Chapter 14 Interlude: Memory API

Q: How to allocate and manage memory ?

- What interfaces are commonly used?
- What mistakes should be avoided?

14.1 Types of Memory

Stack memory:

- Allocations and deallocations managed ***implicitly*** by the compiler
  - For you the programmer
- Sometimes this is referred to as automatic memory

Example. Declaring an integer *x* on the stack:

```
int func() {
        var x int // in Go
        int x; // declare integer on the stack
        …
        return 0;
}
```

Variables on the stack are lost when func() returns.

- Not suitable for long-lived memory (data); image

Heap memory:

For long-lived memory:

- Allocations and deallocation ***explicitly*** handled by you the programmer.
  - A heavy responsibility!
  - Cause of many many bugs!

Example. Declaring and allocating an integer on the heap:

```
void func() {
        int *x = (int *) malloc(sizeof(int));
        …
}
```

Note:

- Compiler: actually makes space on the stack for (int *)
- malloc() requests space for integer on the heap
  - Returns address of integer (if success, NULL otherwise)
  - The returned address is stored on the stack

14.2 The malloc() Call

malloc(): pass it a size (asking for some room on the heap)

- Success: gives back pointer to newly allocated space
- Fail: returns NULL

To use malloc() we should *include <stdlib.h>* to let the compiler know
— so that it can check that we are calling malloc() correctly.

malloc(): takes a size as input

-

-

Note: sizeof() of a variable, doesn't return the size of the memory being pointed to:

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

printf("%d\n", sizeof(x));
```

First line does what you expect: allocate space for 10 integers.

However, sizeof(x): only returns the size of an integer,
which is 4 or 8 (for 32-bit / 64-bit architectures)

Another note: malloc() returns a pointer to type void.

This is C’s way to tell the programmer that she needs to decide
what type should be used… and must cast it to the expected type…

```
double *d = (double *) malloc(sizeof(double));
```

Casting is only a help for the programmer (and the compiler) to keep the code correct…

Casting doesn’t mean anything for the CPU instructions…

14.3 The free() Call

Freeing memory is easy!

Just call free(x), where x is the pointer to the memory allocated by malloc().

It is also easy to forget to call free().

14.4 Common Errors

Newer languages: Support automatic memory management

- Allocate memory with new()
- Free memory with garbage collector: Go and Java and C#
- Newer versions of C++ and Rust
  - Support smart pointers
  - Automatically frees memory when pointer goes out of scope.

```
fn f() {
        let x: ~int = ~1024;          // allocate space and initialize an int on the heap
        println(fmt!("%d", *x));      // print it on the screen
} // <— the memory that x pointed at is automatically freed here…
```

Common Error #1: Forgetting to Allocate Memory

Many functions expect memory to be allocated before you call them.

Ex. strcpy(dst, src) — copy string from source ptr to a destination ptr.

```
#include <string.h>

char *src = “hello”;
char *dst; // unallocated
strcpy(dst, src);
```

Fix:

```
#include <stdlib.h>

char *src = “hello”;
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src);
```

Better option: strdup()

Common Error #2: Not Allocating Enough Memory

```
char *src = “hello”;
char *dst = (char *) malloc(strlen(src)); // too small
strcpy(dst, src);
printf(“Hey: %s\n”, dst);
```

Buffer overflow!
Compiles and runs seemingly without problems…

Common Error #3: Forgetting to Initialize Allocated Memory

```
char *src = “hello”;
char *dst; // forget to initialize
printf(“Hey: %s\n”, dst);
```

Without calling strcpy()
If lucky: program works with zero values
Otherwise: some random/harmful thing may happen

Common Error #4: Forgetting to Free Memory

- Memory leak: occur when you forget to free memory
  - Problem in long-running applications/systems including the OS itself
  - Restart required
- Also a problem in GC-ed languages
  - If you still have a reference/pointer to an object (chunk of memory)
  - The GC won’t free it!!

OS will clean up allocated memory when a process exists (short-lived)…

- Good habit to free memory even for short-lived programs
- What if your code gets converted into a library or long-lived program… (web server)

Common Error #5: Freeing Memory Before You are done with it!

Using the *dst pointer after calling free(dst);
Dangling pointer. Bad things can happen! But not always… which is also a problem… because you won’t discover the problem (yet!)

Common Error #6: Freeing Memory Repeatedly

Double free: undefined behavior

The memory allocation library will typically throw a runtime error.

Common Error #7: Calling free() incorrectly

Pass src to free(src). Also undefined. But bad things could happen.

The memory allocation library will typically throw a runtime error.