Memory Allocation

CSE 220: Systems Programming

Ethan Blanton & Carl Alphonce

Department of Computer Science and Engineering

University at Buffalo

Allocating Memory

We have seen how to use pointers to address:

- An existing variable
- An array element from a string or array constant

This lecture will discuss requesting memory from the system.

Memory Lifetime

All data we have seen so far have well-defined lifetime:

- Persisting for the entire life of the program
- Persisting for the duration of a single function call

Sometimes we need programmer-controlled lifetime.

For example:

- Data created in one function, and destroyed in another
- Data created during program execution, but lasts forever

Examples

```
int global;             /* Lifetime of program */

void foo() {
    int x;              /* Lifetime of foo() */
}
```

Here, `global` is statically allocated:

- It is allocated by the compiler
- It is created when the program starts
- It disappears when the program exits

Examples

```
int global;           /* Lifetime of program */

void foo() {
    int x;            /* Lifetime of foo() */
}
```

Whereas x is automatically allocated:

- It is allocated by the compiler
- It is created when `foo()` is called¶
- It disappears when `foo()` returns

Effective Questions

For programming questions, ask:

- What did I do?
- What did I expect to happen?
- What actually happened?
- How are they different?

You must know what you expected to identify the problem!

When asking us questions, tell us what you did, what you expected, and what you got.

The Heap

The heap represents memory that is:

- allocated and released at run time
- managed explicitly by the programmer
- only obtainable by address

Heap memory is just a range of bytes to C.

Memory from the heap is given a type by the programmer.

We will see much more about the heap later!

Heap Allocations

Each allocation from the heap is accessed via a pointer.

Each allocation has a fixed size.

This size is determined at allocation time.

Accesses outside of the allocation must not be made using the allocated pointer!

Releasing Memory

Memory can be released back to the heap.

This memory can then be used for future heap allocations.

It can potentially (but often is not) be returned to the OS.

Memory that has been released must not be accessed again.

The C language will not detect accesses to released memory!

`void *`

The type `void *` is used to indicate a pointer of unknown type.

You may recall that `void` indicates a meaningless return value.

`void *` is treated specially by the C compiler and runtime:

- A `void *` variable can store any pointer type
- Type checks are mostly bypassed assigning to/from `void *`
- Any attempt to dereference a `void *` pointer is an error

Pointer Assignments

Consider the following:

```
int i;
double d;
int *pi = &i;
double *pd = &d;
```

Each of these pointers is typed. These are errors:

```
pi = pd;
pd = pi;
```

Pointer Assignments

Consider the following:

```
int i;
double d;
int *pi = &i;
double *pd = &d;
```

This is where it gets dangerous:

```
void *p = pi;
pd = p;
```

This is perfectly legal.

(What does it mean?)

Aside: The sizeof operator

There are several operators used to help with reflection in C.

One of these is the `sizeof` operator.

It returns the size in bytes of its operand, which can be:

- A variable
- An expression that is “like” a variable
- A type

(Expressions “like” a variable include, *e.g.*, members of structures.)

Looking at sizeof

Examples:

```
void func(int matrix[2][3]) {
    double dist;

    sizeof(int);                  // yields 4
    sizeof(dist);                 // yields 8
    sizeof(matrix);               // yields ... 8?
}
```

Note that sizeof arrays is not reliable.

Only arrays declared within the current scope will be correct.¶

We will discuss the sizes of things in more detail, later.

The Standard Allocator

The C library contains a standard allocator.

```
#include <stdlib.h>

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

These functions allow you to:

- Request memory (`malloc()`, `calloc()`, `realloc()`)
- Release memory (`free()`)

Allocating

The allocating functions request memory in slightly different ways.

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

All three return a non-null `void` pointer on success.

All three return `NULL` on failure.

malloc()

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

Malloc returns a `void *` pointer, which can point to anything.

It allocates at least `size` bytes.

`size` is often the result of a `sizeof()` expression.

To allocate an integer:

- Determine the size of an `int`
- Request enough memory to hold one

```
int *pi = malloc(sizeof(int));
```

Allocating an array

To allocate an array with 10 `int` entries dynamically, we:

- Determine the size of a single `int`
- Tell the system we want ten of those
- Assign the result to an appropriate pointer

```
int *array = malloc(10 * sizeof(int));
```

The variable `array` can now be used as a regular `int` array.

calloc()

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

The closely-related `calloc()` allocates cleared memory.

The memory returned by `malloc()` is uninitialized.

The memory returned by `calloc()` is set to bitwise zero.

Note that invocation is slightly different!

realloc()

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

Allocation sizes are fixed, but you can request a resize.

`realloc()` will attempt to change the size of an allocation.

If it cannot, it may create a new allocation of the requested size.

Normal usage is:

```
ptr = realloc(ptr, newsize);
```

This handles the case where the resize is not possible.

free()

```
void free(void *ptr);
```

Free accepts a `void *` pointer, which can point to anything.

Freed memory returns to the system to be allocated again later via `malloc()`.

```
free(array);
```

Note that free does not modify the value of its argument.
Thus you cannot “tell” that a particular location has been freed!

Failed allocations

Allocations can fail.

A failed allocation will return `NULL`.

On a modern machine, this *usually* means an unreasonable allocation.

E.g., you accidentally allocated 2 GB instead of 2 KB.

On smaller systems, failed allocations are normal.

Often you can't do much about a failed allocation, of course.

Use-after-free

A common class of error is use-after-free.

This is when a freed pointer is used.

This is particularly dangerous, because the allocator may reuse that pointer.

Therefore, it is:

- Pointing to usable memory
- Not valid
- Likely to corrupt data!

Setting free’d pointers to `NULL` can help prevent this.

Out-of-bounds access

Because heap allocations have no obvious size, out-of-bounds access is easy.

```c
int *array = malloc(2 * sizeof(int)); /* int[2]   */
for (int i = 0; i <= 2; i++) {        /* 0, 1, 2! */
    array[i] = 0;                     /* Illegal  */
}
```

The compiler will not catch this.

Practice Example 1

```
1  int a[2];
2  int *b = malloc(2 * sizeof(int));
3  int *c;
4
5  a[2] = 5;
6  b[0] += 2;
7  c = b + 3;
8  free(&(a[0]));
9  free(b);
10 free(b);
11 b[0] = 5;
```

Where are the errors?

1. Line 5
2. Line 6
3. Line 8
4. Line 10
5. Line 11

Practice Example 2

```
1  int a[2];
2  int *b = malloc(2 * sizeof(int));
3  int *c;
4
5  a[2] = 5;
6  b[0] += 2;
7  c = b + 3;
8  free(&(a[0]));
9  free(b);
10 free(b);
11 b[0] = 5;
```

Where is the first guaranteed error?

1. Line 5
2. Line 6
3. Line 8
4. Line 10
5. Line 11

Summary

- The heap is where you manually allocate memory.
- The C standard library contains a flexible allocator.
- Heap allocations are sized by the programmer.

Next Time …

- Integer properties
- Bit widths
- Integer representation

References I

Required Readings

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Second Edition. Chapter 2: 2.7. Prentice Hall, 1988.

License