On Hash maps and their shortest implementation possible

About Hash maps

Explaining hash maps, their implementation and showing a very short but functioning implementation in c.
Hash maps are the backbone of fast running programs. Hash maps power caches, make searching really fast (for certain workloads faster than search trees), they allow databases to create indexes for really fast lookups and they are used to create sets.

Hashes and Hashing functions

A hash function always computes the same integer for the same input, called a hash. This integer is then used to index into the underlying array of the map.
Lets take a look at some common hashing applications: Java hashes strings by summing the characters of the string, while each is xored with the length. [1].

```
var s = "Hello World";
for (int i = 0, h = 0; i < s.length(); i++)
    h += s.codePointAt(i)*31
        ^ (s.length()-i);
```

We will use a similar, but different algorithm for hashing our key strings: fnv-1a [2]. The key of fnva-1a is to start with a default value for the hash, called the base, modify it by xoring it with the current character and then multiplying it with a prime number. On that basis we can create the first function of our naive implementation, `hash()` to hash our string keys:

```
const size_t BASE = 0x811c9dc5;
const size_t PRIME = 0x01000193;
size_t hash(Map *m, char *str) {
    size_t initial = BASE;
    while(*str) initial ^= *str++ * PRIME;
    return initial & (m->cap - 1);
}
```

The first things to notice, is the two constants required by fnva-1a, the parameter of the hash function of the `Map` type and the bitwise and in the return statement. The `m` parameter is used specifically in combination with the bitwise `&` to restrict the resulting hash to the size of the underlying array, thus eliminating out of bounds errors - this way of computing modulo is faster than `initial % (m->cap-1)`, but only works for the cap being a power of too. We control the size of the map, thus we can keep this in mind.

Map Initialisation

The Map structure contains the capacity, the size and the array of buckets, each bucket containing `void *`.

[1] String.hashCode()
[2] Fnv Hashing Wikipedia

This type can also just be a value, such as a double. C however allows for erasing the type of a pointer by casting it: `(void *)p`. Therefore this map can contain any pointer and does not assume ownership over the value itself - the downside is, the user has to cast the inserted and extracted pointers, while keeping track of their lifetimes.

```
typedef struct Map {
    size_t size;
    size_t cap;
    void **buckets;
} Map;
```

The `Map` is initialised with a size of 0, the defined cap and by allocating the buckets. We check for allocation failures with the assertion.

```
Map init(size_t cap) {
  Map m = {0, cap};
  m.buckets = malloc(sizeof(void *) * m.cap);
  assert(m.buckets != NULL);
  return m;
}
```

Pointer Insertion

Inserting a pointer into the map consists of incrementing the size field, computing the hash and assigning the element at the index to the pointer we want to insert:

```
void put(Map *m, char *str, void *value) {
  m->size++;
  m->buckets[hash(m, str)] = value;
}
```

Pointer Extraction

Extracting a pointer works the same way as the insertion: computing the hash and returning the value at the index:

```
void *get(Map *m, char *str) {
    return m->buckets[hash(m, str)];
}
```

Usage Example

The callee of the map functions can even insert pointers to stack variables, even if they do not outlive the scope. They also have to free the allocated bucket array.

```
int main(void) {
  Map m = init(1024);
  double d1 = 25.0;
  put(&m, "key", (void *)&d1);
  printf("key=%f\n", *(double *)get(&m, "key"));
  free(m.buckets);
  return EXIT_SUCCESS;
}
```