CSE 220: Systems Programming

Memory Allocation

Ethan Blanton

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.

The Heap

We will see more about the heap later, but it 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.

Heap Allocations

Each allocation from the heap is represented by a pointer.

Each allocation has a fixed size.

This size is declared at allocation time.

Accesses outside of the allocation must not be made using the returned 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 compiler and runtime 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:

- It can contain 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?)

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(size_t nmemb, 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 pointer 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.

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 …

- Aggregate data types (for real this time)

References I

License