Person, list);
        printf("%2ld: %s\n", person->id, person->name);
    }
}

void FreeList(struct List *head) {
    while(!list_is_empty(head)) {
        struct List *pos = list_remove_first(head);
        Person *person = //TODO 1: get person from pos
        FreePerson(person);
    }
}
```

[sort_list.c]

```
int main() {
    struct List head;
    list_init_head(&head);

    ReadNames(&head);
    PrintList(&head);
#if ORDERED_INSERT == 0
    SortList(&head);
    PrintList(&head);
#endif
    FreeList(&head);

    return 0;
}
```

Common Memory Related Bugs

There are numerous mistakes even experienced programmers make when working with dynamic memory allocators.

- Dereferencing Bad Pointers
- Reading Uninitialized Memory
- Allowing Stack Buffer Overflows
- Assuming objects and pointers are the same size
- Off-by-One Errors
- Referencing a Pointer rather than the object it points to
- Misunderstanding Pointer Arithmetic
- Referencing Non-existant Variables
- Referencing Data in Free Heap Blocks
- Introducing Memory Leaks

Common Memory Related Bugs

- Dereferencing Bad Pointers

  Ex:

```
scanf("%d", val);   // shoulds be scanf("%d", &val);
```

- Reading Uninitialized Memory
- Allowing Stack Buffer Overflows

  Ex:

```
void bufoverflow() {
   char buf[64];

   gets(buf);  // gets() does not limit data
   return;
}
```

The gets() function in C is used to read a line of text from the standard input (usually the keyboard) and stores it as a null-terminated string in the provided character array. However, it is important to note that gets() is considered highly unsafe and is not recommended for use in modern C programming. This is because it does not provide any way to limit the size of the input, which can lead to buffer overflows.

Common Memory Related Bugs

- Assuming objects and pointers are the same size

Ex:

```
int **A = (int **)Malloc(n * sizeof(int));    // should be sizeof(int *)
```

- Off-by-One Errors

Ex:

```
void makeArray2() {
    int **A = (int **)Malloc(n * sizeof(int *));
    for (i = 0; i <= n; i++) {    // use A
     …
}
```

- Referencing a Pointer rather than the object it points to

Ex:

```
int *binheapDelete(int **binheap, int *size) {

…

   *size--;      // !!! Should be (*size)--; The code changes the value of the pointer not the contents
```

Common Memory Related Bugs

- (*size)-- is used when you want to modify the value at the memory location pointed to by the pointer size (assuming size is a pointer to an integer).
- *size-- is used when you want to access the value at the current memory location pointed to by size, use that value in an expression, and then decrement the pointer to point to the previous location in a sequence (assuming size is a pointer to an array or sequence).

The key distinction is whether you want to modify the value at the memory location ((*size)--) or access the value and update the pointer (*size--).

Common Memory Related Bugs

- Misunderstanding Pointer Arithmetic

Ex:

```
int *search(int *p, int val) {
   while (*p && *p ~= val) {
      p += sizeof(int);   // Should be just p++;
}
```

- Referencing Non-existant variables

Ex:

```
int *stackref() {
    int val = 5;
    return &val;
}
```

Common Memory Related Bugs

- Referencing data in free heap blocks

Ex:

```
int *binheapDelete(int **binheap, int *size) {
…
   *size--;     // !!! Should be (*size)--; The code changes the value of the pointer not the contents
```

- Introducing Memory Leaks

Ex:

```
void leak(int n) {
   int *x = (int *)malloc(n*sizeof(int));
   return;  // x is never freed!
}
```

Questions?