CSE 374 - Week 4 (Fri)
Dynamic Memory Allocation

Instructor: Andrew Hu
TAs: Jim Limprasert, Kaelin Laundry, Keith Jiang, Nick Durand, Simon McFarlane, Yitian Hao
Administrivia

- Anonymous *Canvas* Mid-Quarter Survey to be released soon
  - We can't see who wrote what (anonymous), but we can see if you filled it out
- You get participation grade for completing it, but you *must* complete it by the due date
- Due a week from today, *Friday July 24*
Memory Allocation So Far

- "Allocation" refers to any way of asking for memory from the operating system
- So far, we have seen two kinds of memory allocation:

```c
int counter = 0;  // global var

int main(int argc, char** argv) {
    counter++;
    printf("count = %d\n", counter);
    return EXIT_SUCCESS;
}

int foo(int a) {
    int x = a + 1;  // local var
    return x;
}

int main(int argc, char** argv) {
    int y = foo(10);  // local var
    printf("y = %d\n", y);
    return EXIT_SUCCESS;
}
```
Dynamic Allocation

- Situations where static and automatic allocation aren’t sufficient:
  - Need memory that persists across multiple function calls
  - Memory size is not known in advance to the caller
  - Something like...

```c
char* ReadFile(char* filename) {
    int size = GetFileSize(filename);
    char* buffer = AllocateMem(size);

    ReadFileIntoBuffer(filename, buffer);
    return buffer;
}
```
Dynamic Allocation

- What we want is *dynamically*-allocated memory
  - Your program explicitly requests a new block of memory
    - The language allocates it at runtime, perhaps with help from OS
  - Dynamically-allocated memory persists until either:
    - Your code explicitly deallocated it (*manual* memory management)
    - A garbage collector collects it (*automatic* memory management)

- C requires you to manually manage memory
  - Gives you more control, but causes headaches
Multiple Ways to Store Program Data

- **Static global data**
  - *Fixed size* at compile-time
  - Entire *lifetime of the program*

- **Stack-allocated data**
  - Local/temporary variables
  - *Known lifetime* (deallocated on `return`)

- **Dynamic (heap) data**
  - Size known only at runtime (*i.e. based on user-input*)
  - Lifetime known only at runtime (long-lived data structures)

```c
int array[1024];

void foo(int n) {
    int tmp;
    int local_array[n];

    int* dyn = (int*)malloc(n*sizeof(int));
}
```
Dynamic Memory Allocation

- **Dynamic memory allocators** acquire memory at run time
  - For data structures whose size (or lifetime) is known only at runtime
  - Allocated in the process' heap:

- **Types of allocators**
  - *Explicit allocator:* programmer allocates and frees space
    - Example: `malloc` and `free` in C
  - *Implicit allocator:* programmer only allocates space (no free)
    - Example: `new` in Java
Address Space Visualization

- Higher addresses on top
- Lower addresses on bottom
- Stack grows down as we call functions
- Heap grows up as we dynamically allocate more memory
- Code, globals, and string literals stored below the heap

- Program code (.text)
- Initialized data (.data)
- Heap (via malloc)

0x00.. → 0xFF..
The Heap

- The Heap is a large pool of available memory used to hold dynamically-allocated data
  - `malloc` allocates chunks of data in the Heap; `free` deallocates those chunks
  - `malloc` maintains bookkeeping data in the Heap to track allocated blocks
Allocating Memory in C

- Need to \#include <stdlib.h>

- `void* malloc(size_t size)`
  - Allocates a continuous block of \texttt{size} bytes of \texttt{uninitialized} memory
  - Returns a pointer to the beginning of the allocated block
    - Returns \texttt{NULL} if allocation failed or \texttt{size}==0
  - Different blocks not necessarily adjacent
  - Read/writing memory next to the returned block is \texttt{undefined behavior}
  - \texttt{void*} means a pointer to any type (e.g. \texttt{int, float, char})
malloc()

- General usage:
  \[
  \text{var} = (\text{type}*) \text{malloc}(\text{size in bytes})
  \]

- \text{sizeof} makes code portable to different machines

```c
// allocate a 10-float array
float* arr = (float*) malloc(10*\text{sizeof}(\text{float}));
if (arr == NULL) {
    \text{return} \text{errno};
}
...  // do stuff with arr
```
Reading Uninitialized Memory

- Unlike Java, in C variables are uninitialized by default
  - Meaning that they could hold any value before we first write to them

```c
int i;
printf("%d\n", i);  // Invalid read!
i = 374;
printf("%d\n", i);  // OK, was initialized to 374

float* fptr = (float*) malloc(sizeof(float));
if (fptr == NULL) {
    return errno;
}
printf("%f\n", *fptr);  // Invalid read!
```
calloc()

- General usage:

  ```
  var = (type*) calloc(num, bytes per element)
  ```

- Like `malloc`, but also zeros out the block of memory
  - i.e. initializes memory to all 0s
  - Slightly slower; but useful for non-performance-critical code
  - `malloc` and `calloc` are found in `stdlib.h`
calloc Example

// allocate a 10-double array
double* arr = (double*) calloc(10, sizeof(double));
if (arr == NULL) {
    return errno;
}
printf("%f\n", arr[0]); // OK, will print 0.00000

void* my_calloc(int size, int num) {
    char* arr = malloc(size * num);
    if (arr == NULL) return NULL;
    for (int i = 0; i < size*num; i++) {
        arr[i] = 0;
    }
    return arr;
}
Freeing Memory in C

- Need to `#include <stdlib.h>`
- `void free(void* p)`
  - Releases whole block pointed to by `p` to the pool of available memory
  - Pointer `p` must be the address *originally* returned by `m/c/realloc` (*i.e.* beginning of the block), otherwise system exception raised
  - Don’t call `free` on a block that has already been released (double free)
  - Pointer is unaffected by call to `free`
    - Defensive programming: can set pointer to `NULL` after freeing it
free()

- Usage: \textcolor{blue}{\textbf{free(}pointer\textbf{);}}

\begin{verbatim}
float* arr = (float*) malloc(10*sizeof(float));
if (arr == NULL)
    return errcode;
...
    // do stuff with arr
free(arr);
arr[0] = 1; // BAD
arr = NULL;  // OPTIONAL
free(arr);   // Bad! Double free is undefined
\end{verbatim}
Poll Question: PollEv.com/andrewhu
Poll Question (PollEv.com/andrewhu)

Which is a **correct** use of malloc and/or free?

A. ```
int x[] = {1, 2, 3};
int sum = x[0]+x[1]+x[2];
free(x);
``` 

B. ```
long* make_array() {
return (long*)malloc(sizeof(long)*5);
}
// later...
long* val = make_array();
free(val);
``` 

C. ```
// Assume arr is a malloc'd int*
int* arr = random_arr(LEN);
long* p = find_int(42, arr, LEN);
int idx = my_arr - p;
free(p);
``` 

D. ```
char** strings =
(char**)malloc(sizeof(char)*5);
// use strings...
free(strings);
```
Summary: Common Memory Errors

- Dereferencing a non-pointer
  \[ \text{long } x = 123; \]
  \[ \text{*(char*)} x \]
- Accessing freed memory
- Double free
- Forgetting to free memory ("memory leak")
- Out-of-bounds access
- Reading memory before initialization
  \[ \text{calloc} \]
- Wrong allocation size
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size * sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```

### Heap and Stack Example

#### Stack
- Main
- Copy
- Alloc

#### Heap (malloc/free)
- Allocates memory on the heap using `malloc`
- Frees memory using `free`

#### OS kernel [protected]
- Stack
- Heap (malloc/free)
- Read/Write Segment
- Read-Only Segment
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;

    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;

    for (i = 0; i < size; i++)
        a2[i] = a[i];

    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
Heap and Stack Example

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```
Heap and Stack Example

```c
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
```
# include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
#include <stdlib.h>

int* copy(int a[], int size) {
    int i, *a2;
    a2 = malloc(size*sizeof(int));
    if (a2 == NULL)
        return NULL;
    for (i = 0; i < size; i++)
        a2[i] = a[i];
    return a2;
}

int main(int argc, char** argv) {
    int nums[4] = {1, 2, 3, 4};
    int* ncopy = copy(nums, 4);
    // .. do stuff with the array ..
    free(ncopy);
    return EXIT_SUCCESS;
}
Reallocating Memory

- Need to `#include <stdlib.h>`

- `void* realloc(void* p, size_t size)`
  - Creates a new allocation with given size, copies the contents of `p` into it, and frees `p`
  - Saves a few lines of code
  - Can sometimes be faster due to allocator optimizations
Aside: `errno` (must `#include <errno.h>`)  

- How do you know if an error has occurred in C?  
  - C does not have exceptions like Java
- Usually, return a special error value (e.g. NULL)  
- `stdlib` functions set a global variable called `errno`
- Can check error type by comparing to error code
  - `if (errno == ENOMEM)` // Allocation failure
- Or call `perror(message)` to print to stderr
  - `perror("malloc");` // "malloc: Cannot allocate memory"
Memory Allocation Example in C

```c
void foo(int n, int m) {
    int i, *p;
    p = (int*) malloc(n*sizeof(int)); /* allocate block of n ints */
    if (p == NULL) { /* check for allocation error */
        perror("malloc");
        exit(0);
    }
    for (i=0; i<n; i++) /* initialize int array */
        p[i] = i;
    /* add space for m ints to end of p block */
    p = (int*) realloc(p, (n+m)*sizeof(int));
    if (p == NULL) { /* check for allocation error */
        perror("realloc");
        exit(0);
    }
    for (i=n; i < n+m; i++) /* initialize new spaces */
        p[i] = i;
    for (i=0; i<n+m; i++) /* print new array */
        printf("%d\n", p[i]);
    free(p); /* free p */
}
```
Wouldn’t it be nice…

● If we never had to free memory?
● Do you free objects in Java?
  ○ Reminder: *implicit* allocator
● There are two main kinds of implicit allocators
  ○ Garbage collection
  ○ Reference counting
    ■ We'll get to this later in the course!
Garbage Collection (GC)

(Automatic Memory Management)

- **Garbage collection**: automatic reclamation of heap-allocated storage – application never explicitly frees memory

```c
void foo() {
    int* p = (int*) malloc(128);
    return; /* p block is now unreachable! */
}
```

- Used in many modern languages:
  - Lisp, Haskell, Java, C#, Ruby, Python, JavaScript, MATLAB, and more…

- Variants (“conservative” garbage collectors) exist for C and C++, but cannot collect all garbage
Garbage Collection

- How does the memory allocator know when memory can be freed?
  - In general, we cannot know what is going to be used in the future since it depends on conditionals
  - But, we can tell that certain blocks cannot be used if they are unreachable (via pointers in registers/stack/globals)

- Memory allocator needs to know what is a pointer and what is not – how can it do this?
  - Sometimes with help from the compiler
Memory as a Directed Graph

- Each allocated heap block is a node in the graph
- Each pointer is an edge in the graph

A node (block) is **reachable** if there is a path from any root to that node
Non-reachable nodes are **garbage** (cannot be needed by the application)
Garbage Collection

- Dynamic memory allocator can free blocks if there are no pointers to them
- How can it know what is a pointer and what is not?
- We’ll make some assumptions about pointers:
  - Memory allocator can distinguish pointers from non-pointers
  - All pointers point to the start of a block in the heap
  - Application cannot hide pointers (e.g. by coercing them to a long, and then back again)
Mark and Sweep

● Interrupt the program at a regular interval
  ○ "Stop the world" pauses

● Deallocate any unreachable nodes

● Resume program execution until the next pause and repeat