CSE 303
Lecture 11

Heap memory allocation (`malloc`, `free`)

reading: *Programming in C* Ch. 11, 17

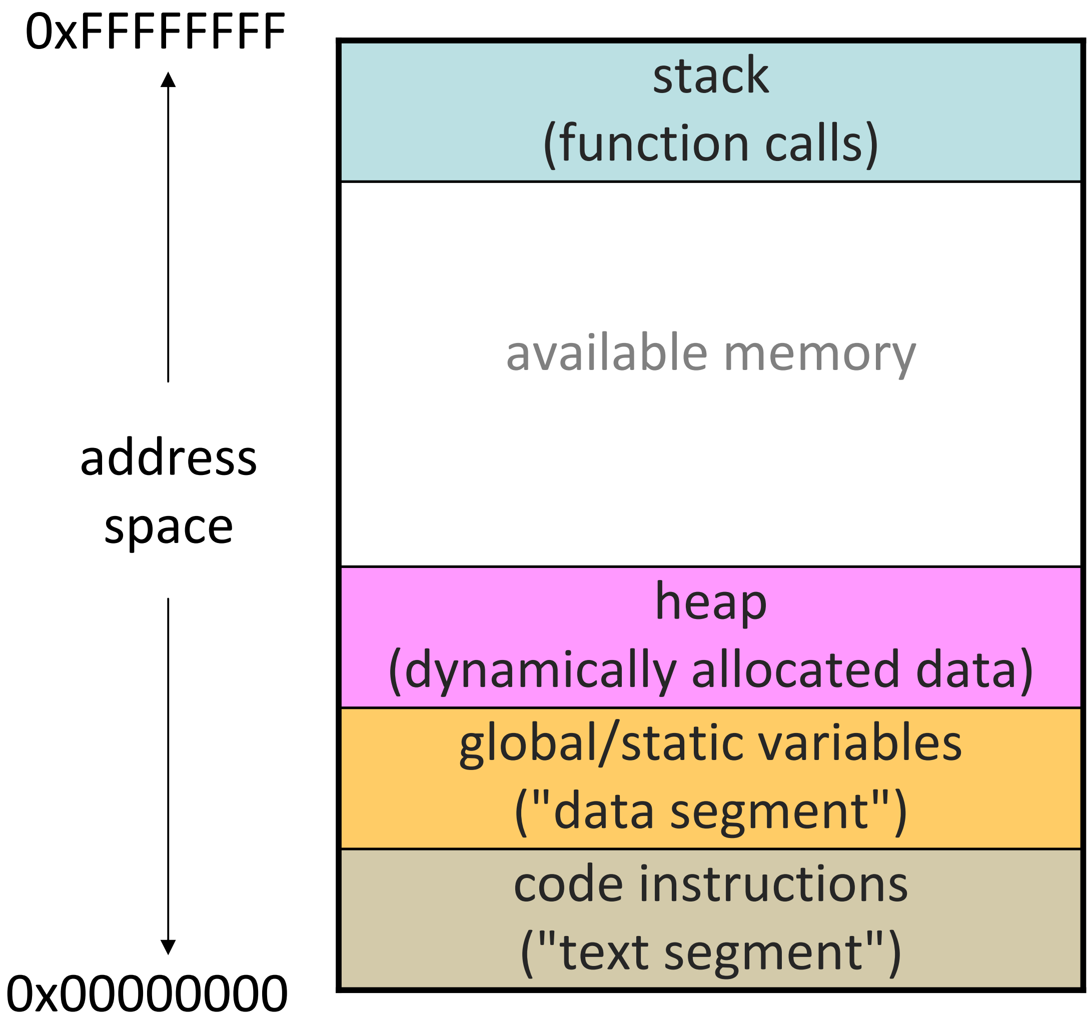
slides created by Marty Stepp
http://www.cs.washington.edu/303/

Lecture summary

- arrays as parameters and returns
  - arrays vs. pointers

- the heap
  - dynamic memory allocation (malloc, calloc, free)
  - memory leaks and corruption

Process memory layout



- as functions are called, data goes on a **stack**
- dynamic data is created on a **heap**

The sizeof operator

```
#include <stdio.h>

int main(void) {
    int x;
    int a[5];

    printf("int=%d, double=%d\n", sizeof(int), sizeof(double));
    printf("x    uses %d bytes\n", sizeof(x));
    printf("a    uses %d bytes\n", sizeof(a));
    printf("a[0] uses %d bytes\n", sizeof(a[0]));
    return 0;
}
```

Output:

```
int=4, double=8
x    uses 4 bytes
a    uses 20 bytes
a[0] uses 4 bytes
```

sizeof continued

- `sizeof(`***type***`)` or `(`***variable***`)` returns memory size in bytes
  - arrays passed as parameters do not remember their size

```
#include <stdio.h>

void f(int a[]);

int main(void) {
    int a[5];
    printf("a uses %d bytes\n", sizeof(a));
    f(a);
    return 0;
}

void f(int a[]) {
    printf("a uses %2d bytes in f\n", sizeof(a));
}
```

Output:

```
a uses 20 bytes
a uses  4 bytes in f
```

Arrays and pointers

- a pointer can point to an array element
  - an array's name can be used as a pointer to its first element
  - you can use [ ] notation to treat a pointer like an array
    - ***pointer*[*i*]** is ***i*** elements' worth of bytes forward from ***pointer***

```
int a[5] = {10, 20, 30, 40, 50};

int* p1 = &a[3];    // refers to a's fourth element
int* p2 = &a[0];    // refers to a's first element
int* p3 = a;        // refers to a's first element as well

*p1 = 100;
*p2 = 200;
p1[1] = 300;
p2[1] = 400;
p3[2] = 500;
```

Final array contents:

```
{200, 400, 500, 100, 300}
```

Arrays as parameters

- array parameters are really passed as pointers to the first element
  - The [ ] syntax on parameters is allowed only as a convenience

```
// actual code:
#include <stdio.h>

void f(int a[]);

int main(void) {
    int a[5];
    ...
    f(a);
    return 0;
}

void f(int a[]) {
    ...
}
```

```
// equivalent to:
#include <stdio.h>

void f(int* a);

int main(void) {
    int a[5];
    ...
    f(&a[0]);
    return 0;
}

void f(int* a) {
    ...
}
```

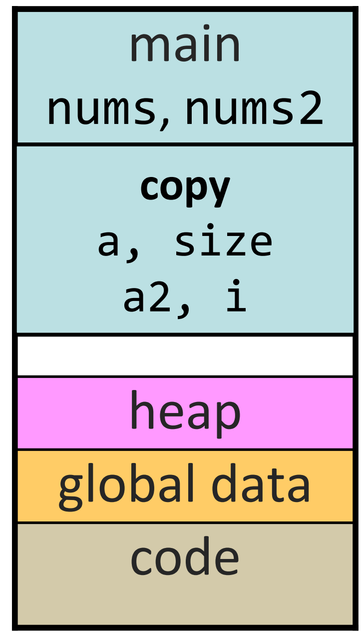
Returning an array

- stack-allocated variables disappear at the end of the function
  - this means an array cannot be safely returned from a method

```
int[] copy(int a[], int size);

int main(void) {
    int nums[4] = {7, 4, 3, 5};
    int nums2[4] = copy(nums, 4);   // no
    return 0;
}

int[] copy(int a[], int size) {
    int i;
    int a2[size];
    for (i = 0; i < size; i++) {
        a2[i] = a[i];
    }
    return a2;   // no
}
```



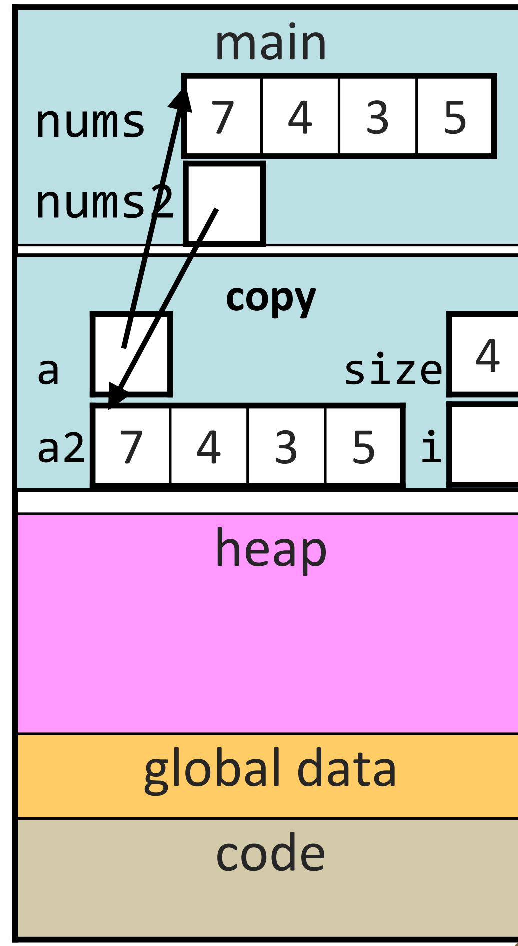
Pointers don't help

- **dangling pointer**: One that points to an invalid memory location.

```c
int* copy(int a[], int size);

int main(void) {
    int nums[4] = {7, 4, 3, 5};
    int* nums2 = copy(nums, 4);
    // nums2 dangling here
    ...
}

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



Our conundrum

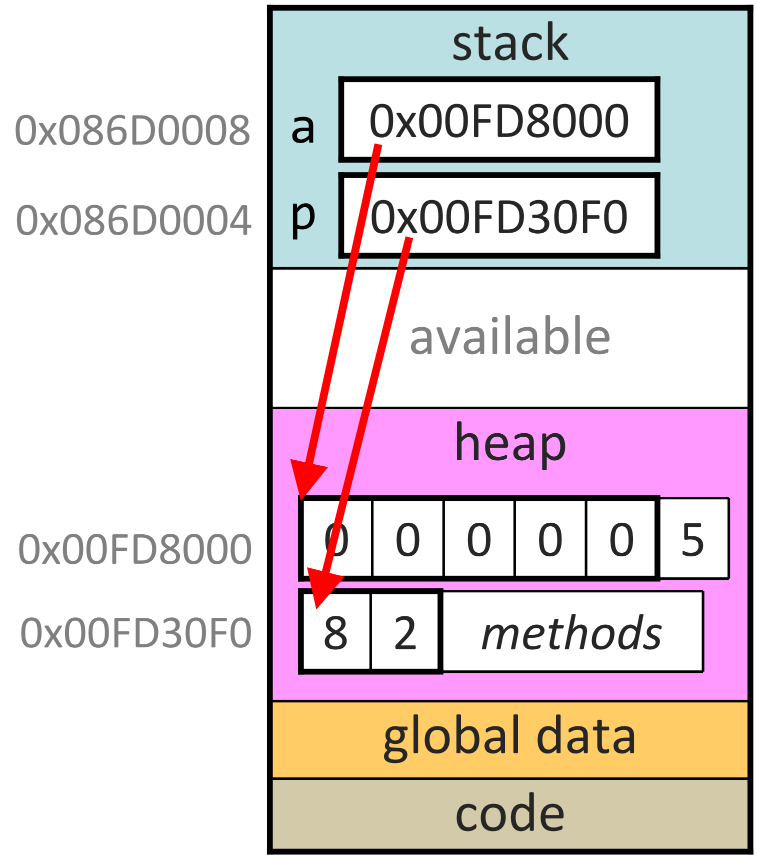
- We'd like to have data in our C programs that is:
  - dynamic (size of array changes based on user input, etc.)
  - long-lived (doesn't disappear after the function is over)
  - bigger (the stack can't hold all that much data)

- Currently, our solutions include:
  - declaring variables in main and passing as "output parameters"
  - declaring global variables (do not want)

The heap

- **heap** (or "free store"): large pool of unused memory that you can use for dynamically allocating data and objects
  - for dynamic, long-lived, large data
  - many languages (e.g. Java) place <u>all</u> arrays/ objects on the heap

```
// Java
int[] a = new int[5];
Point p = new Point(8, 2);
```



`malloc`

variable `= (`***type***`*) malloc(`***size***`);`

- `malloc` function allocates a heap memory block of a given size
  - returns a pointer to the first byte of that memory
  - you should cast the returned pointer to the appropriate type
  - initially the memory contains garbage data
  - often used with `sizeof` to allocate memory for a given data type

```
// int a[8];   <-- stack equivalent

int* a = (int*) malloc(8 * sizeof(int));
a[0] = 10;
a[1] = 20;
...
```

calloc

```
variable = (type*) calloc(count, size);
```

- `calloc` function is like `malloc`, but it zeros out the memory
  - also takes two parameters, number of elements and size of each
  - preferred over `malloc` for avoiding bugs (but slightly slower)

```
// int a[8] = {0};   <-- stack equivalent
int* a = (int*) calloc(8, sizeof(int));
```

- `malloc` and `calloc` are found in library `stdlib.h`

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

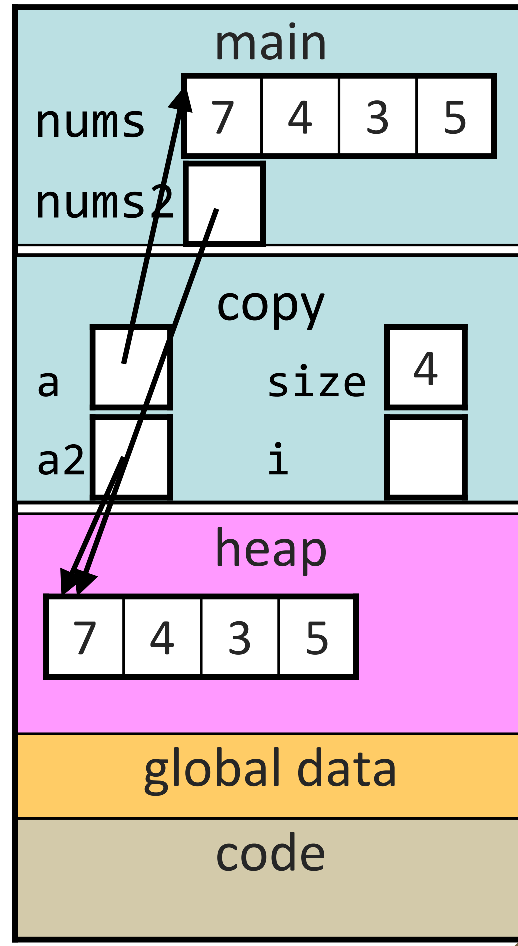
Returning a heap array

- when you want to return an array, `malloc` it and return a pointer
  - array will live on after the function returns

```
int* copy(int a[], int size);

int main(void) {
    int nums[4] = {7, 4, 3, 5};
    int* nums2 = copy(nums, 4);
    ...
    return 0;
}

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



NULL

- **NULL**: An invalid memory location that cannot be accessed.
  - in C, `NULL` is a global constant whose value is 0
  - if you `malloc/calloc` but have no memory free, it returns `NULL`
  - you can initialize a pointer to `NULL` if it has no meaningful value
  - dereferencing a null pointer will crash your program

```
int* p = NULL;
*p = 42;            // segfault
```

- *Exercise* : Write a program that figures out how large the stack and heap are for a default C program.

Deallocating memory

- heap memory stays claimed until the end of your program
- **garbage collector**: A process that automatically reclaims memory that is no longer in use.
  - keeps track of which variables point to which memory, etc.
  - used in Java and many other modern languages; not in C

```
// Java
public static int[] f() {
    int[] a = new int[1000];
    int[] a2 = new int[1000];
    return a2;
}   // no variables refer to a here; can be freed
```

Memory leaks

- **memory leak**: Failure to release memory when no longer needed.
  - easy to do in C
  - can be a problem if your program will run for a long time
    - when your program exits, all of its memory is returned to the OS

```
void f(void) {
    int* a = (int*) calloc(1000, sizeof(int));
    ...
}   // oops; the memory for a is now lost
```

free

```
free(pointer);
```

- releases the memory pointed to by the given pointer
  - *precondition*: pointer must refer to a heap-allocated memory block that has not already been freed

```
int* a = (int*) calloc(8, sizeof(int));
...
free(a);
```

  - it is considered good practice to set a pointer to `NULL` after freeing

```
free(a);
a = NULL;
```

Memory corruption

- if the pointer passed to `free` doesn't point to a heap-allocated block, or if that block has already been freed, bad things happen

```
int* a1 = (int*) calloc(1000, sizeof(int));
int a2[1000];
int* a3;
int* a4 = NULL;

free(a1);      // ok
free(a1);      // bad (already freed)
free(a2);      // bad (not heap allocated)
free(a3);      // bad (not heap allocated)
free(a4);      // bad (not heap allocated)
```

- you're *lucky* if it crashes, rather than silently corrupting something