/unc/comp211

Systems Fundamentals

Design Principles of C Functions with Pointer Parameters !

© 2020 Kris Jordan All Rights Reserved

Best Practice: const struct Pointer Params

```
void Point_print(const Point *self)
{
    self->x = self->y; // YIKES
    printf("(%f, %f)\n", (*self).x, (*self).y);
}
```

```
point.c: In function ‘Point_print’:
point.c:20:13: error: assignment of member ‘x’ in read-only object
     self->x = self->y; // YIKES
             ^
```

- Declaring a pointer value const signals you do not intend to mutate it
- If your implementation of the function breaks this contract the compiler will typically catch it (if you try hard enough in C, you can get around the const)
- Rule of thumb: always declare pointer parameters as const
  - If it turns out you need to mutate it, then you should think very critically about the design of your function

Consider the following function signatures...

```
Point add_optA(const Point *a, const Point *b);

void add_optB(const Point *a, Point *b);

void add_optC(Point *a, const Point *b);

void add_optD(const Point *a, const Point *b, Point *out);
```

- Each of these could describe a function that adds two Point structs together.
- What can you infer about how each function works by just reading its signature?

Structs as Parameters

- Just like primitive data types, functions can have struct parameters

- A struct argument's members are *copied* to parameter member's
  - In other words, `structs` are *pass-by-value*
  - Complete copies are made into the function call frame just like primitives
    - This is not possible with objects in memory managed languages like Java/JavaScript/etc!

- Using *struct pointers* as parameters is often preferred to structs
  - Usually more efficient to copy a pointer than every member of a large struct
  - *However*, must be careful and intentional in the use of the pointers.
    - Rule of thumb: declare all struct pointer parameters as const unless you intend to mutate it

Be *intentional* when you *want* mutable params.

- Two common patterns:

1. OOP-inspired first parameter points to the subject being mutated
   - Note: `self` is not a keyword in C, the parameter could have been named anything (i.e. this or p)

```
void Point_translate(Point *self, double x, double y)
{
    self->x += x;
    self->y += y;
}
```

2. Have an explicit "out" parameter
   - This signals to the caller an intent to write a result to the address passed to the parameter.

```
void Point_translate_to(const Point *self, double x, double y, Point *out)
{
    out->x = self->x + x;
    out->y = self->y + y;
}
```

Ownership: Who is responsible for freeing?

- Each time Path list was extended, more heap memory was allocated
  - Is freeing *just* the head Path enough?
  - Let's use valgrind to find out.

- The documentation of extend notes:

  ```
  Extend a Path by creating a new Path Node at its tail.
  Returns a pointer to the next Path for future extensions.
  The returned Pointer is considered owned by the head Path
  in the list and MUST NOT be freed manually.
  ```

- The current implementation of Path_free introduces a memory leak. The function should free *all subsequent Path* values pointed to by next *before* freeing itself. Let's implement that recursively.

Data Structures and Ownership

- In an unmanged heap memory environment (like C, C++, Obj-C, Rust) you must think deeply about the **ownership** of values on the heap

- The **owner** of allocated heap memory is responsible for freeing it.

- For every allocation, you should be able to discern its owner
  - The ownership in a linked list is recursive.
  - The head variable in main owned a Path, that owned a Path, and so on.
  - Freeing the head variable (using Path_free!) freed all its owned Paths

- You would *not* consider tail to have ownership, just a reference
  - You should only free the owner and only once!
  - Freeing a reference will lead to double free.
  - You also want to be careful never to use a reference beyond the lifetime of its referent.