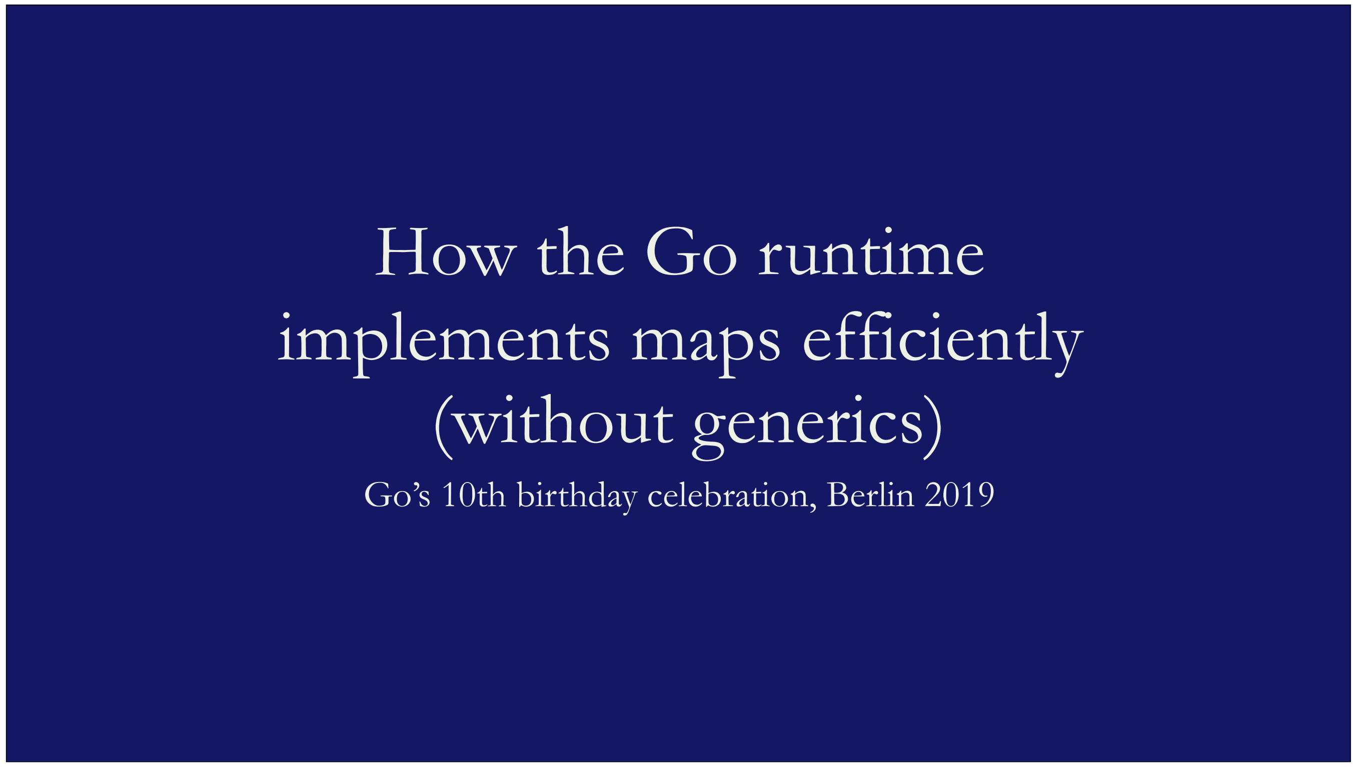


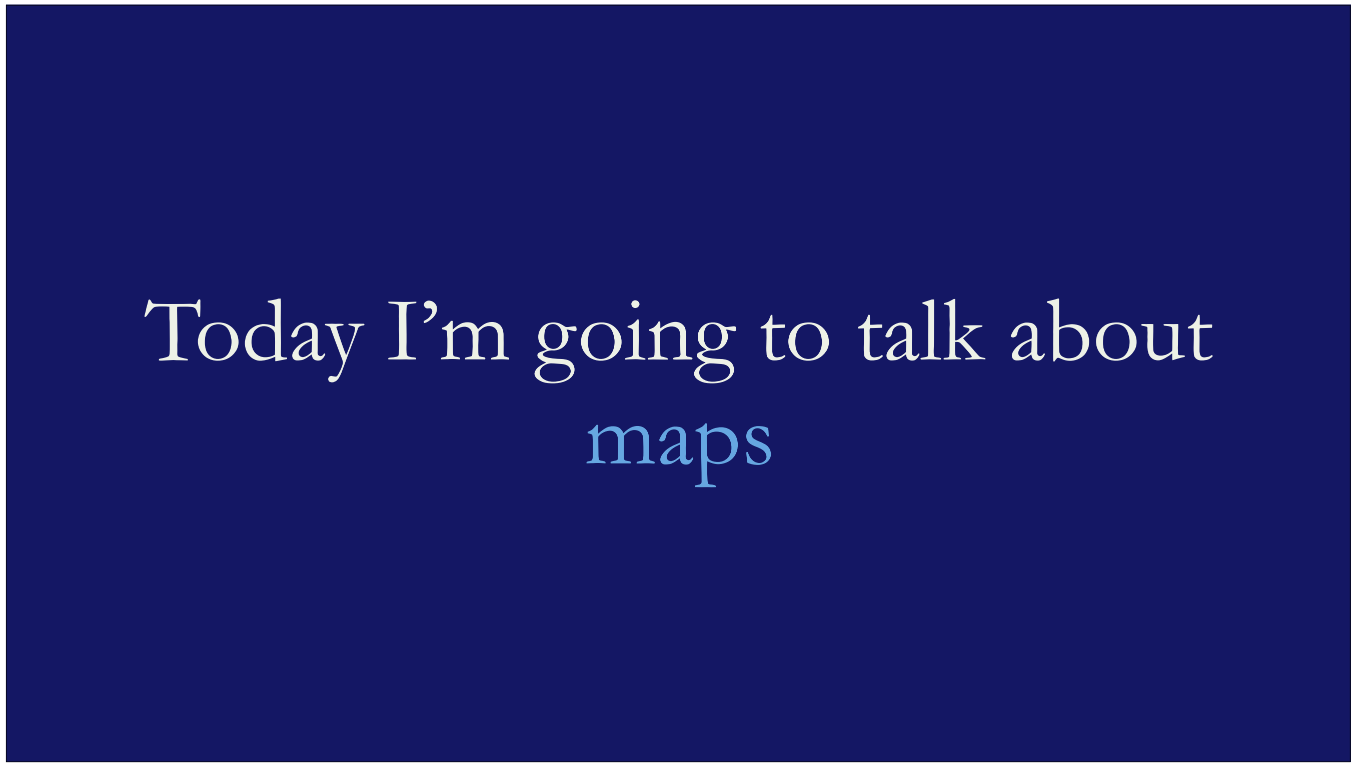
How the Go runtime implements maps efficiently (without generics)

Go's 10th birthday celebration, Berlin 2019

Hello, and thank you for coming along tonight.

Before we get started I want to give a shout out to Natalie. We first met nearly five years ago, right here in Berlin. It's been amazing to watch the Berlin Go community grow as the direct result of Natalie's efforts. Not just this meetup, but women who go, women tech makers, and the GDG devfest she organised this weekend.

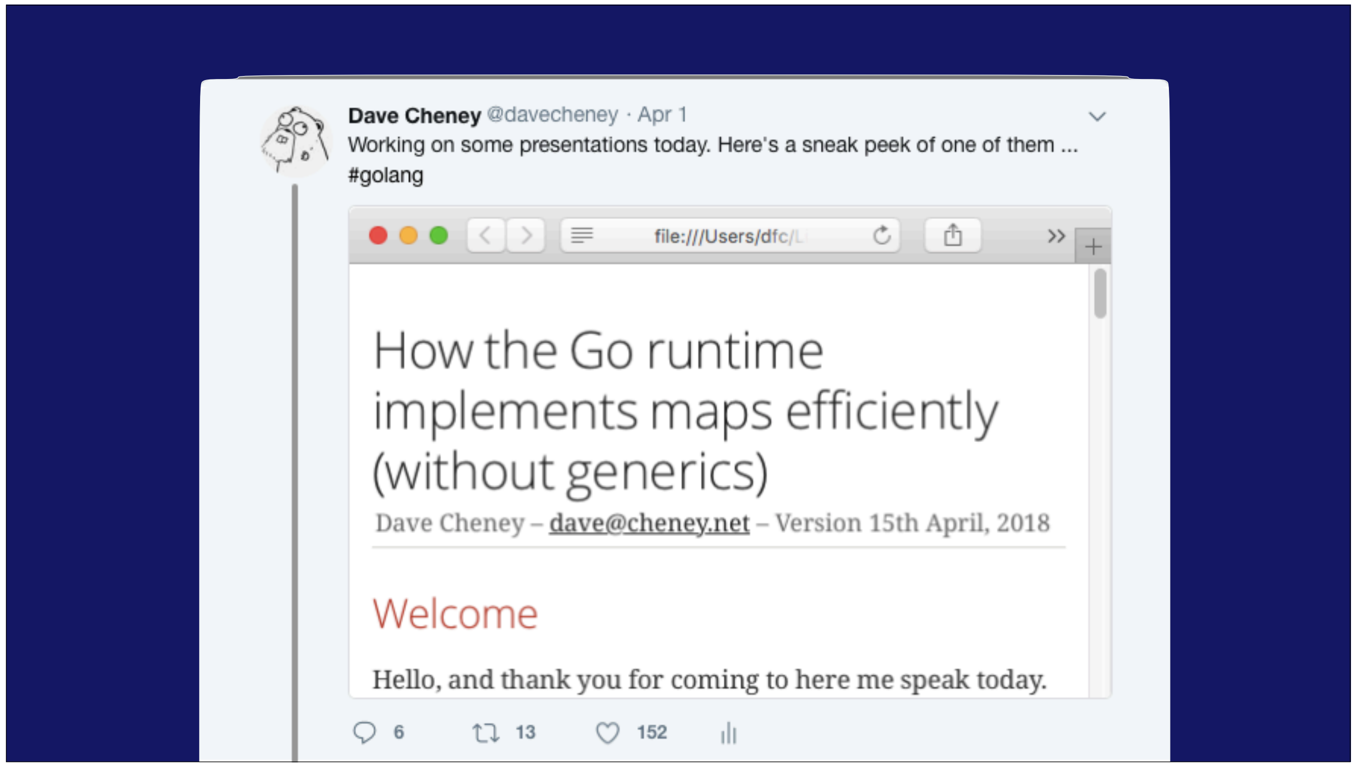
We all owe Natalie a debt of thanks for her tireless efforts to create a vibrant, caring, inclusive Go scene here in Berlin.



Today I'm going to talk about maps

Because Go is turning 10 today, I wanted to take this opportunity to talk about one of my favourite implementation details of the language.

While I think everyone in this room knows that maps, slices, and channels are sort of special, because they are implemented by the runtime, not our code, perhaps not everyone in this room knows _how_ they are implemented.



Dave Cheney @davecheney · Apr 1
Working on some presentations today. Here's a sneak peek of one of them ...
#golang

file:///Users/dfc/

How the Go runtime implements maps efficiently (without generics)

Dave Cheney – dave@cheney.net – Version 15th April, 2018

Welcome

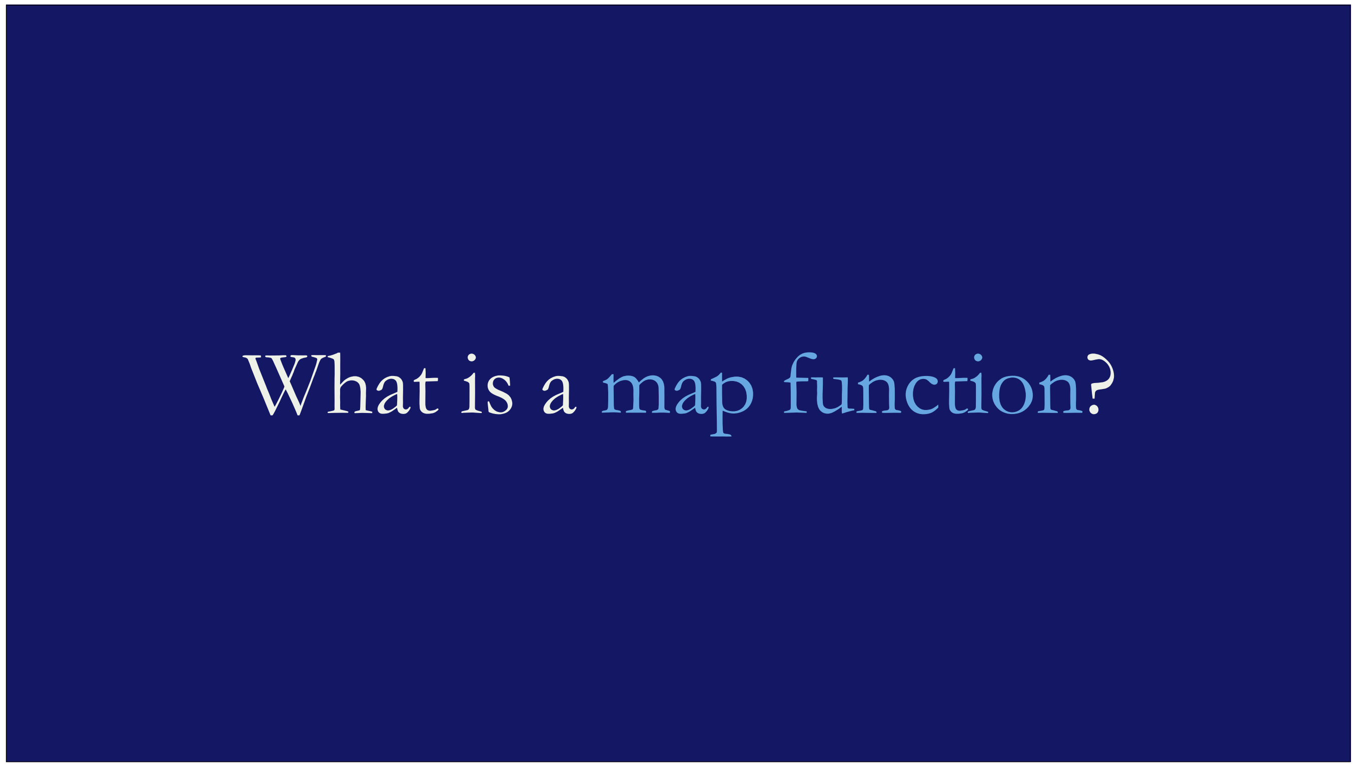
Hello, and thank you for coming to here me speak today.

6 13 152

Now, to break the fourth wall, way back when I was working on this talk I tweeted out a teaser. And this was the response.

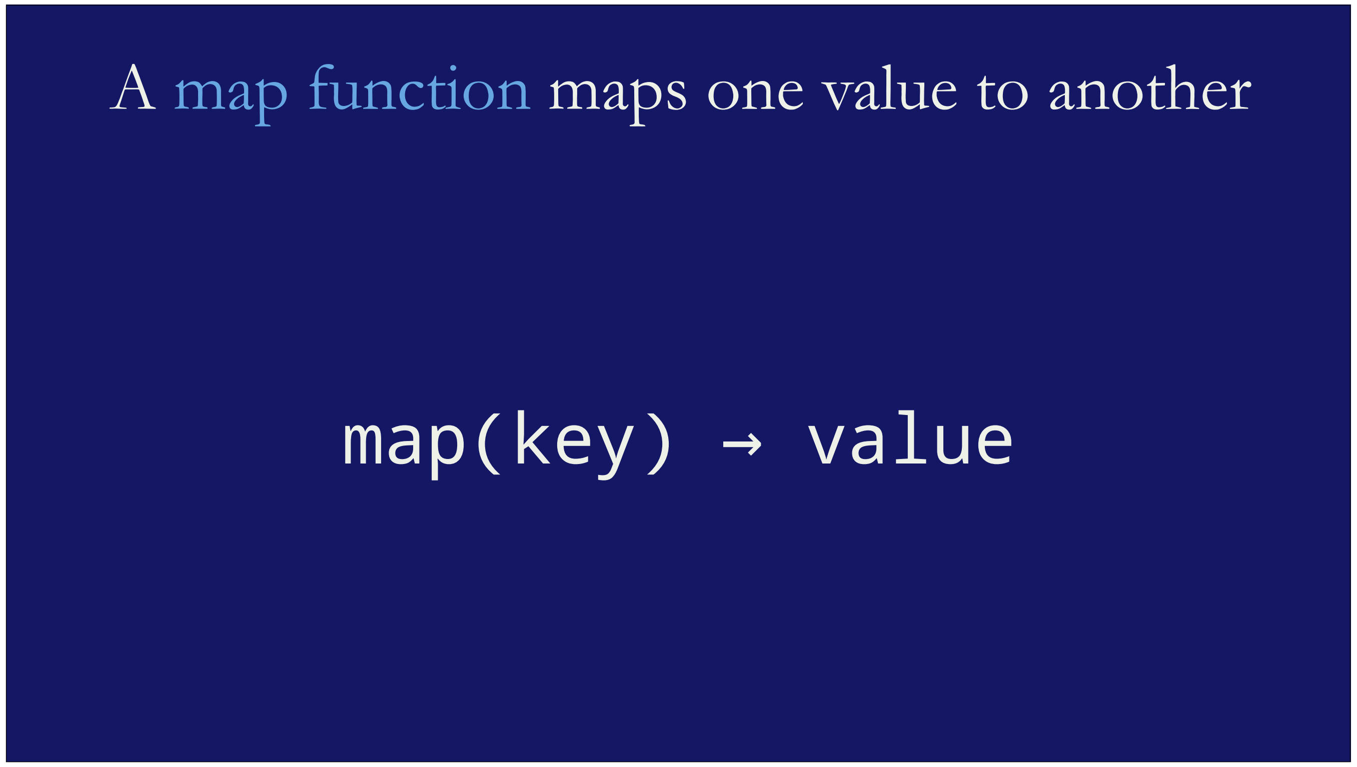
(click)

And too a certain extent brad is right, but there isn't as much unsafe as you might think, and as you'll see the parts of the map implementation that matter are not unsafe at all.



What is a map function?

To understand what a map type does, let's talk about the idea of the _map function_.



A map function maps one value to another

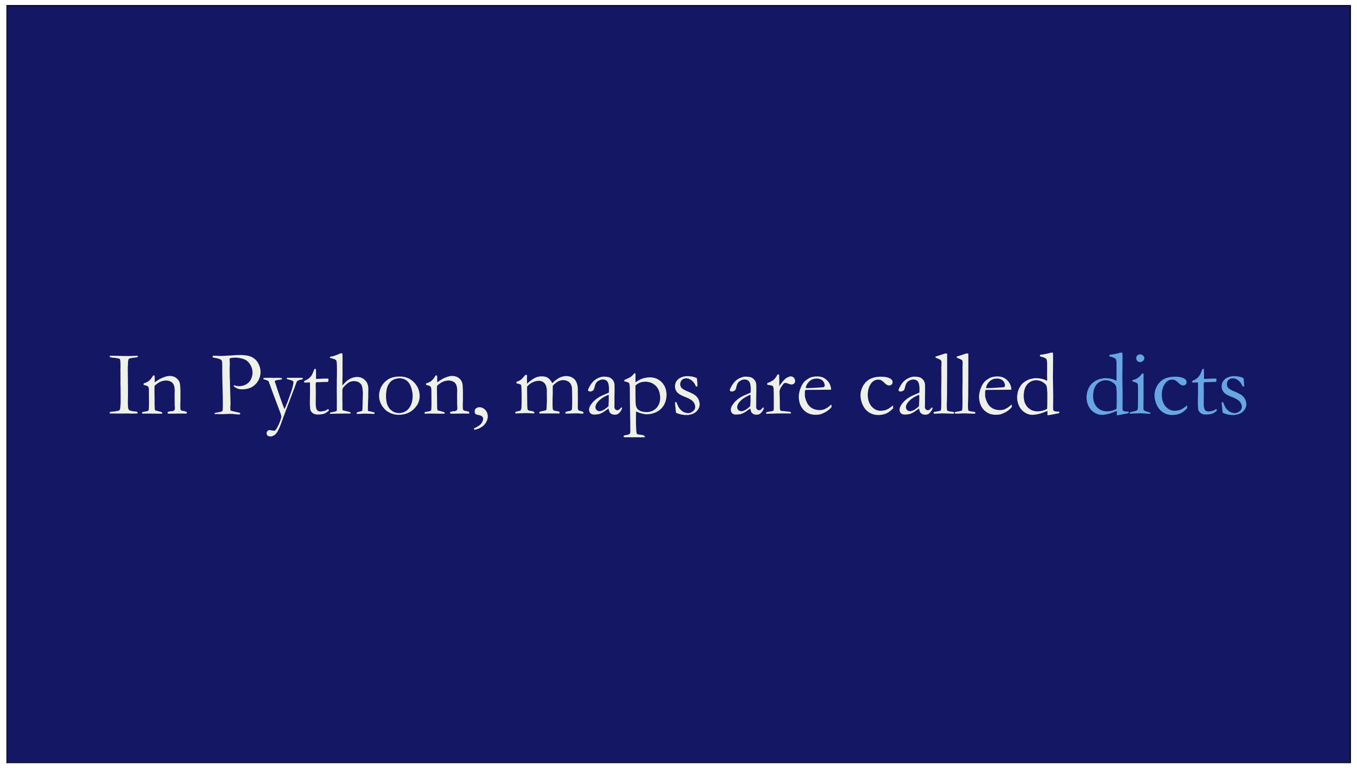
```
map(key) → value
```

Map is a function that given one value, let's call it a key, it will return a second.

reddis / memcache / kv stores

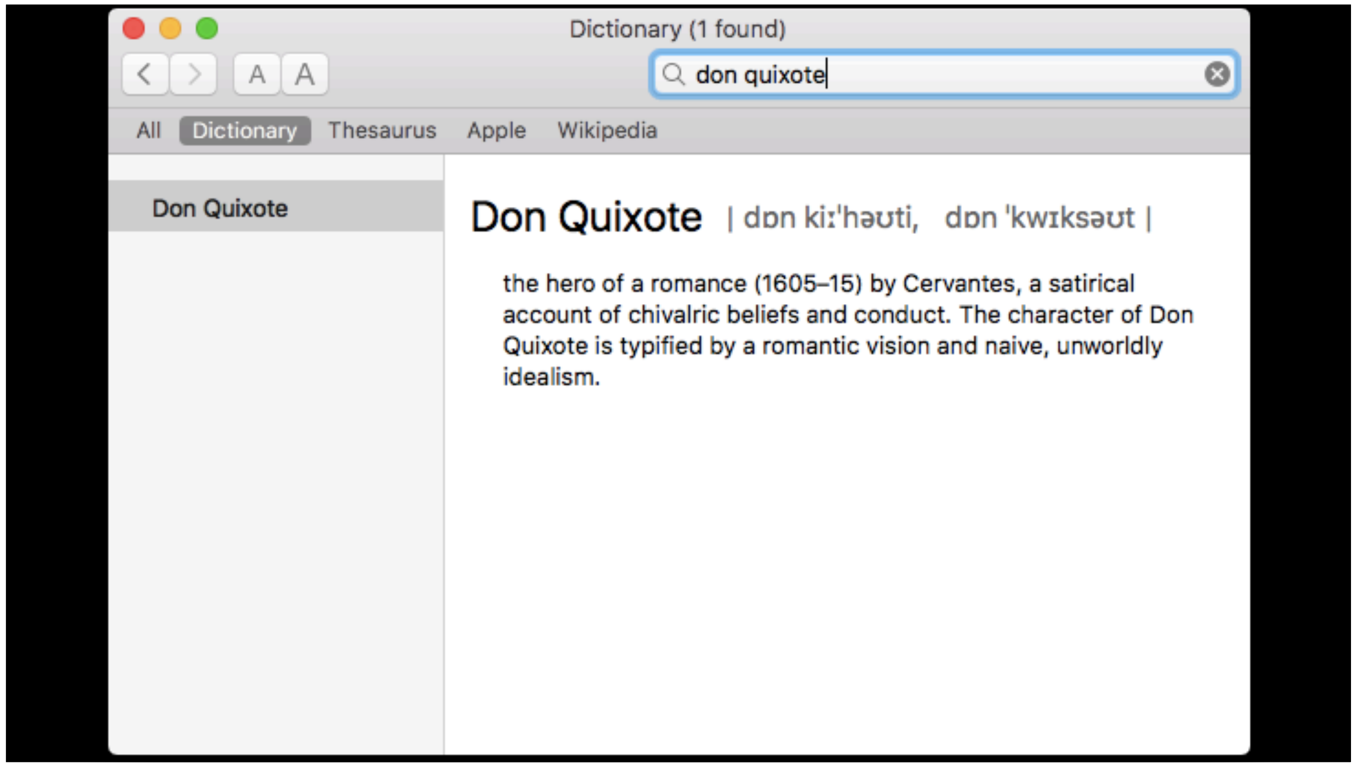
map(key) → value

That's it, that is the core idea of the map function, and maps as we know them as programming concepts.



In Python, maps are called dicts

If you're a python user, you might know maps as dicts, with is short for the english word, dictionary.

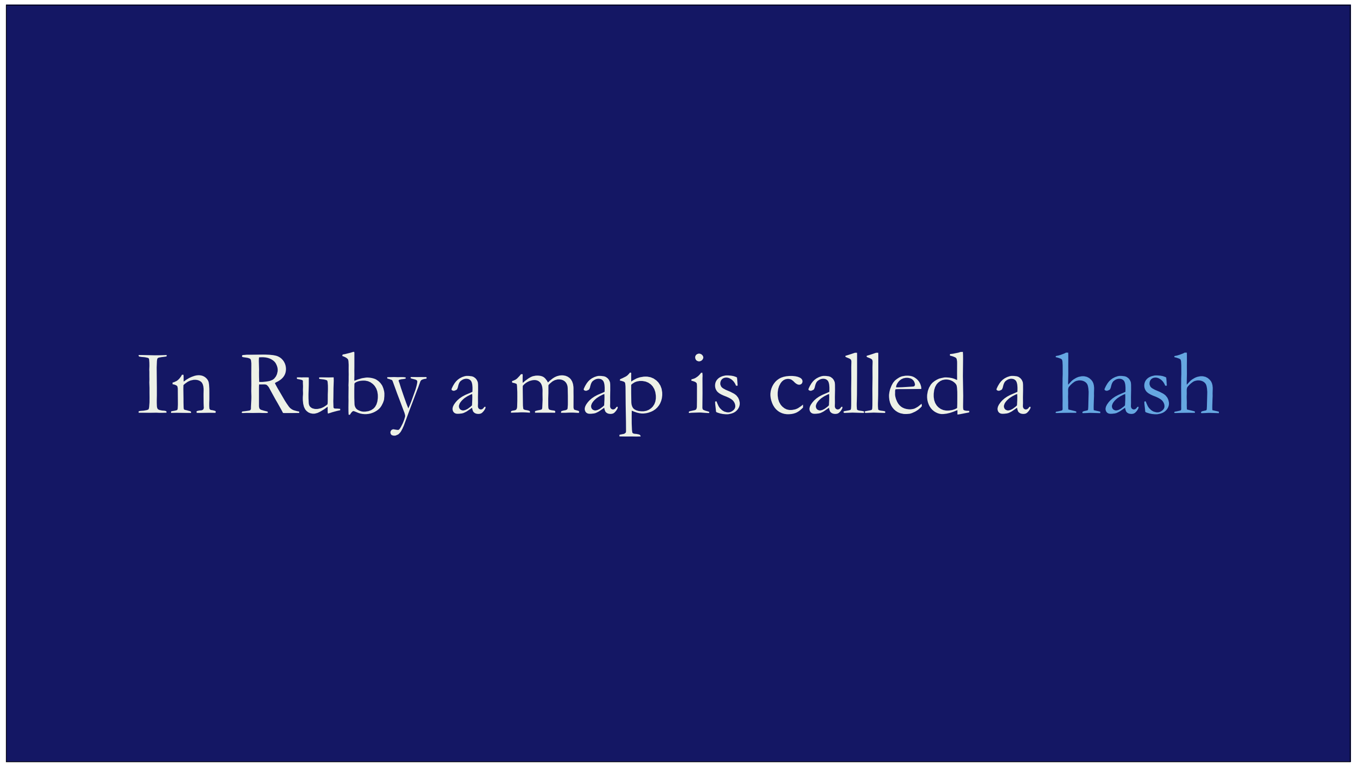


Don Quixote | dɒn kiːˈhəʊti, dɒn ˈkwɪksəʊt |

the hero of a romance (1605–15) by Cervantes, a satirical account of chivalric beliefs and conduct. The character of Don Quixote is typified by a romantic vision and naive, unworldly idealism.

But the same idea applies, I can go from a word to its definition because that is what dictionaries do. The word in the dictionary is itself the key, and the value returned is the words definition

```
% python
Python 2.7.10 (default, Feb  7 2017, 00:08:15)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0
(clang-800.0.34)] on darwin
Type "help", "copyright", "credits" or
"license" for more information.
>>> million = 1000000
>>> population = { "sydney": 5 * million,
"tokyo": 37.8 * million }
>>> population["sydney"]
5000000
```

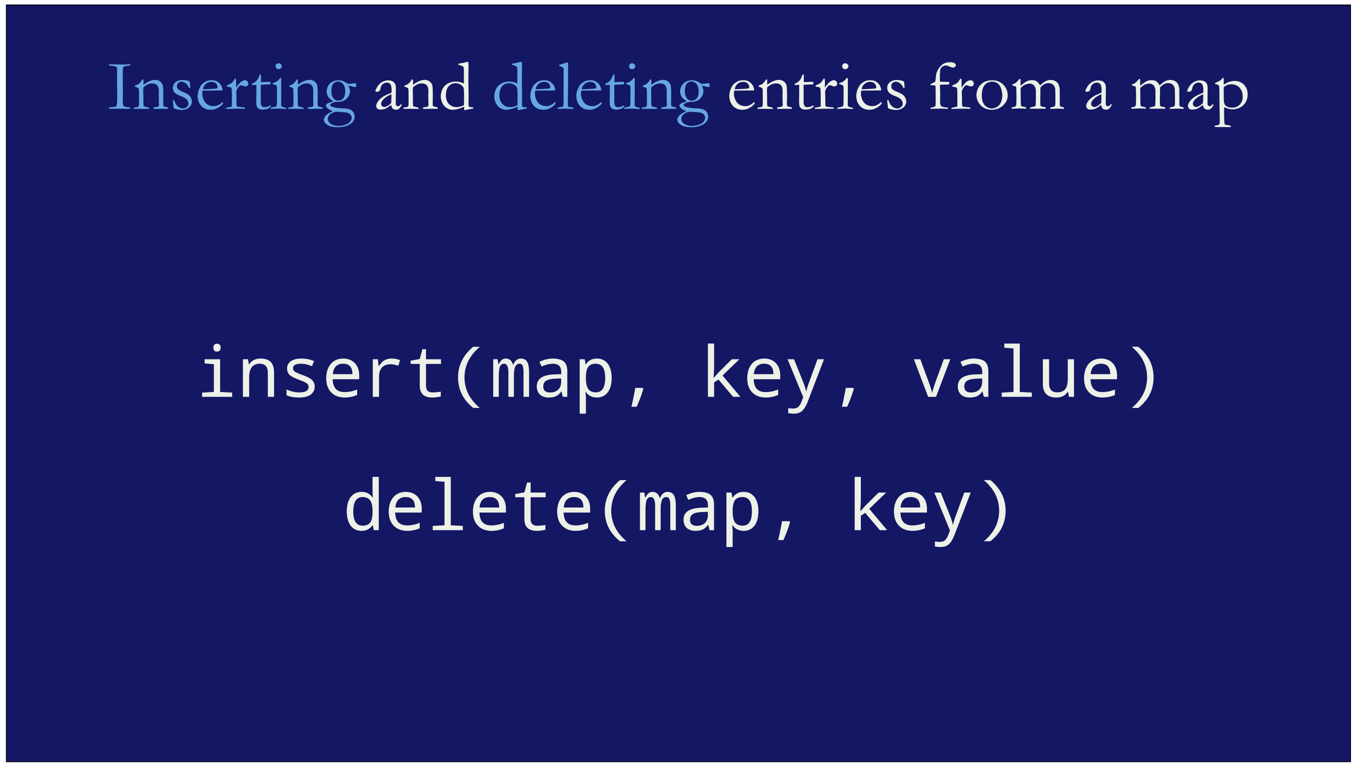


In Ruby a map is called a hash

we’ll come back to why its called a hash in a second

```
% irb
irb(main):001:0> million = 1000000
=> 1000000
irb(main):002:0> population = { :sydney
=> 5 * million, :tokyo => 37.8 *
million }
=> {:sydney=>5000000, :tokyo=>37800000.0}
irb(main):003:0> population[:tokyo]
=> 37800000.0
```

37.8 million point zero



Inserting and deleting entries from a map

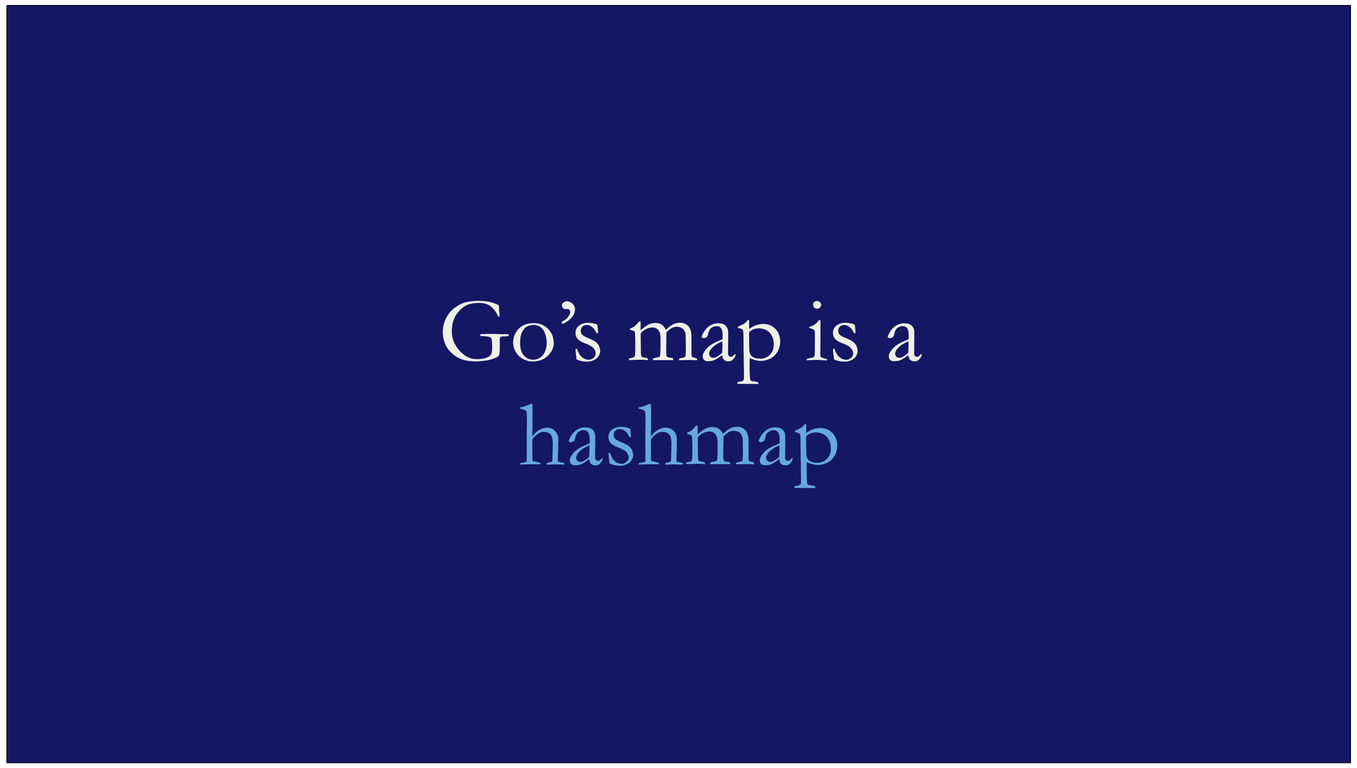
```
insert(map, key, value)
delete(map, key)
```

A map isn't going to be that useful unless we can put some data in the map.

We'll need a function that adds data to the map

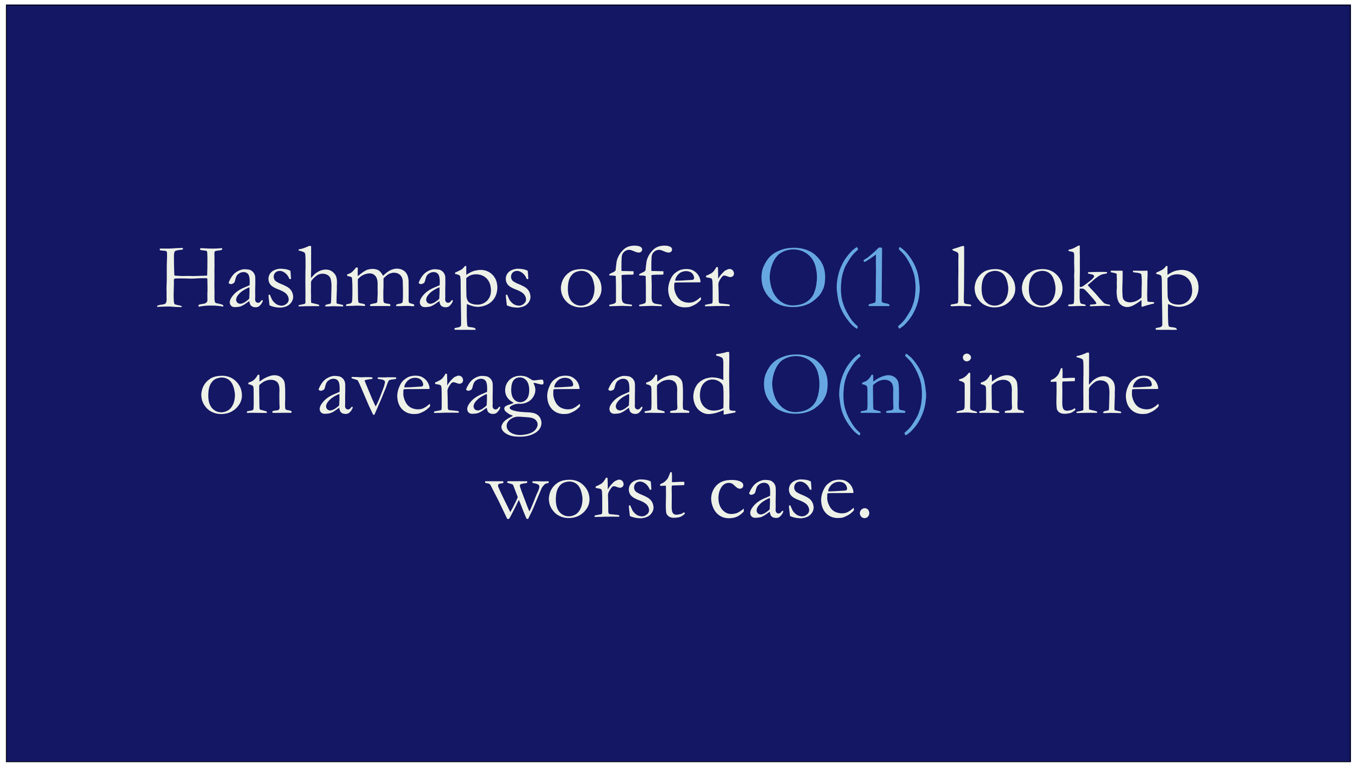
(click)

and a function that removes data from the map



Go's map is a hashmap

The map implementation I'm going to talk about today is the _hashmap_, because this is the implementation that the go runtime uses.


Hashmaps offer O(1) lookup on average and O(n) in the worst case.

A hashmap is a classic data structure because it offers O(1) lookups on average and O(n) in the worst case.

That is, when things are working well the time to execute the map function is a, usually small, constant.

The size of this constant is part of the hashmap design and the point at which the map moves from actual value of O(1) and when and where it becomes O(n) is determined by the hash function.

The hash function

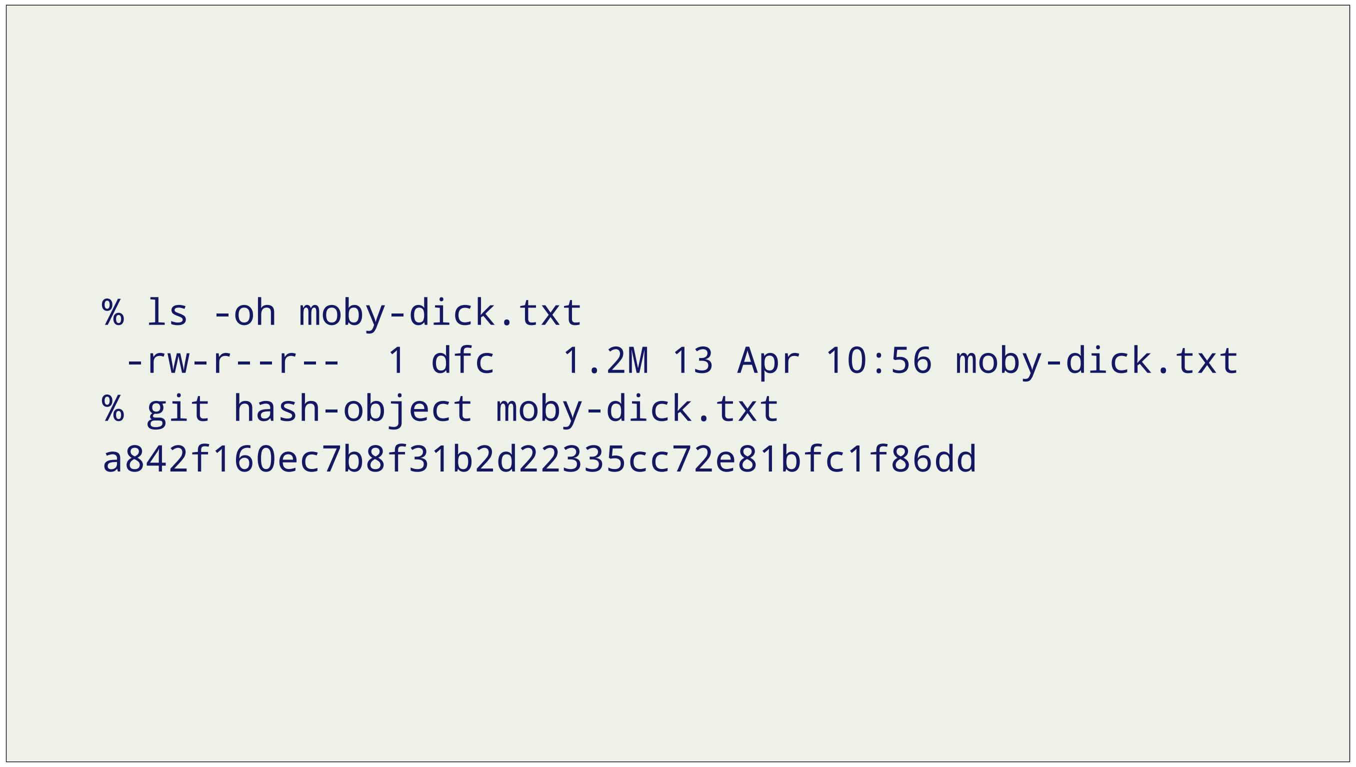
```
hash(key) → integer
```

What is the hash function? A hash function takes a key of an unknown length and returns a value with a fixed length.

hash(key) → integer

that value is almost always an integer, and we’ll see why in a second.

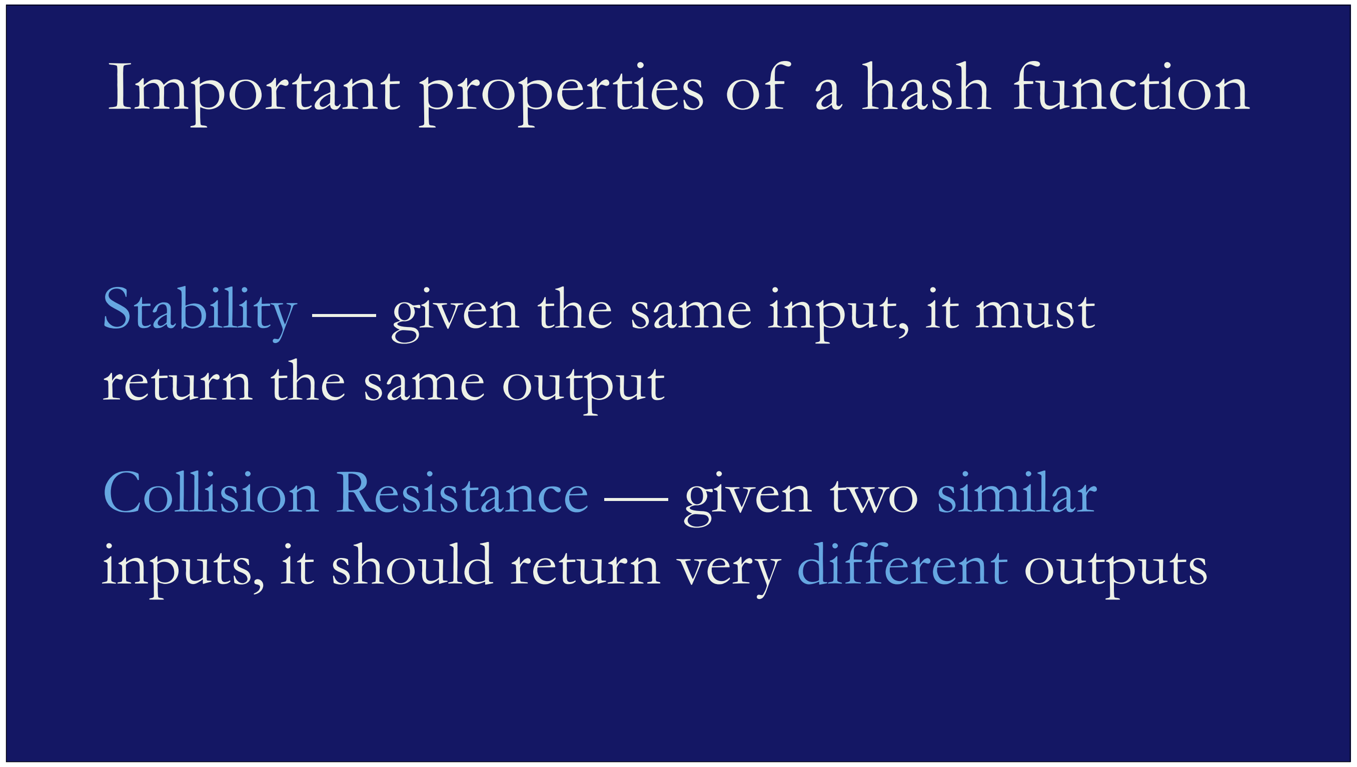
hash and map are similar, they both take a key and return a value, however in the case of hash, it returns a value which is computed as _function_ of the key, not the value associated with the key.



```
% ls -oh moby-dick.txt
 -rw-r--r--  1 dfc   1.2M 13 Apr 10:56 moby-dick.txt
% git hash-object moby-dick.txt
a842f160ec7b8f31b2d22335cc72e81bfc1f86dd
```

Intuitively we know how this works because we all use tools like git every day.

We know that git turns the contents of a blob into the hash of its contents, giving a fixed length value, the sha1, this is how content addressable storage works.



Important properties of a hash function

Stability — given the same input, it must return the same output

Collision Resistance — given two similar inputs, it should return very different outputs

Let’s talk about the properties of the hash function because as I said above the quality of the hash function determines how likely the map function is to run near O(1)

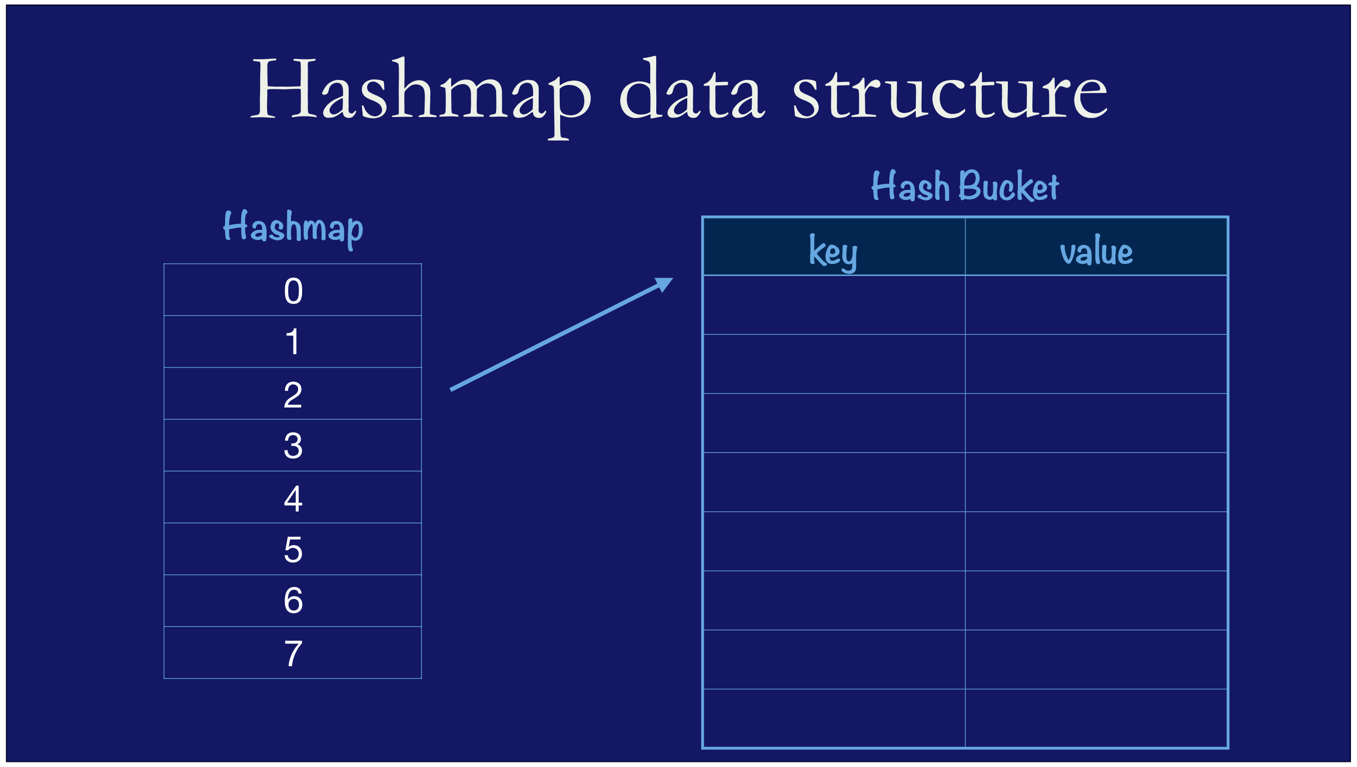
The hash function used by the hashmap has relies on two important properties:

1. It must be stable; give the same key, it must return the same answer. If it doesn't you will not be able to find things you put in the map.

This is why most maps don’t let you update the key stored in a map.

2. It should have good distribution; given two near identical keys, the result should be wildly different.

This is important for two reasons; the first is hash distribution, as we'll see, values in a hashmap should be well distributed over the buckets, otherwise the access time is not O(1), and secondly as the user can control some of the aspects of the data that is hashed, they may be able to control the output of the hash function, leading to poor distribution which has been a DDoS vector for some languages -- I believe there was an interesting attack on the Perl hash a while back.



The second part of a hashmap is the way data is stored inside it.

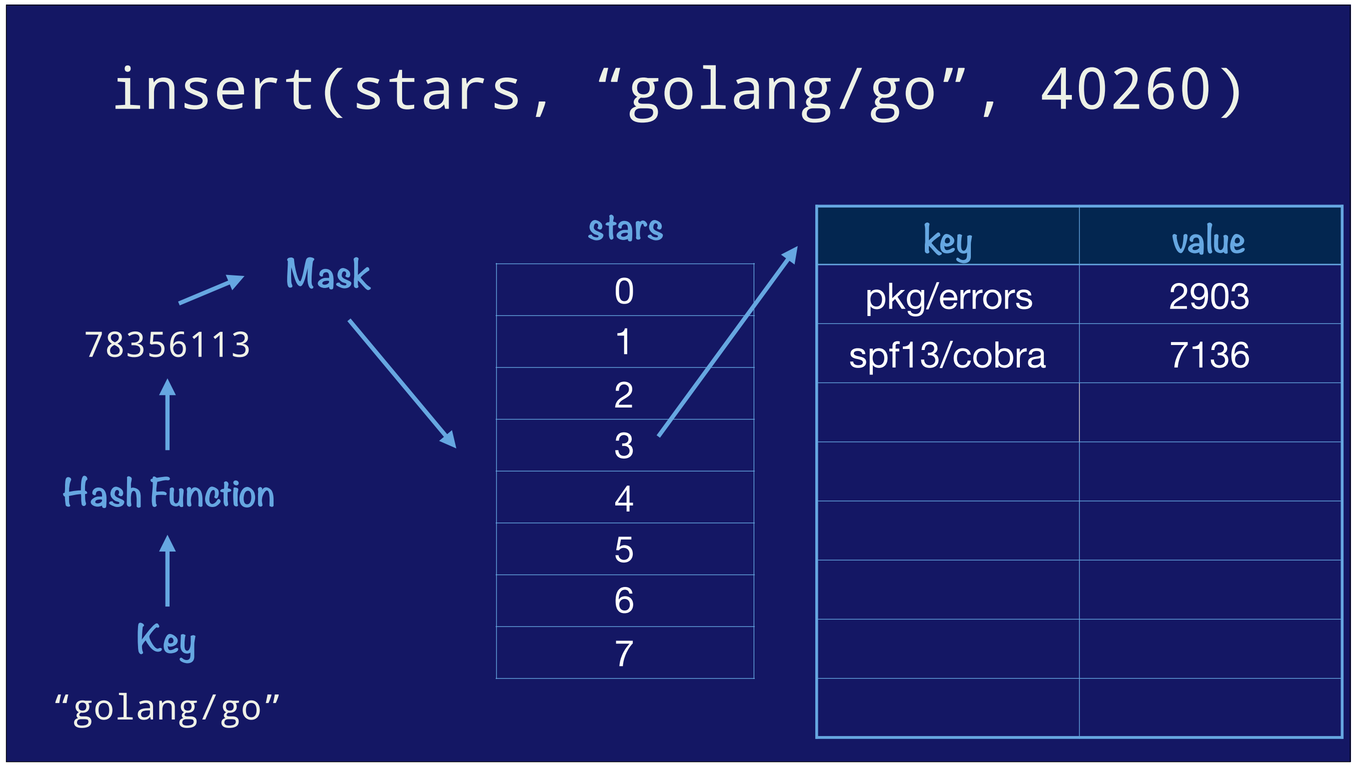
The classical hashmap is an array of "buckets" each of which contain an array of key/value entries

In this case the hashmap has 8 buckets, and each bucket can contain 8 entries each

Its very common to use powers of two everywhere so that you can use cheap bit masking operations rather than expensive division

As the map grows the way to create more space is double the number of buckets and redistribute keys across them

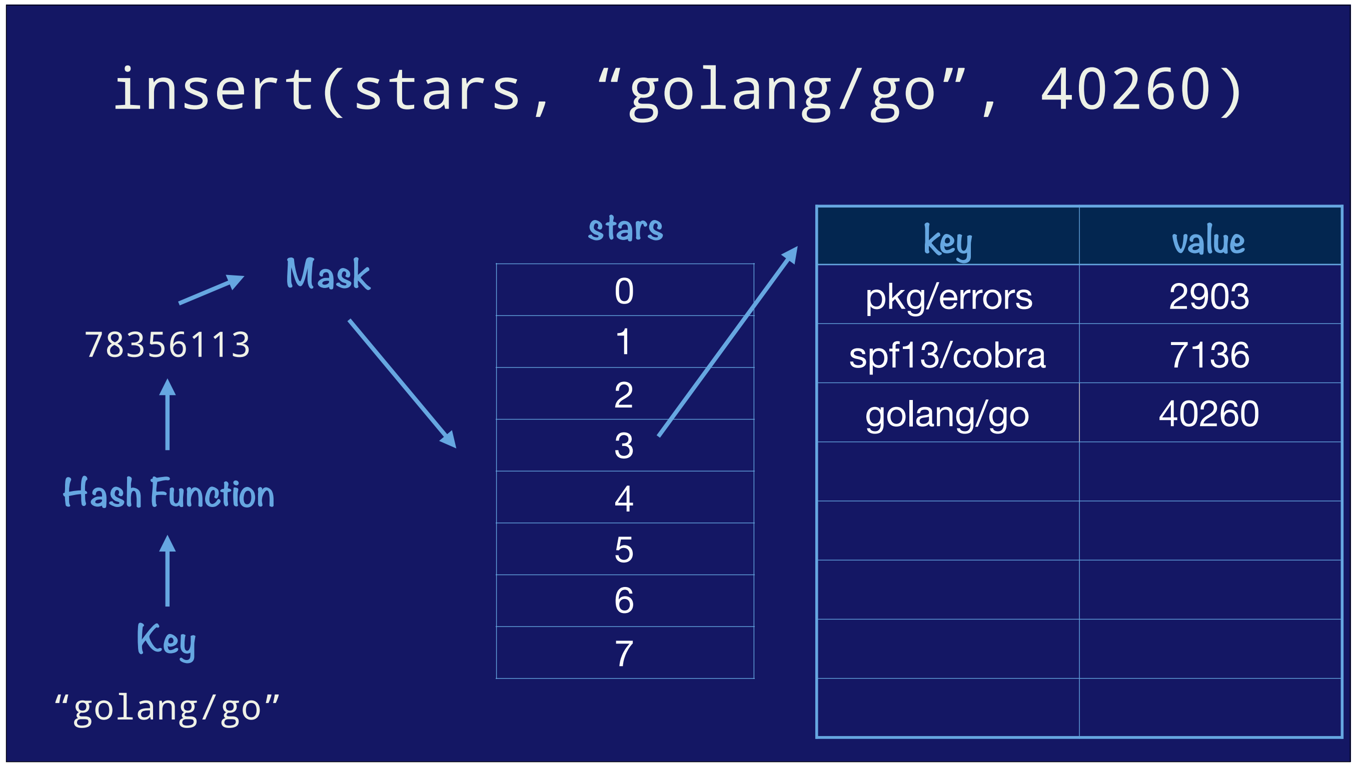
insert(stars, “golang/go”, 40260)



key	value
pkg/errors	2903
spf13/cobra	7136

So, with this hashmap structure, how do we go about inserting a value into the map

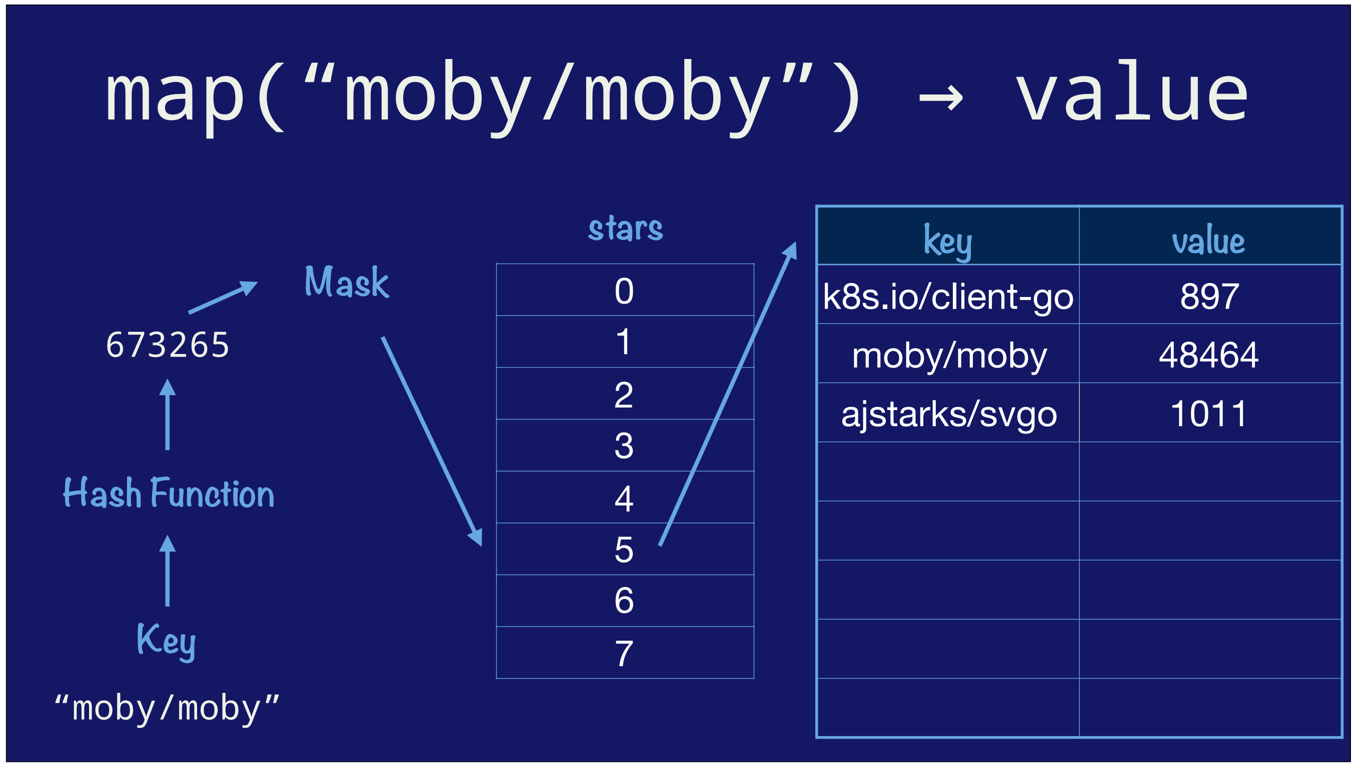
We start with the key


insert(stars, “golang/go”, 40260)

key	value
pkg/errors	2903
spf13/cobra	7136
golang/go	40260

So, with this hashmap structure, how do we go about inserting a value into the map

We start with the key

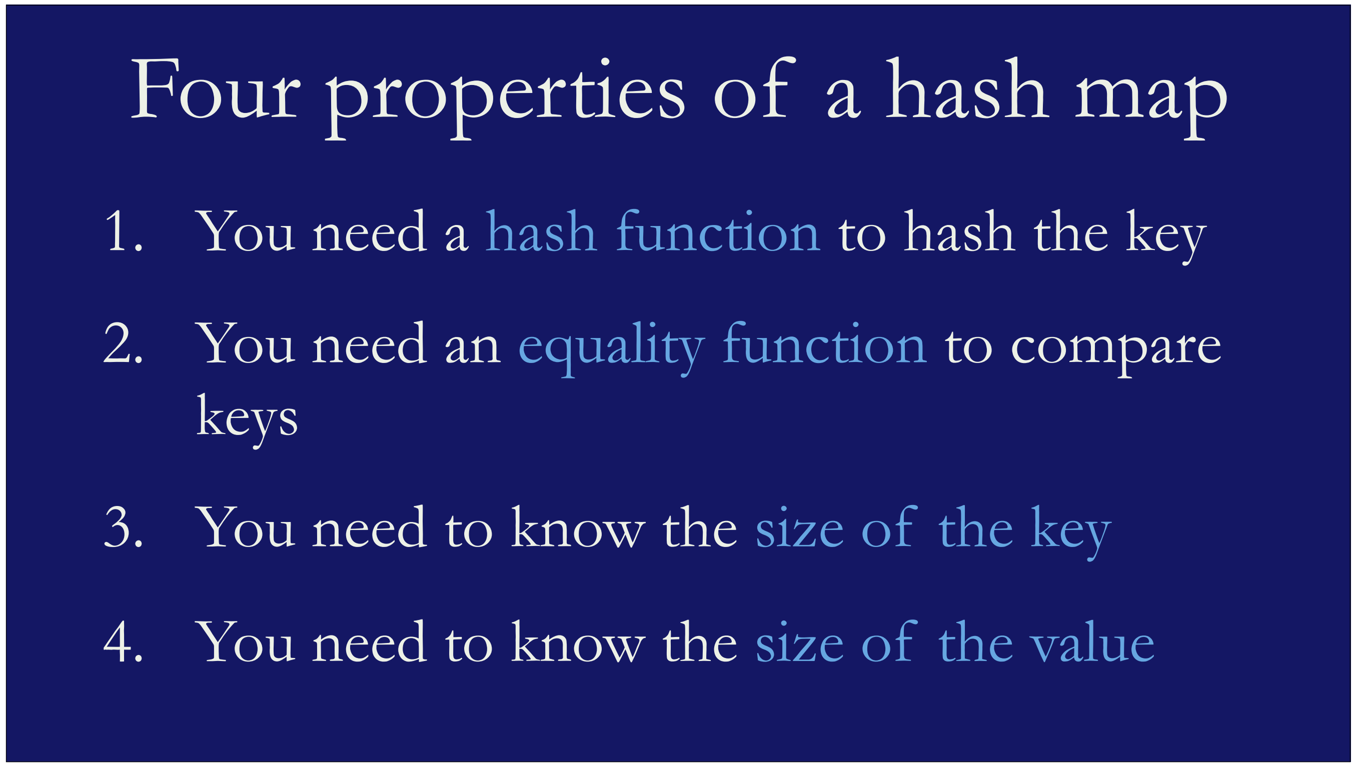


map("moby/moby") → value

key	value
k8s.io/client-go	897
moby/moby	48464
ajstarks/svgo	1011

So, with this hashmap structure, how do we go about inserting a value into the map

We start with the key



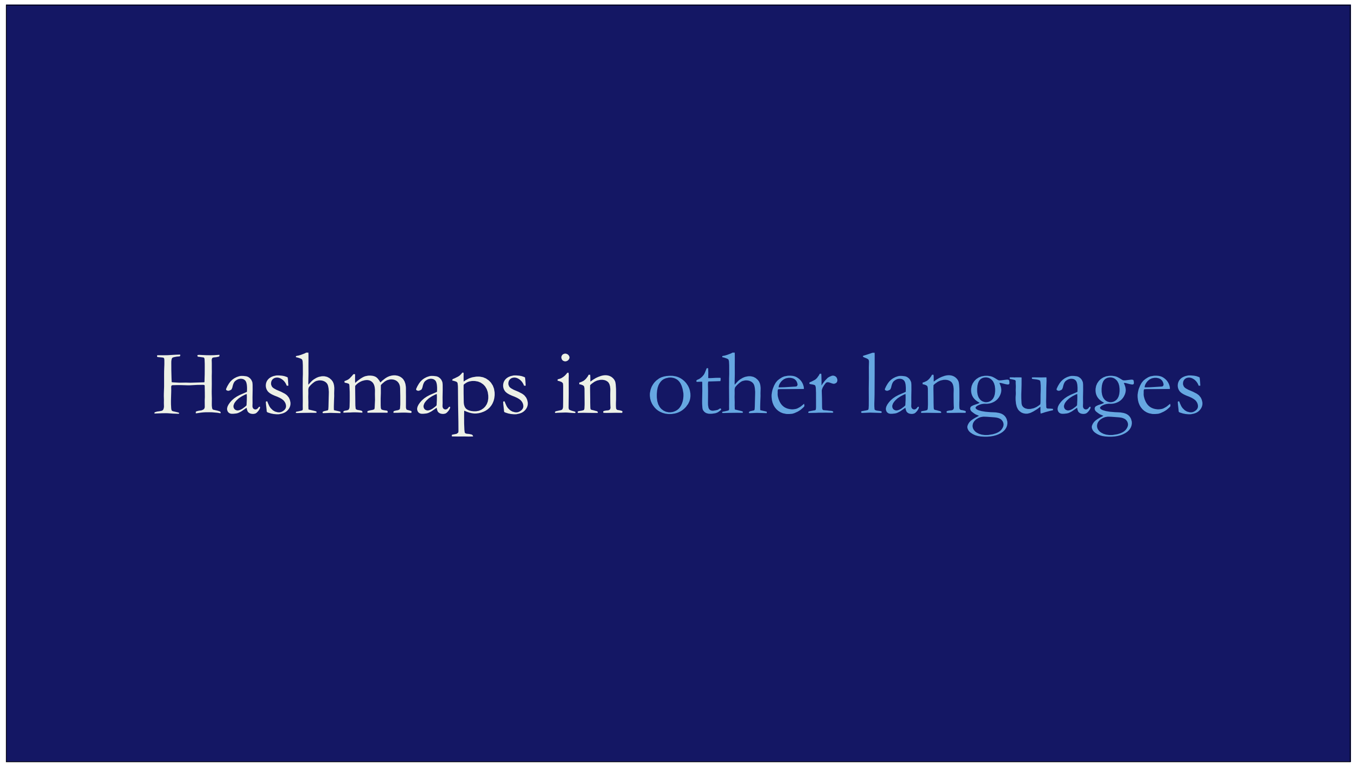
Four properties of a hash map

1. You need a hash function to hash the key
2. You need an equality function to compare keys
3. You need to know the size of the key
4. You need to know the size of the value

That was a very high level explanation of the classical hashmap.

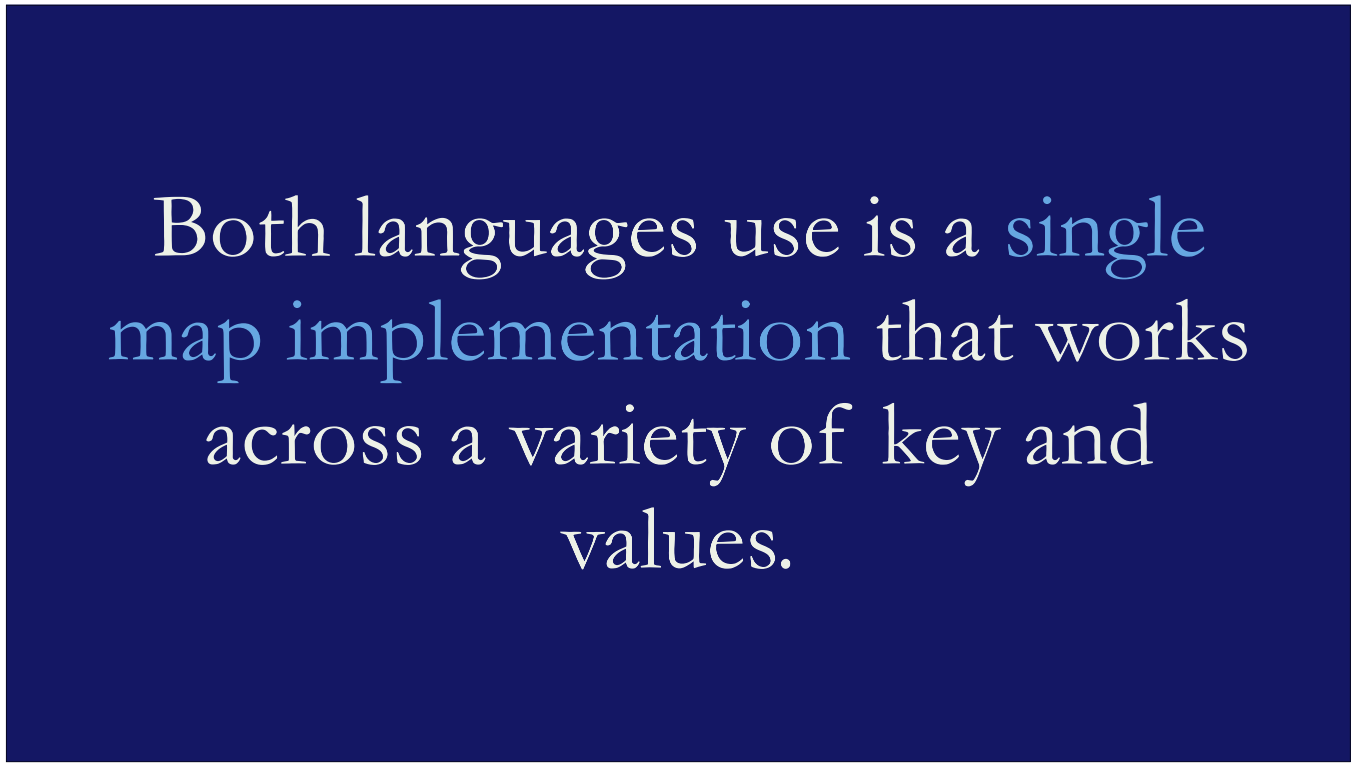
We've seen that the four properties you need to implement a hashmap;

- You need a hash function for the key
- You need an equality function to compare keys
- You need to know the size of the key and the value because they affect the size of the bucket structure and you need to know as you walk or insert into that structure, how far to advance in memory



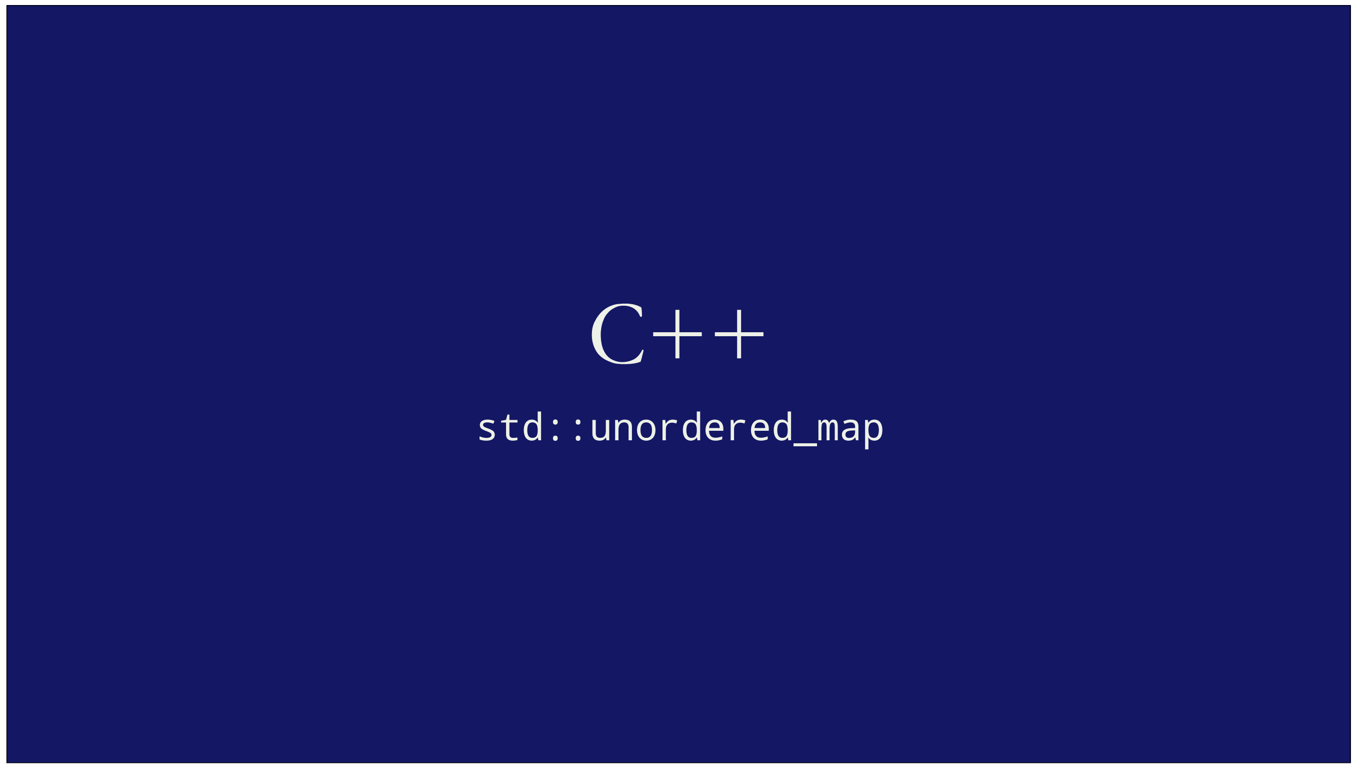
Hashmaps in other languages

Before we talk about the way Go implements a hashmap, I wanted to give a brief overview of how two popular languages implement hashmaps.



Both languages use is a single map implementation that works across a variety of key and values.

In both of these languages there is a single map type that works across a variety of key and values.



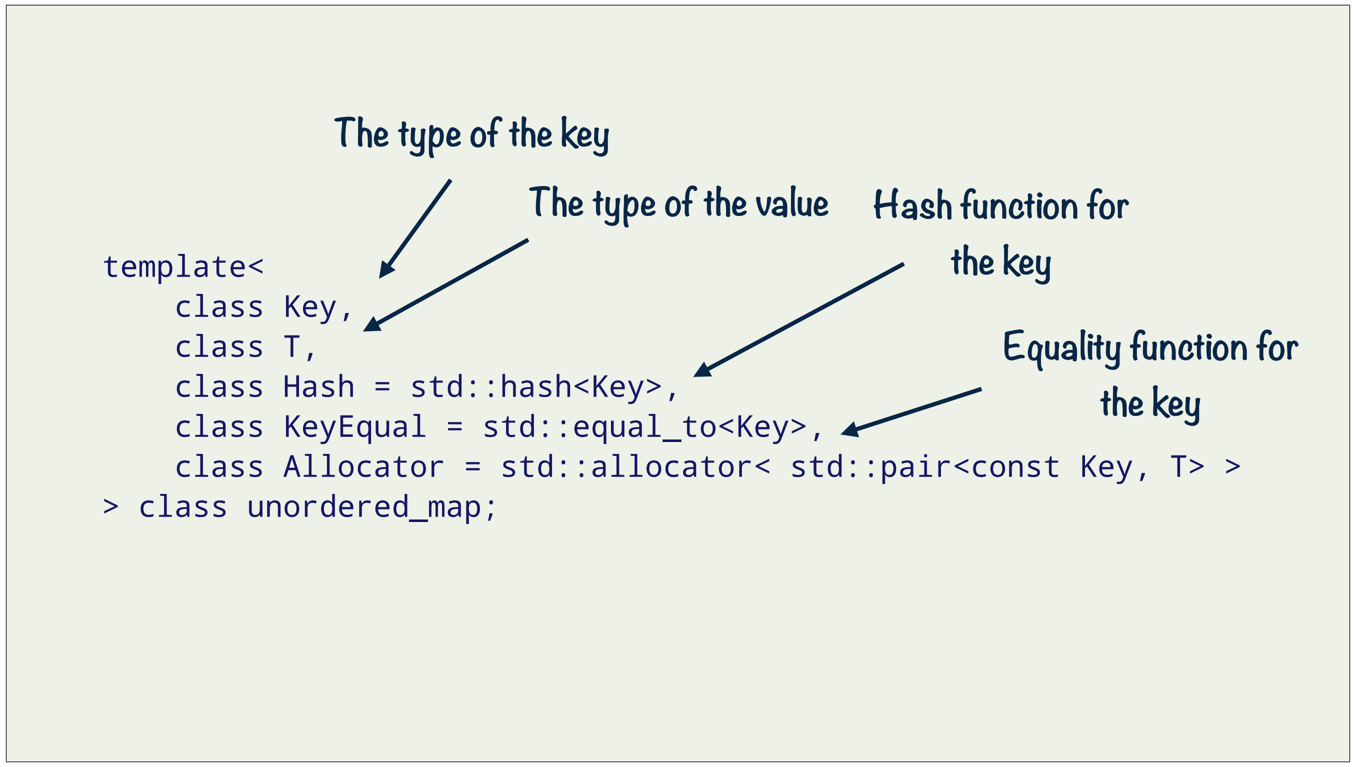
C++

`std::unordered_map`

The first language we’ll discuss is C++.

This is not a talk about C++ so I'm sure I'll get some of the details wrong.

The C++ Standard Template Library (STL) provides `std::unordered_map` which is usually implemented as a hashmap.

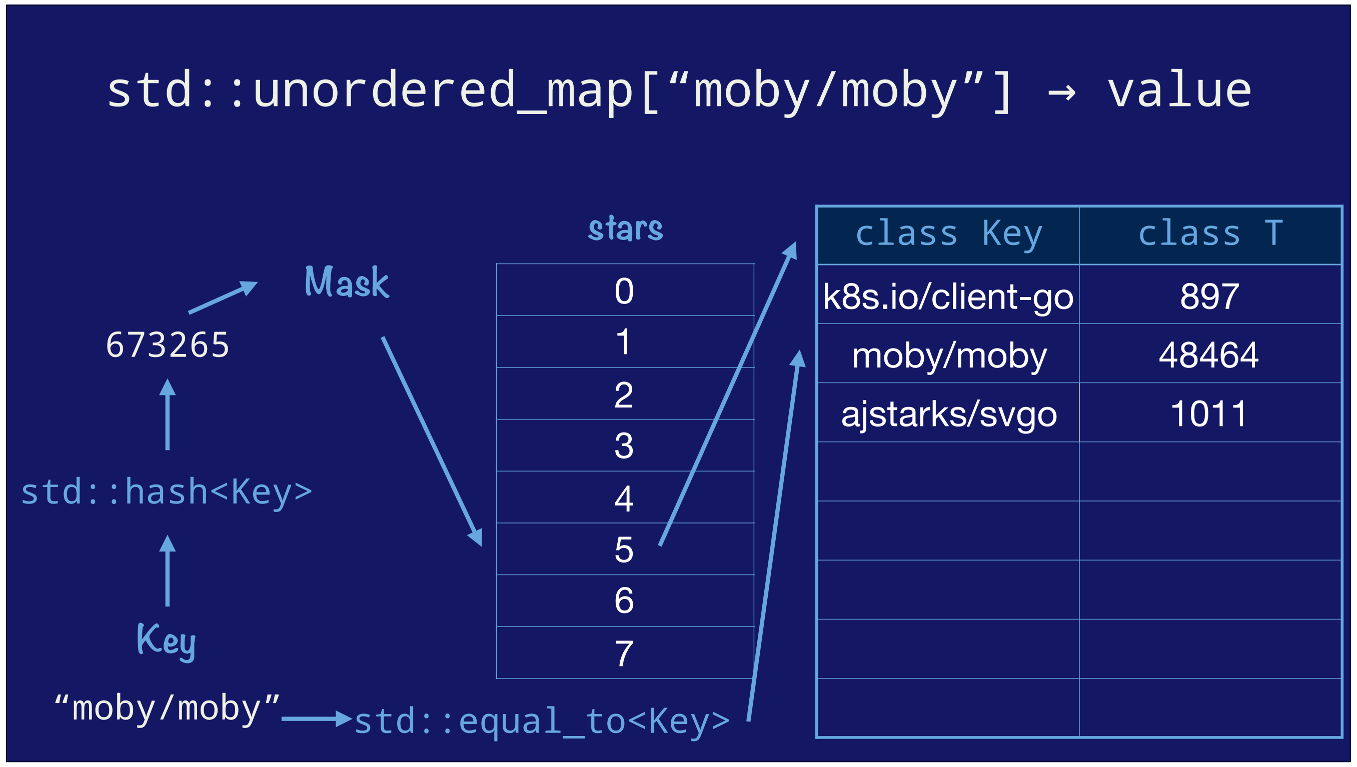

The type of the key
The type of the value
Hash function for the key
Equality function for the key

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
```

This is the declaration for std::unordered_map. It's a template, so the actual values of the parameters depend on how the template is instantiated.

There is a lot here, but the key things to take away are; the template takes the key and value `T` types, so it knows their size.
It takes a hash function specalised on the Key, so it knows how to hash a key passed to it.
And it takes an `equal_to` function, specialised on Key, so it knows how to compare two keys.

You spell out all these things to the compiler when you are constructing an unordered map.



`std::unordered_map["moby/moby"] → value`

class Key	class T
k8s.io/client-go	897
moby/moby	48464
ajstarks/svgo	1011

So, with this hashmap structure, how do we go about inserting a value into an unordered_map

This all works because at the time of construction the compiler has used the template to construct a map implementation specific for exactly this map of strings to integers for this hash function and this comparator

Java

`java.util.HashMap`

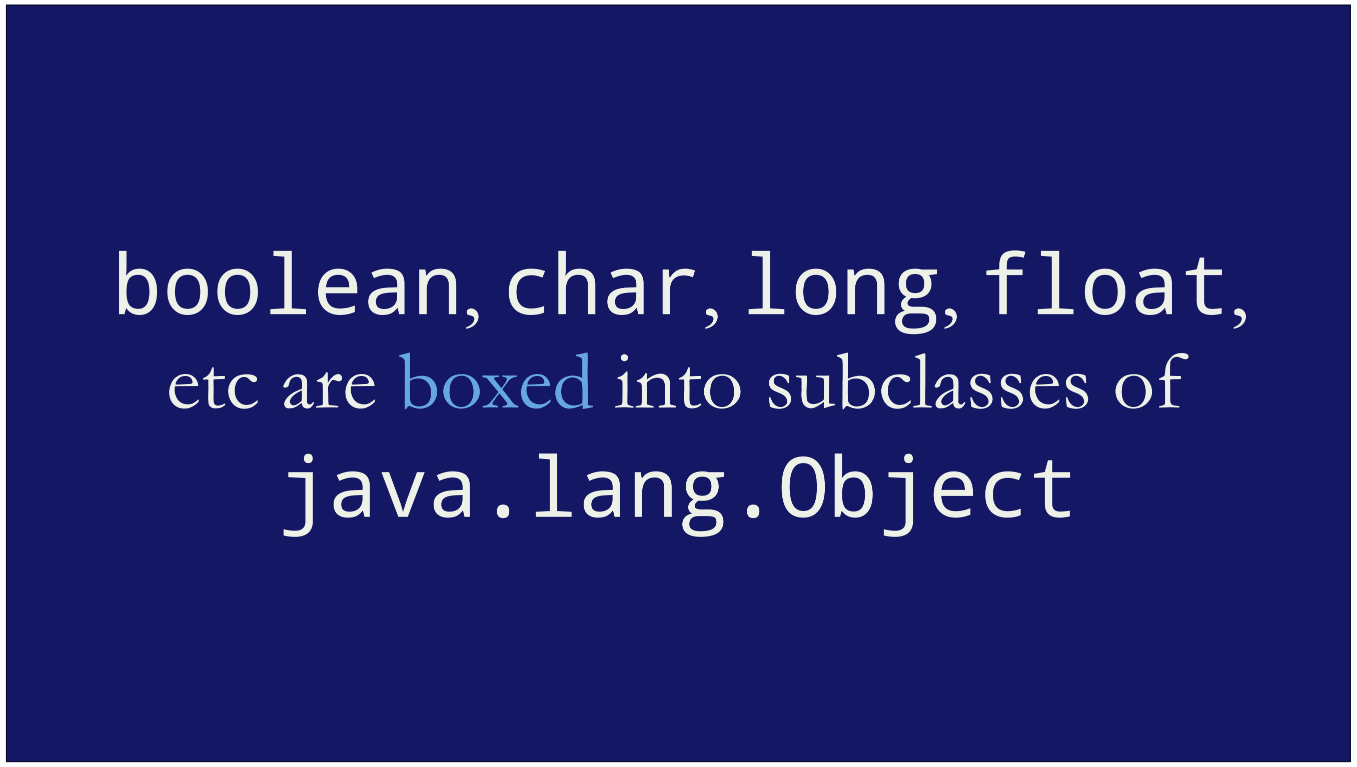
The second language we'll discuss is Java.

In java the hashmap type is called, well, java.util.Hashmap

`java.util.HashMap` can only store `java.lang.Object`s

In java, the +java.util.Hashmap+ type can only operate on objects, which is fine because in Java _almost_ everything is a subclass of +java.lang.Object+.

Because every object in java descends from +java.lang.Object+ they inherit a +hashCode+ and an equals method.



`boolean`, `char`, `long`, `float`, etc are *boxed* into subclasses of `java.lang.Object`

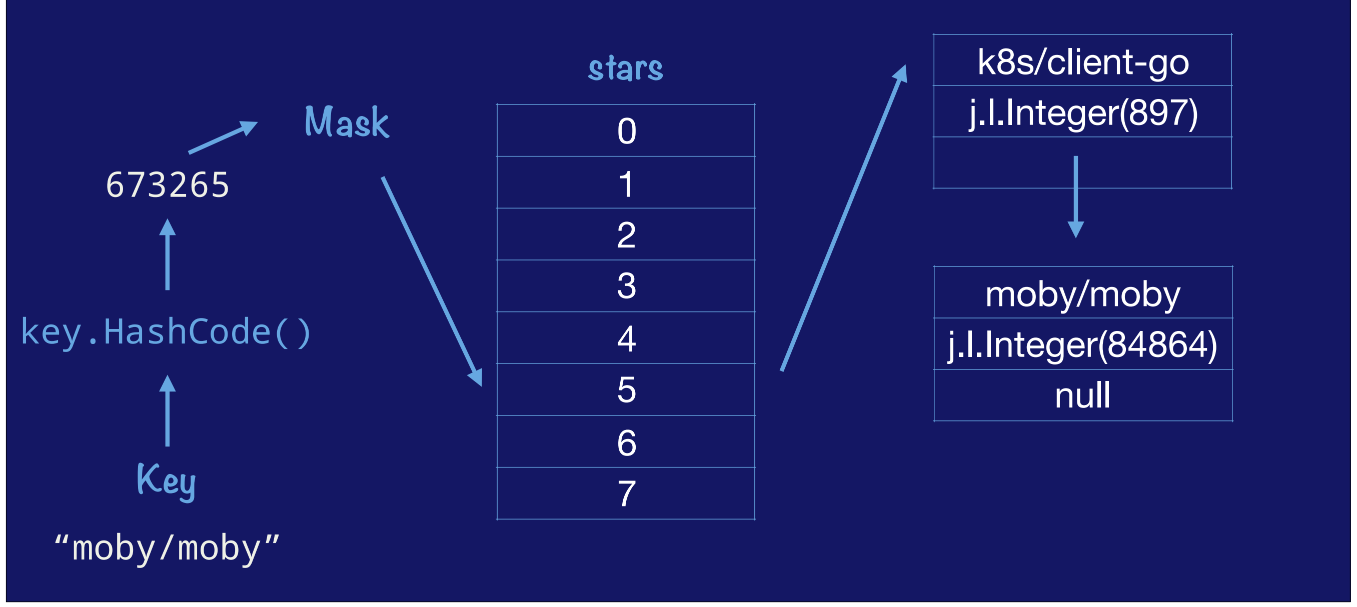
You cannot directly store the seven primatives types; int, short, long, byte, float, double and bool, because they are not subclasss of object

You cannot use them as a key, you cannot store them as a value.

For those they are silently _boxed_ into objects.

This is magic that the compiler does for the Java programmer, which is convenient, but generates a shittonne of garbage, and a shittone of garbage can have performance impacts.

HashMap.get("moby/moby") → value

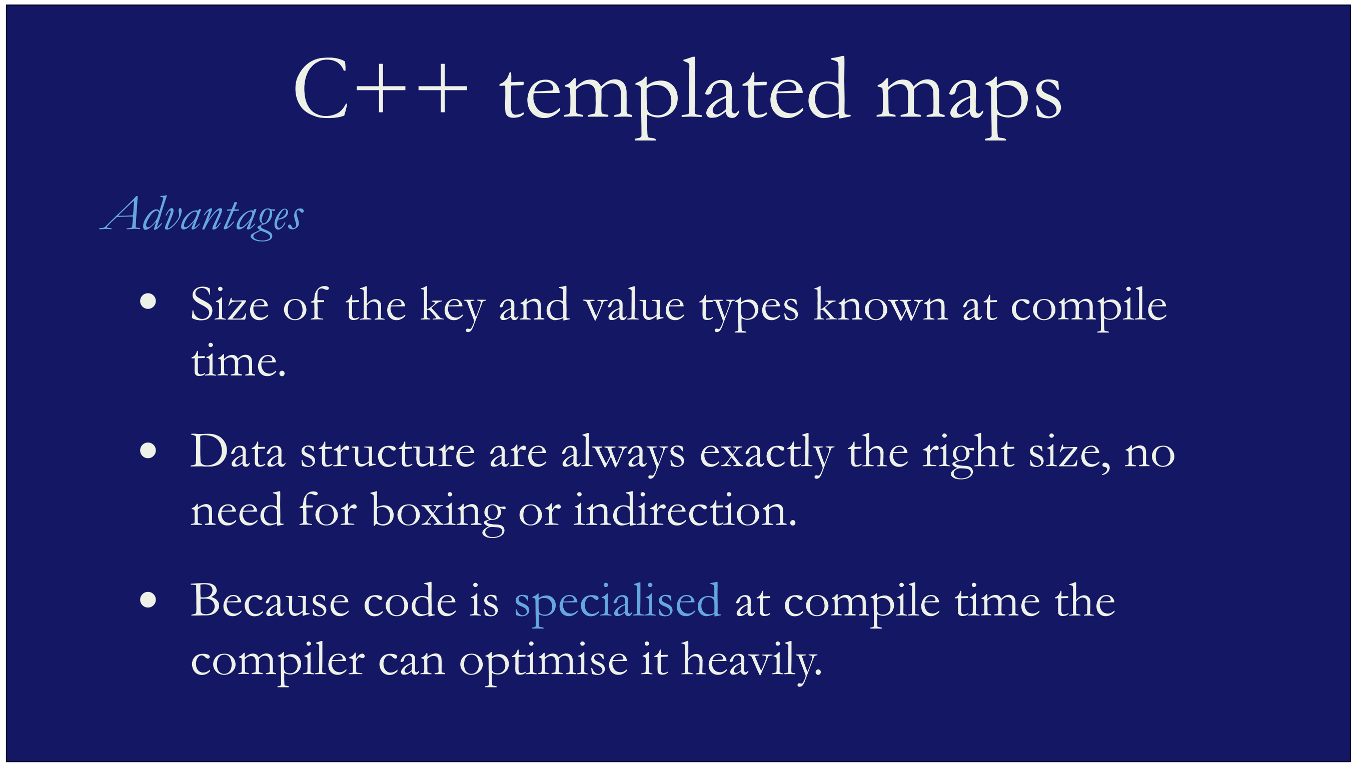


Let's look at how java's hashmap operates

We ask the keys class to hash itself; string is a subclass of object, so it has a hashcode method.

Tradeoffs

Now that we've seen how C++ and Java implement a Hashmap, let's compare their relative advantages and disadvantages



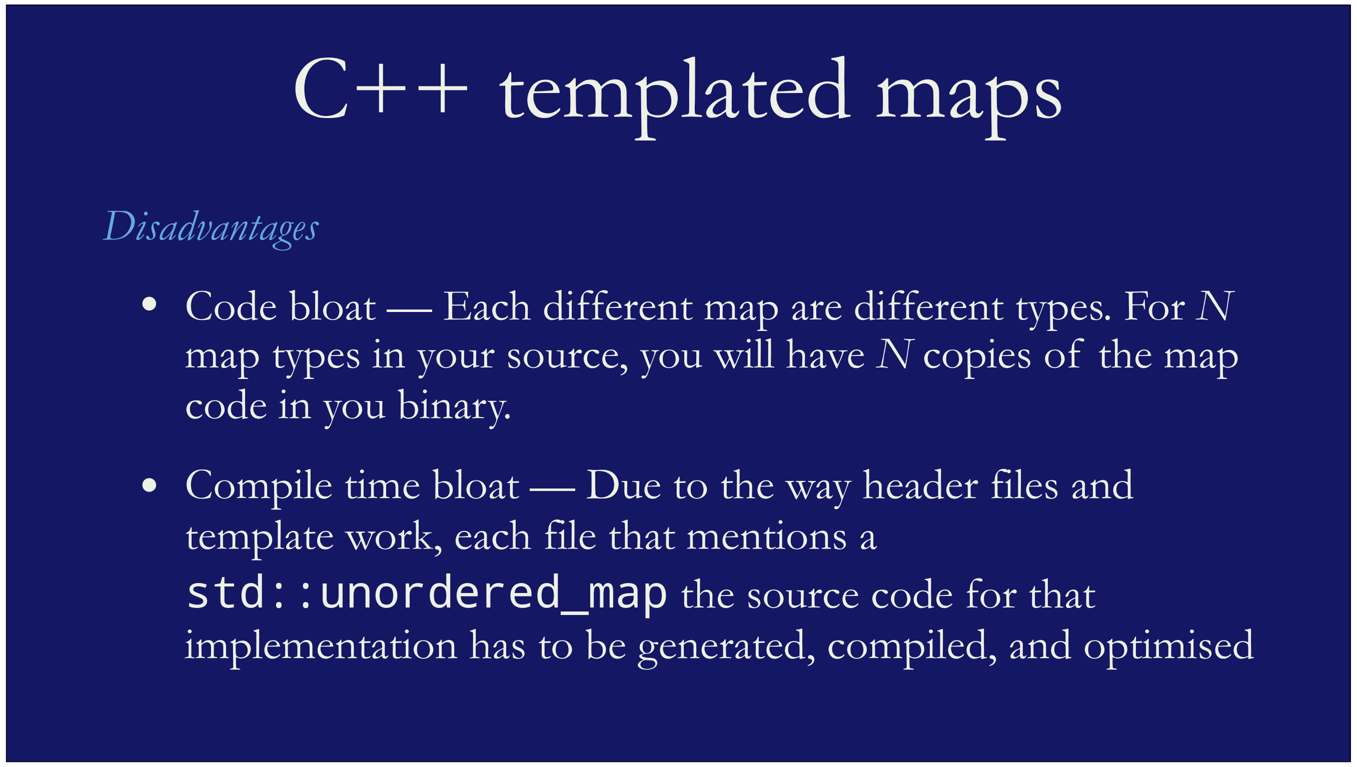
C++ templated maps

Advantages

- Size of the key and value types known at compile time.
- Data structure are always exactly the right size, no need for boxing or indirection.
- Because code is specialised at compile time the compiler can optimise it heavily.

Advantages
Size of the key and value types known at compile time.
Data structure are always exactly the right size, no need for boxing or indiretion.
Because code is specialised at compile time, other compile time optimisations like inlining, constant folding, and dead code elimination, can come into play.

In a word, maps in C++ _can be_ as fast as hand writing a custom map for each K,V combination, because that is what is happening.



C++ templated maps

Disadvantages

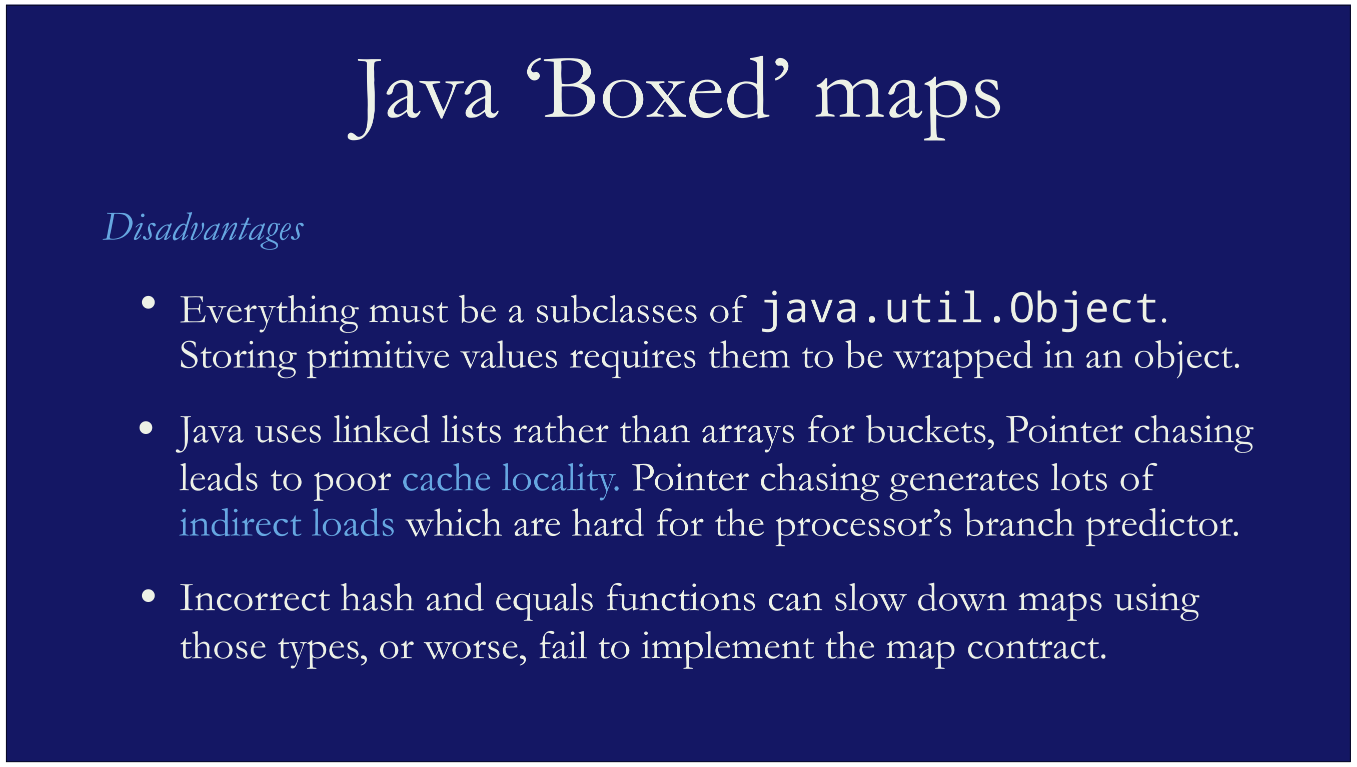
- Code bloat — Each different map are different types. For *N* map types in your source, you will have *N* copies of the map code in you binary.
- Compile time bloat — Due to the way header files and template work, each file that mentions a `std::unordered_map` the source code for that implementation has to be generated, compiled, and optimised

This greatly increases compile time, and then places a lot of work on the linker to deduplicate each copy of the same map's code down to a single copy.

Java ‘Boxed’ maps

Advantages

- One implementation of a map that works for any subclass of `java.util.Object`.
- Only one copy of `java.util.HashMap` is compiled, and its referenced from every single class.



Java 'Boxed' maps

Disadvantages

- Everything must be a subclasses of `java.util.Object`. Storing primitive values requires them to be wrapped in an object.
- Java uses linked lists rather than arrays for buckets, Pointer chasing leads to poor cache locality. Pointer chasing generates lots of indirect loads which are hard for the processor's branch predictor.
- Incorrect hash and equals functions can slow down maps using those types, or worse, fail to implement the map contract.

Everything must be an object, even things which are not objects, this means maps of primative values must be 'converted' to objects via boxing. This adds gc pressure for 'wrapper' objects, and cache pressure because of additional pointer indirections (each object is effective another pointer lookup)

Java uses a linked list of entries, rather than an array for buckets, this leads to lots of pointer chasing while comparing objects.

Pointer chasing is bad for cache locality; rather than storing values in a sequence, they are tied together with pointers leading to a lot of loading from main memory and cache thrashing

Indirect loads are hard for the processor to speculate on; if you are following a chain of pointers, you don't know the address of the nexd pointer until you've loaded the current one. Storing values sequentally in memory allows the processor to preload subsiquent values

Hash and equality functions are left as an exercise to the author of the class. Incorrect hash and equals functions can slow down maps using those types, or worse, fail to implemnt the map behaviour.

Go’s Hashmap implementation

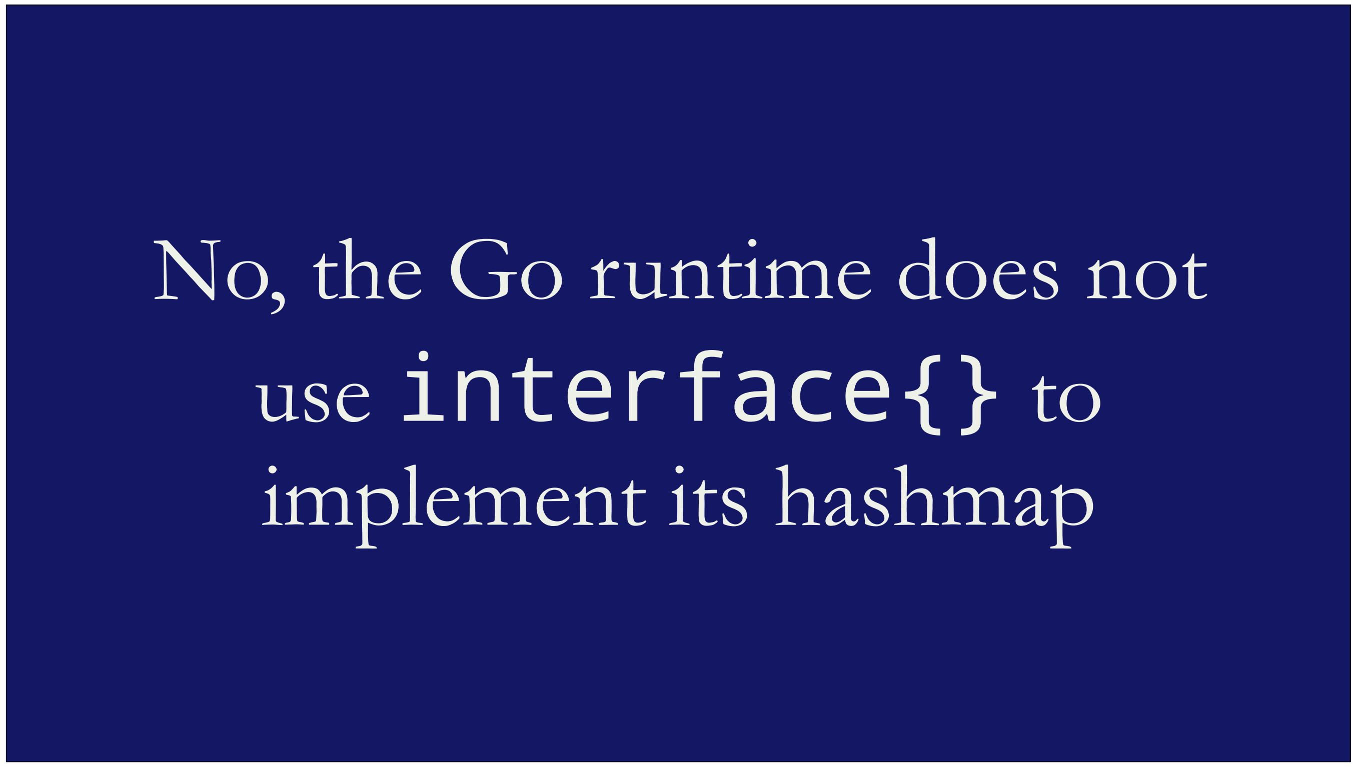
Now, let's talk about how hashmap implementation in Go allows us to retain many of the benfits of the best map implementations without paying for the disadvantages.

Just like C++ and just like Java, Go's hashmap written _in Go_!

But; Go does not provide generic types, so how can we write a hashmap that works for (almost) any type, in Go?

Does the Go runtime use `interface{}`?

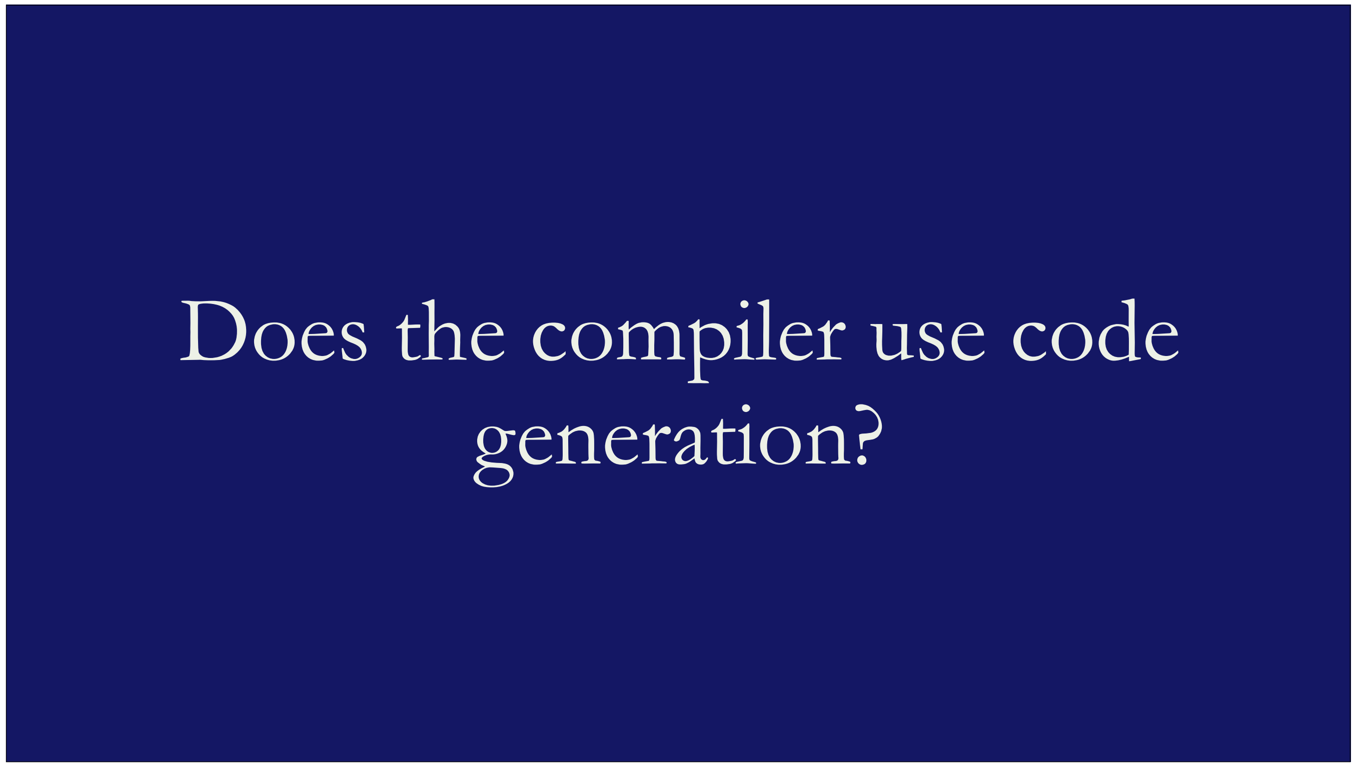
Does the runtime use interface{}?



No, the Go runtime does not use `interface{}` to implement its hashmap

No.

While we have the container/{list,heap} packages which do use interface{}, the runtime's map implementation does not use that.



Does the compiler use code generation?

Does the compiler use code generation?

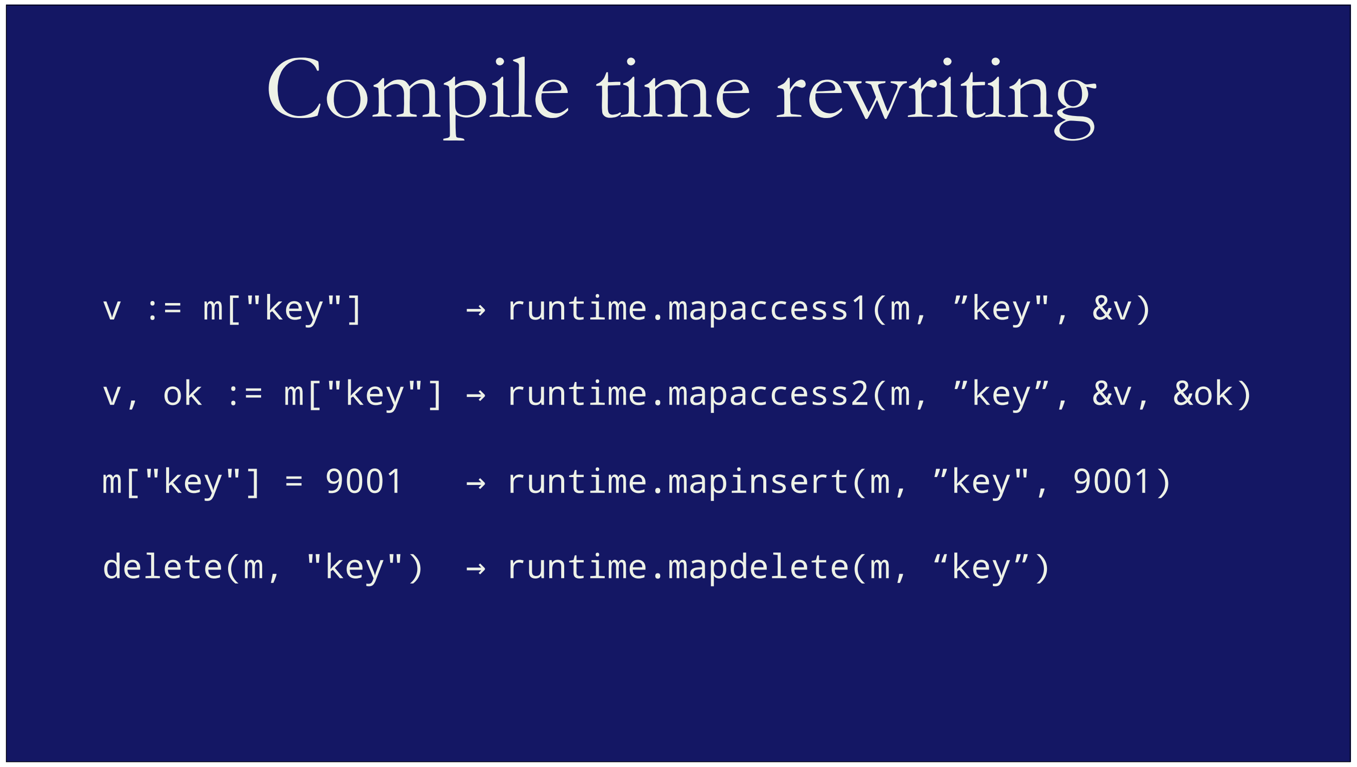
No, there is only one copy of the map implementation in a Go binary

No.

There is only one map implementation, and it doesn't use interface{} boxing; like Java.

So, how does it work.

There are two parts to the answer, and they both involve co-operation between the compiler and the runtime.



Compile time rewriting

```
v := m["key"]     → runtime.mapaccess1(m, ”key", &v)

v, ok := m["key"] → runtime.mapaccess2(m, ”key”, &v, &ok)

m["key"] = 9001   → runtime.mapinsert(m, ”key", 9001)

delete(m, "key")  → runtime.mapdelete(m, “key”)
```

The first part of the answer is to understand that map looksup, insertion and removal, is implemented in the runtime package

If we look at the runtime package, there are no functions dealing with maps, but they are there, they are just not exported.
You just can't access them from normal Go code, for a bunch of reasons that we'll see in a minute.

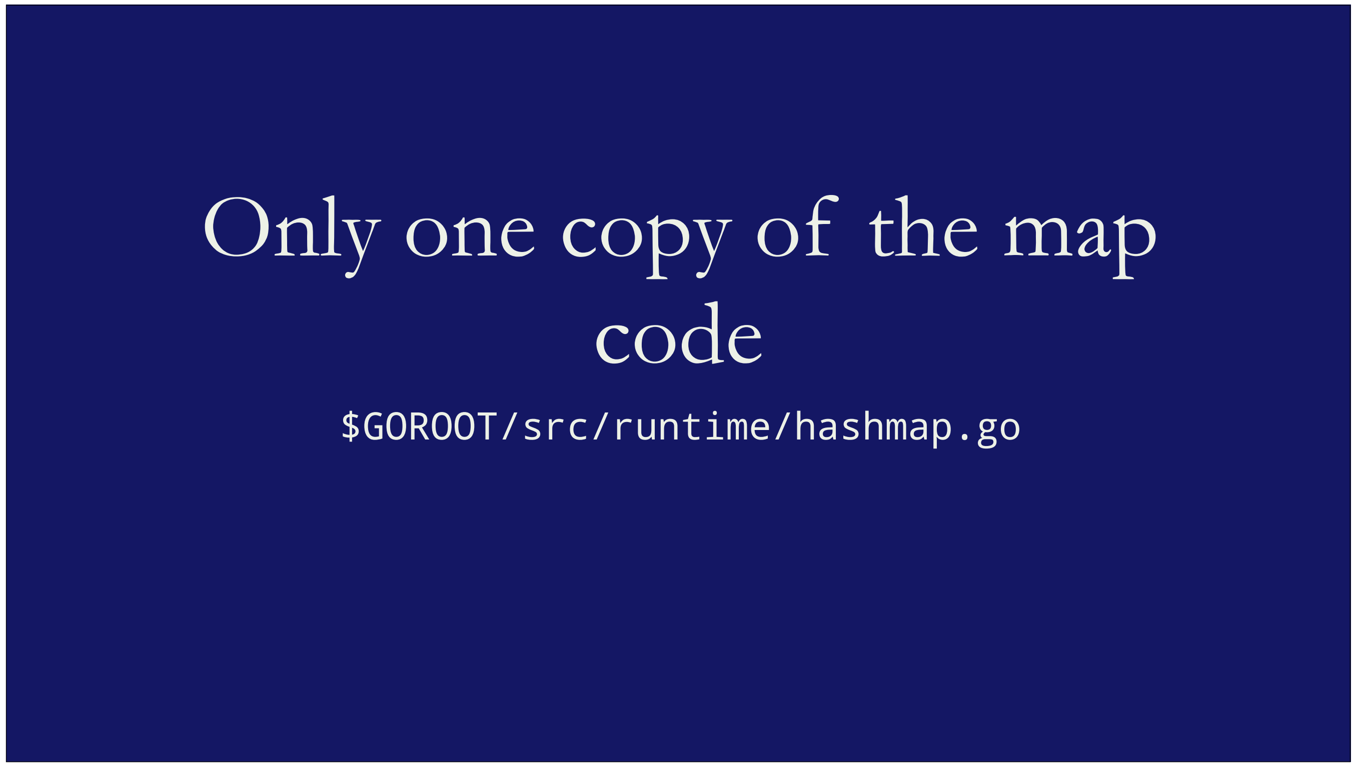
What happens is during compilation map operations are rewritten to calls to the runtime

(click, 4)

It's also useful to note that the _same_ thing happens with channels, but _not_ with slices.

Channels are complicated data types, send, receive, and select has complex interactions with the scheduler so that's delegated to the runtime, but slices are, by comparison, a much simpler data structure, so the compiler natively handles opertaions like slice access, len and +cap+.

the only one that is sometimes handled by the runtime is append

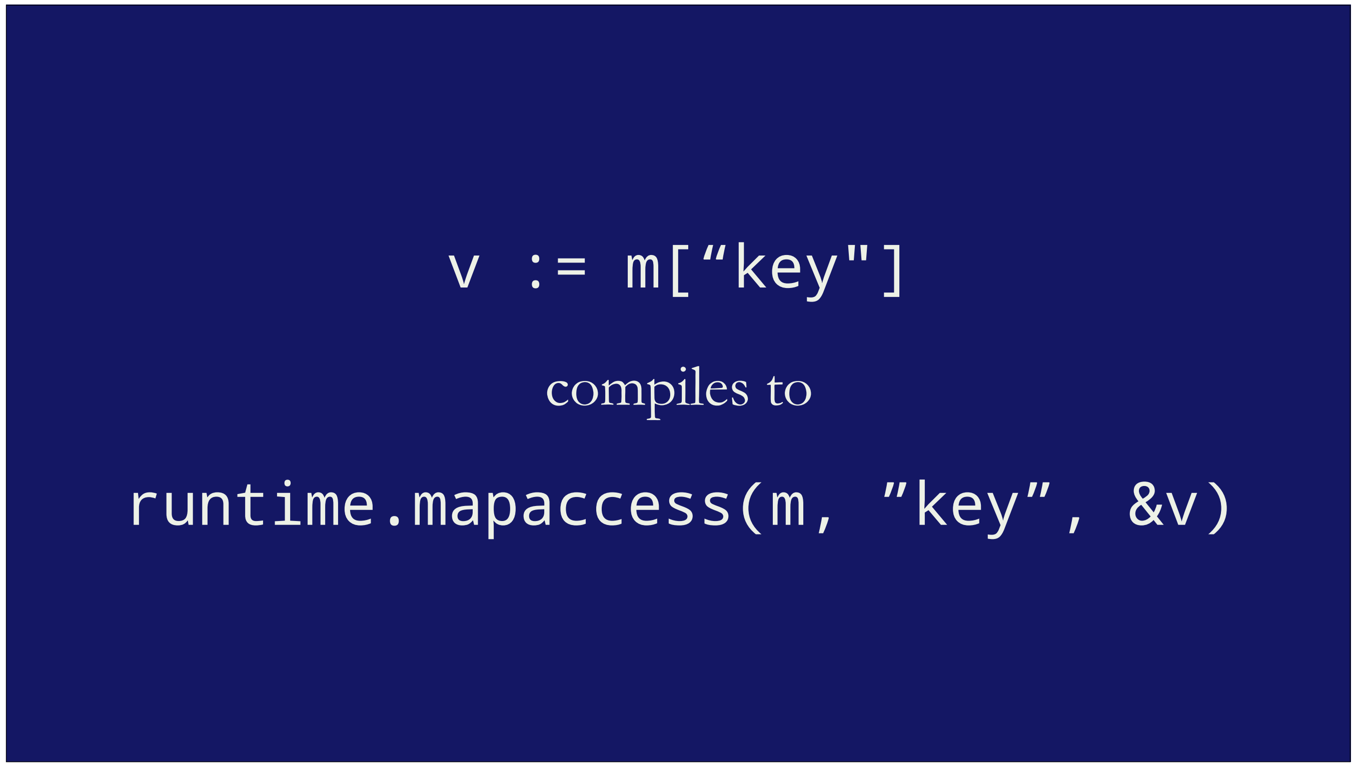


Only one copy of the map code

`$GOROOT/src/runtime/hashmap.go`

So now we know that the compiler rewrites map operations to calls to the runtime, we also know that inside the runtime, because this is Go there is only one functino called `mapaccess`, one function called `mapaccess2`, and so on.

There is no specialisation, and the runtime does not change depending on the types of the map keys and values that are use in your program.
So how does this work?



```
v := m[“key"]
```

compiles to

```
runtime.mapaccess(m, ”key”, &v)
```

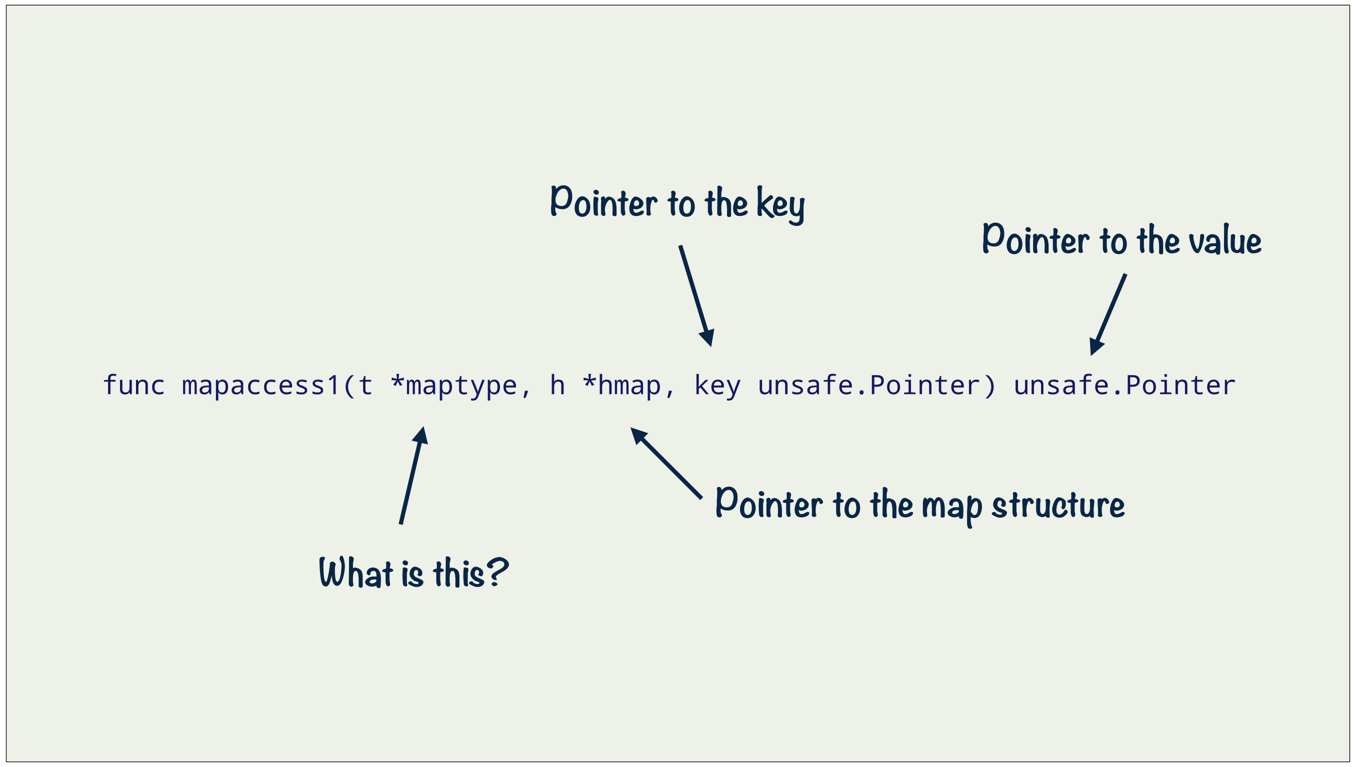
The Go compiler does not support generics, the runtime map functions are not generic, yet the compiler can rewrite

v := m[“key”]

into

 runtime.mapaccess("key", &v)

How can this work?



```
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer
```

The easiest way to explain how map types work in Go is to show you the actual signature of the +runtime.mapaccess1+ function

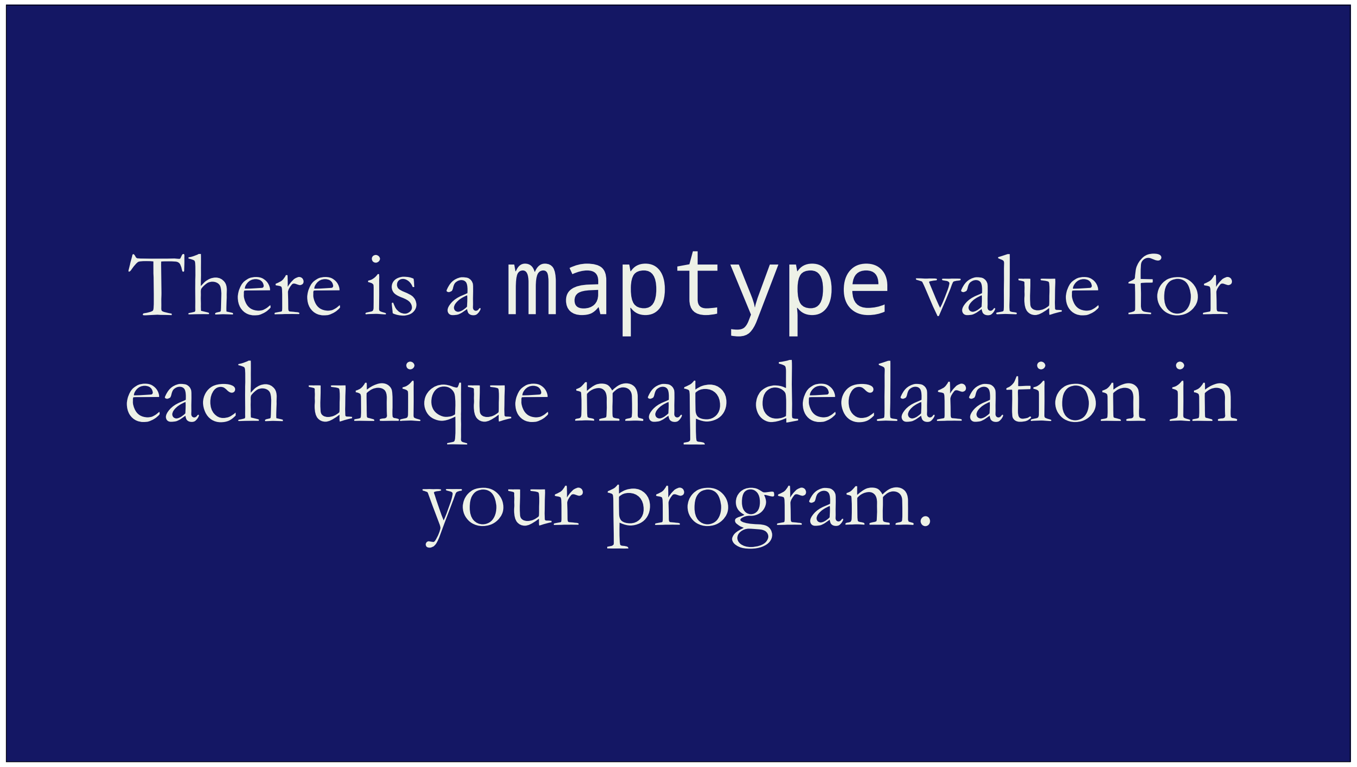
 func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer

Let's walk through the paramters.

- key is a pointer to the key, this is the value you use as the key
- h is a pointer to a hmap structure. hmap is the runtime's hashmap structure that holds the buckets and other housekeeping values.
(You can read more about the runtime.hmap structure here, https://dave.cheney.net/2017/04/30/if-a-map-isnt-a-reference-variable-what-is-it)
- t is a thing called a *maptype, which is odd. Why do we need a *maptype if we already have a *hmap.

`maptype` is the special sauce that makes a single `runtime.hmap` work for (almost) any key and value

maptype is the special sauce that makes the _generic_ hmap work with any map.



There is a `maptype` value for each unique map declaration in your program.

There is a map type value for each unique map declaration in your program. There will be one that describes maps from strings to ints, from strings to httpheaders, from ints to user objects.

This is the secret sauce, rather than having, as C++ has, a complete map implementation for each unique map declaration, the Go compiler creates a map type value and then passes that to the generic map functions depending on the type of the arguments in the program.

The type of the map key

The type of the map value

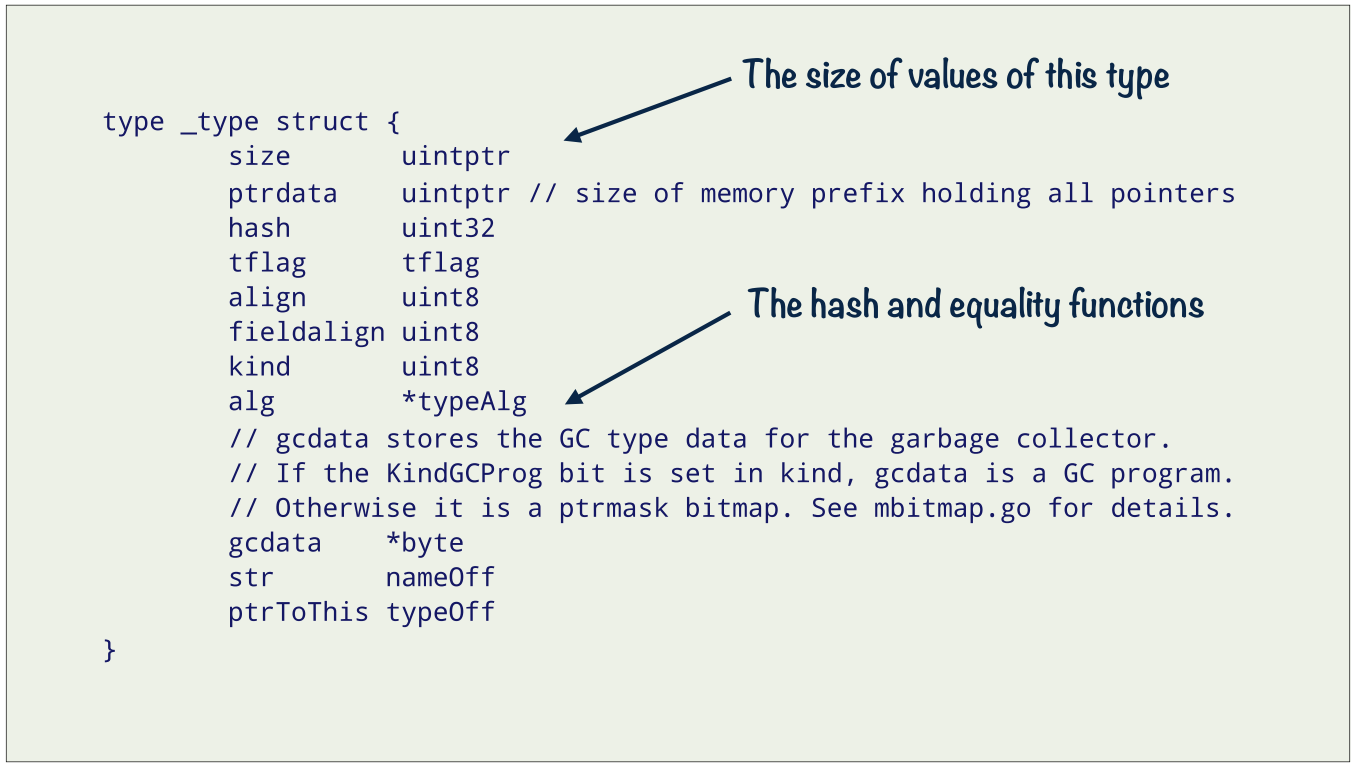
```
type maptype struct {
        typ           _type
        key           *_type
        elem          *_type
        bucket        *_type // internal type representing a hash bucket
        hmap          *_type // internal type representing a hmap
        keysize       uint8  // size of key slot
        indirectkey   bool   // store ptr to key instead of key itself
        valuesize     uint8  // size of value slot
        indirectvalue bool   // store ptr to value instead of value
itself
        bucketsize    uint16 // size of bucket
        reflexivekey  bool   // true if k==k for all keys
        needkeyupdate bool   // true if we need to update key on
overwrite
}
```

The maptype contains details about properties of this kind of map from key to elem.

It contains infomation about the key, and the elements.

maptype.key contains information about the pointer to the key we were passed.

We call these *type descriptors*



The size of values of this type

The hash and equality functions

```
type _type struct {
        size       uintptr
        ptrdata    uintptr // size of memory prefix holding all pointers
        hash       uint32
        tflag      tflag
        align      uint8
        fieldalign uint8
        kind       uint8
        alg        *typeAlg
        // gcdata stores the GC type data for the garbage collector.
        // If the KindGCProg bit is set in kind, gcdata is a GC program.
        // Otherwise it is a ptrmask bitmap. See mbitmap.go for details.
        gcdata    *byte
        str       nameOff
        ptrToThis typeOff
}
```

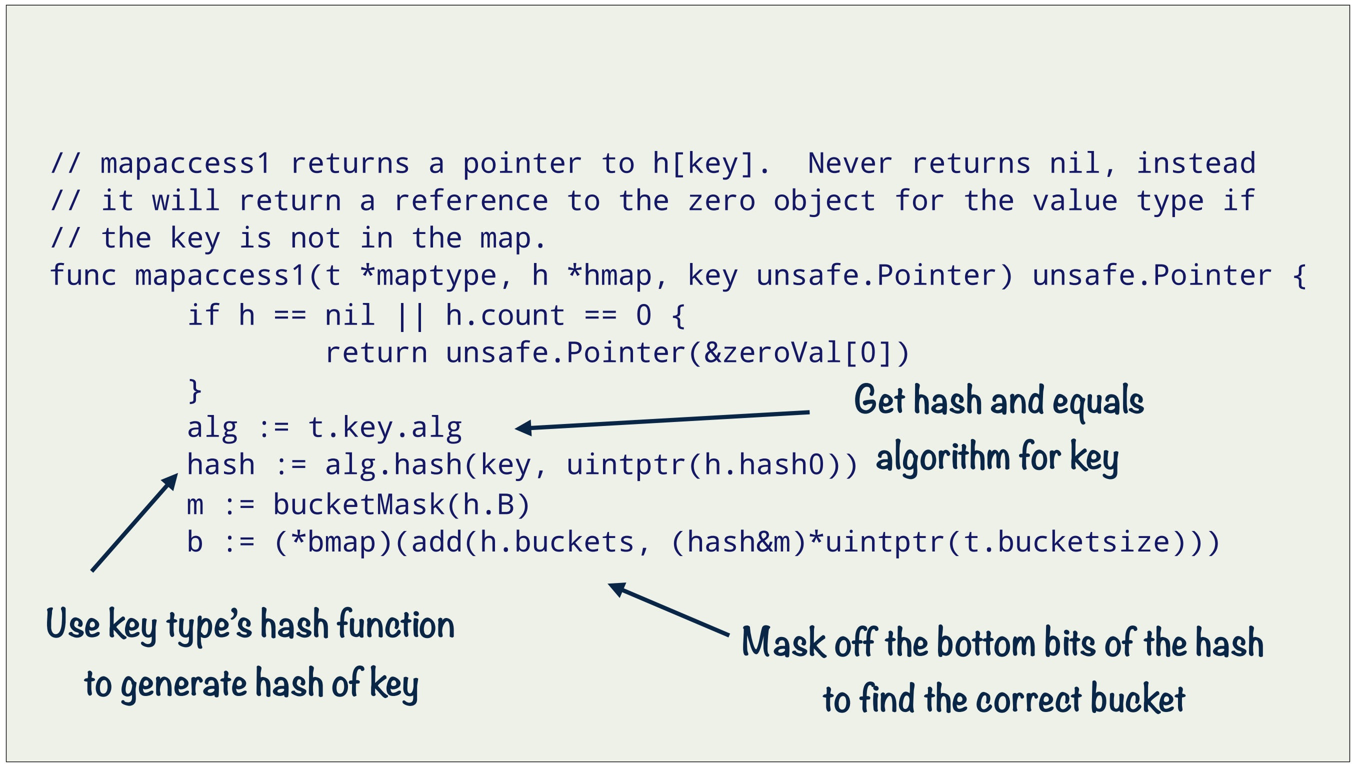
In the _type type, we have things like it’s size, which is important because we just have a pointer to the key value, but we need to know how large it is, what kind of a type it is; it is an integer, is it a struct, and so on.

We also need to know how to compare values of this type and how to hash values of that type, and that is what the _type.alg field is for.

Hash function for the key

```
type typeAlg struct {
        // function for hashing objects of this type
        // (ptr to object, seed) -> hash
        hash func(unsafe.Pointer, uintptr) uintptr
        // function for comparing objects of this type
        // (ptr to object A, ptr to object B) -> ==?
        equal func(unsafe.Pointer, unsafe.Pointer) bool
}
```

Equality function for the key



```go
// mapaccess1 returns a pointer to h[key].  Never returns nil, instead
// it will return a reference to the zero object for the value type if
// the key is not in the map.
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
        if h == nil || h.count == 0 {
                return unsafe.Pointer(&zeroVal[0])
        }
        alg := t.key.alg
        hash := alg.hash(key, uintptr(h.hash0))
        m := bucketMask(h.B)
        b := (*bmap)(add(h.buckets, (hash&m)*uintptr(t.bucketsize)))
```

Get hash and equals algorithm for key

Use key type's hash function to generate hash of key

Mask off the bottom bits of the hash to find the correct bucket

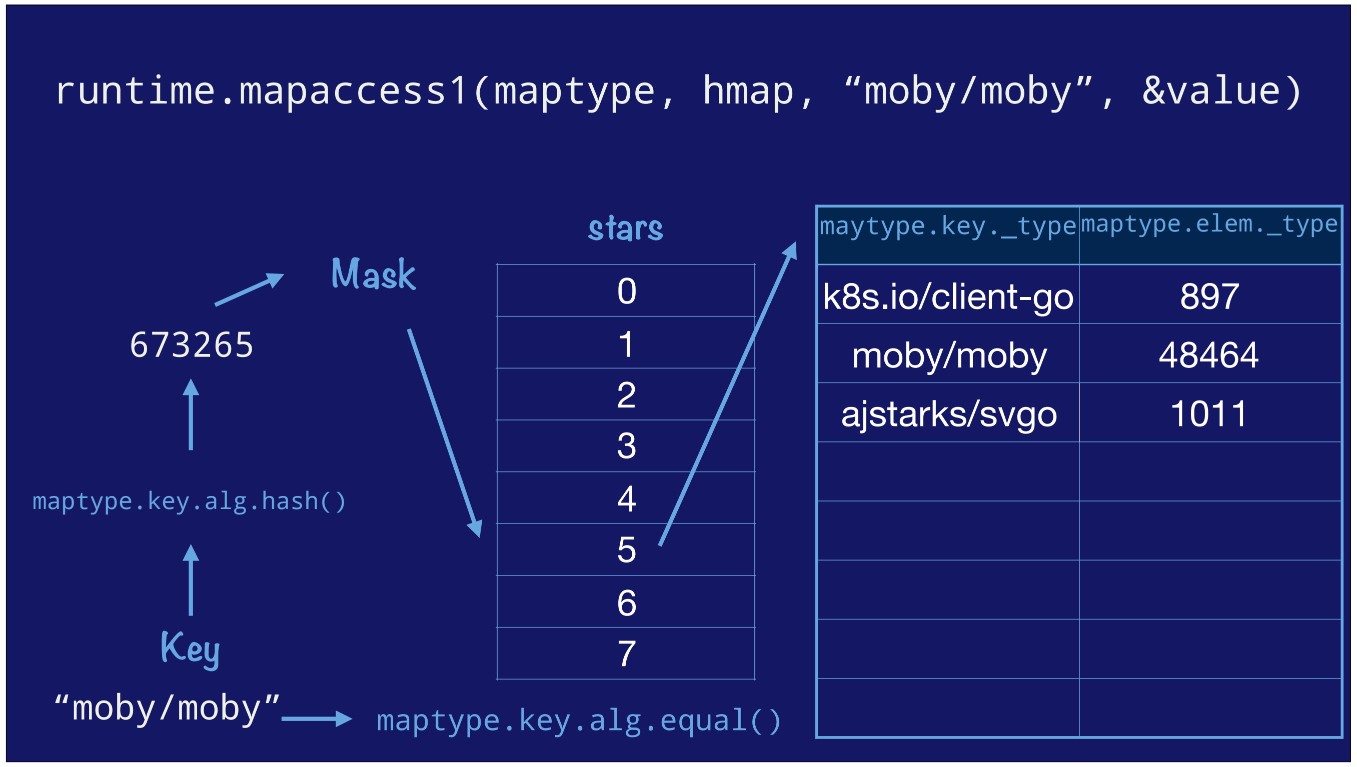
Now we're ready to look at map access 1

```
// mapaccess1 returns a pointer to h[key].  Never returns nil, instead
// it will return a reference to the zero object for the value type if
// the key is not in the map.
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
        if h == nil || h.count == 0 {
                return unsafe.Pointer(&zeroVal[0])
        }
        alg := t.key.alg
        hash := alg.hash(key, uintptr(h.hash0))
        m := bucketMask(h.B)
        b := (*bmap)(add(h.buckets, (hash&m)*uintptr(t.bucketsize)))
```

random seed to avoid hash collisions

One thing to note is the h.hash0 parameter passed into alg.hash. This is how the Go runtime avoids hash collisions. h.hash0 is a random seed generated when the map is created.

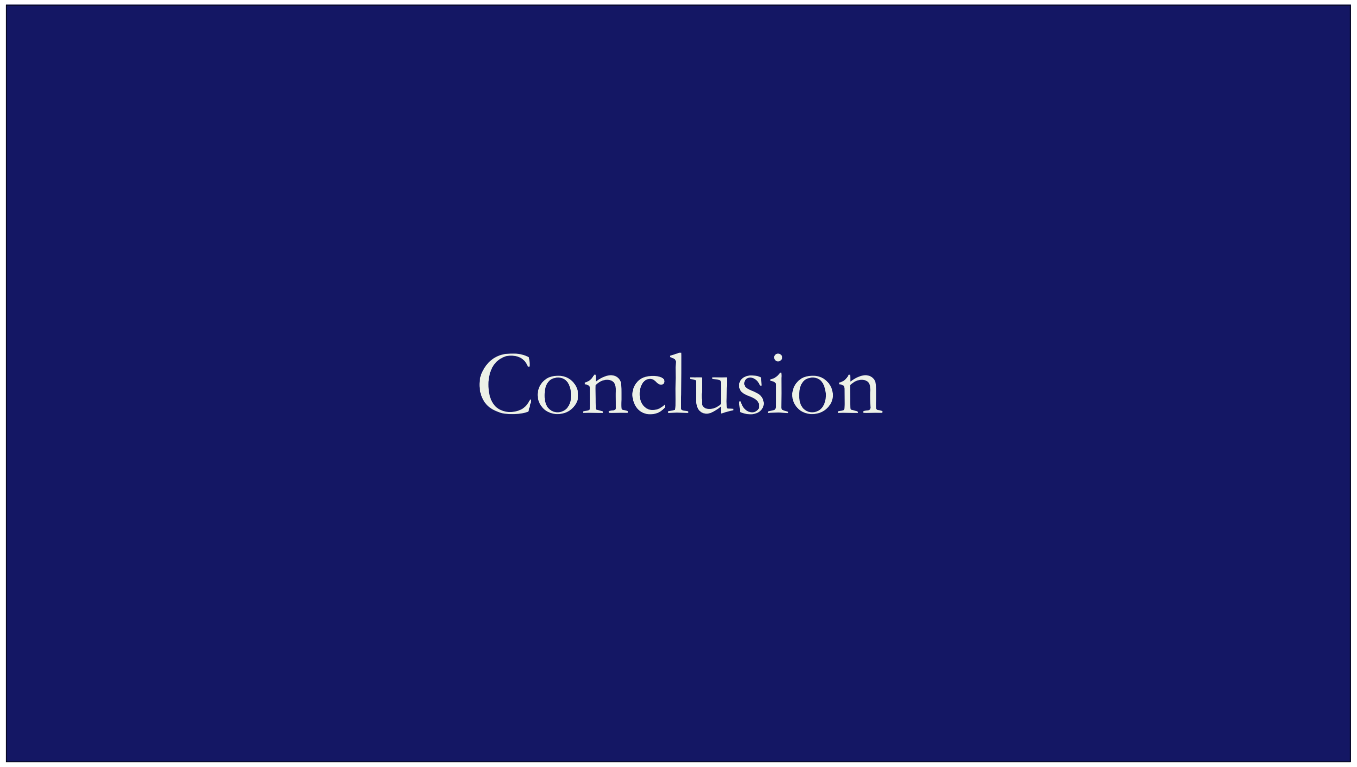
Anyone can read the Go source code, so they could come up with a set of values which, using the hash ago that go uses, all hash to the same bucket. The seed value adds an amount of “randomness” to the hash function, providing some protection against this attack.



```
runtime.mapaccess1(maptype, hmap, “moby/moby”, &value)
```

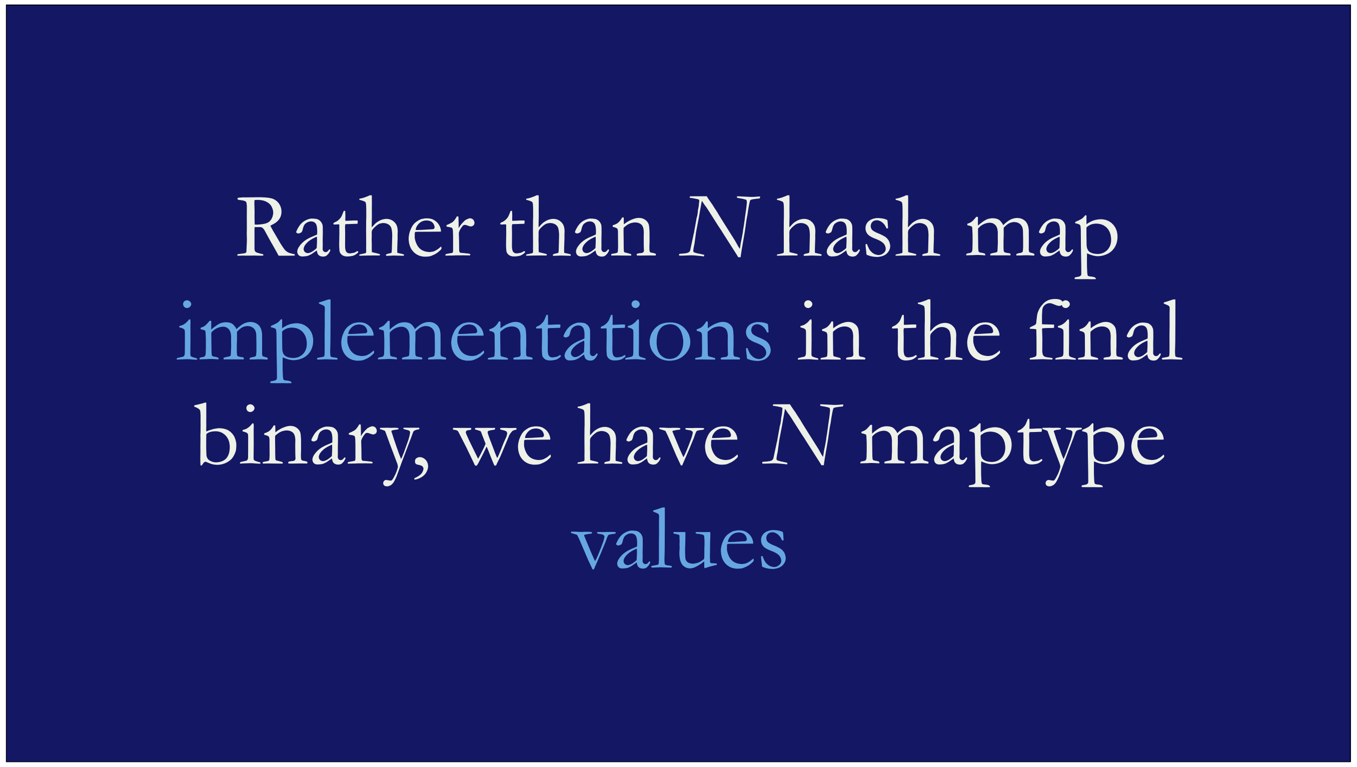
maytype.key._type	maptype.elem._type
k8s.io/client-go	897
moby/moby	48464
ajstarks/svgo	1011

For completeness lets look at how mapaccess works in go.



Conclusion

In conclusion, I hope this was informative for you, and that you enjoyed learning how Go uses a combination of compile time and run time to support fast hashmaps without boxing or code bloat.



Rather than *N* hash map implementations in the final binary, we have *N* maptype values

I was inspired to give this talk because I was impressed that Go’s map implementation is a compromise between C++’s and Java’s, taking most of the good without having to accomodate most of the bad.

Unlike Java, you can use scalar values like characters and integers without the overhead of boxing.

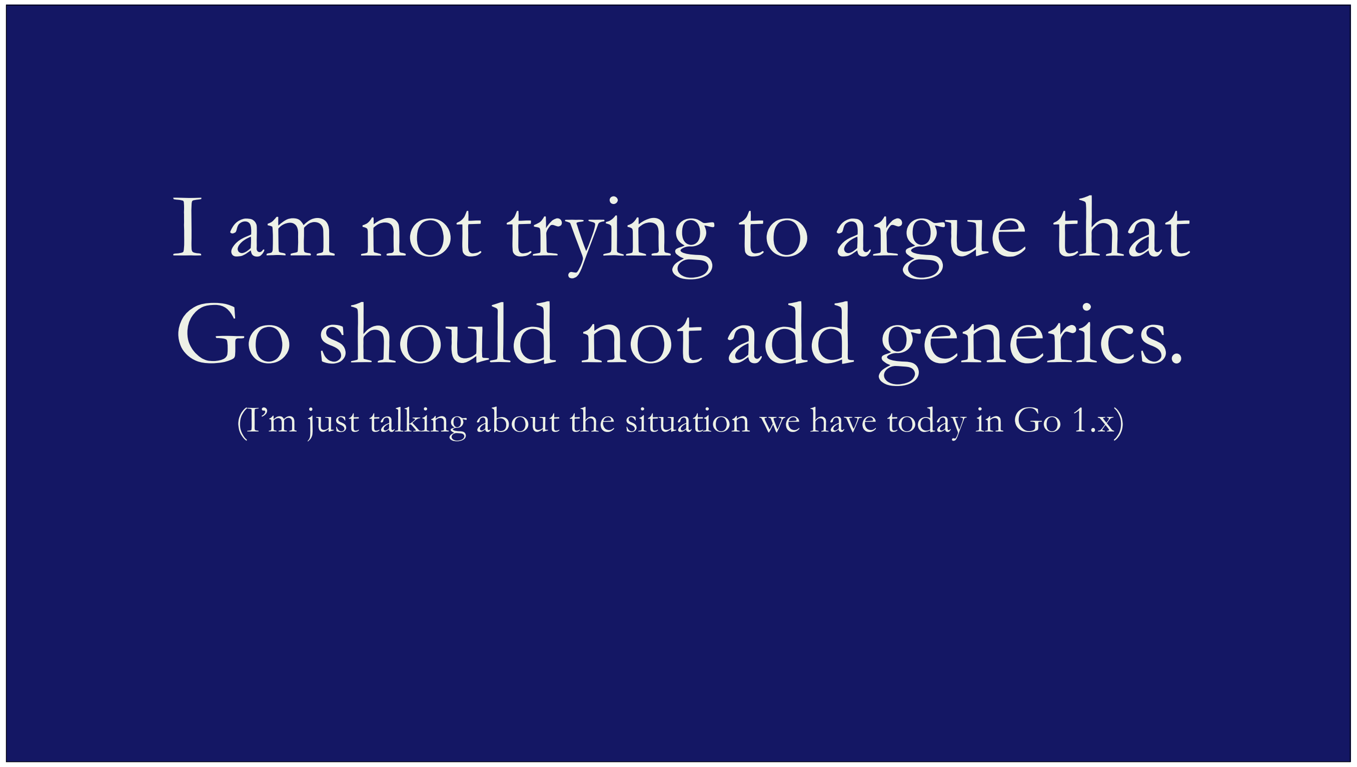
And unlike c++, rather than one copy of the code for a map in the finally binary, there is one maptype value per map.

There is only one copy of the map code in the runtime, and while it cannot be optimised to the degree that the C++ implementation can, Go maps are not slow.

```
name                      time/op
MapAssign/Int32/256-4     20.9ns
MapAssign/Int32/65536-4   40.7ns
MapAssign/Int64/256-4     18.5ns
MapAssign/Int64/65536-4   39.5ns
MapAssign/Str/256-4       23.4ns
MapAssign/Str/65536-4     53.1ns
MapDelete/Int32/100-4     36.1ns
MapDelete/Int32/1000-4    29.6ns
MapDelete/Int32/10000-4   32.6ns
MapDelete/Int64/100-4     36.0ns
MapDelete/Int64/1000-4    30.9ns
MapDelete/Int64/10000-4   34.2ns
MapDelete/Str/100-4       30.3ns
MapDelete/Str/1000-4      32.4ns
MapDelete/Str/10000-4     44.4ns
```

2015 MacBook Air 11”

Taken on a 2015 MacBook in 2017. Its probably a bit better today



I am not trying to argue that Go should not add generics.

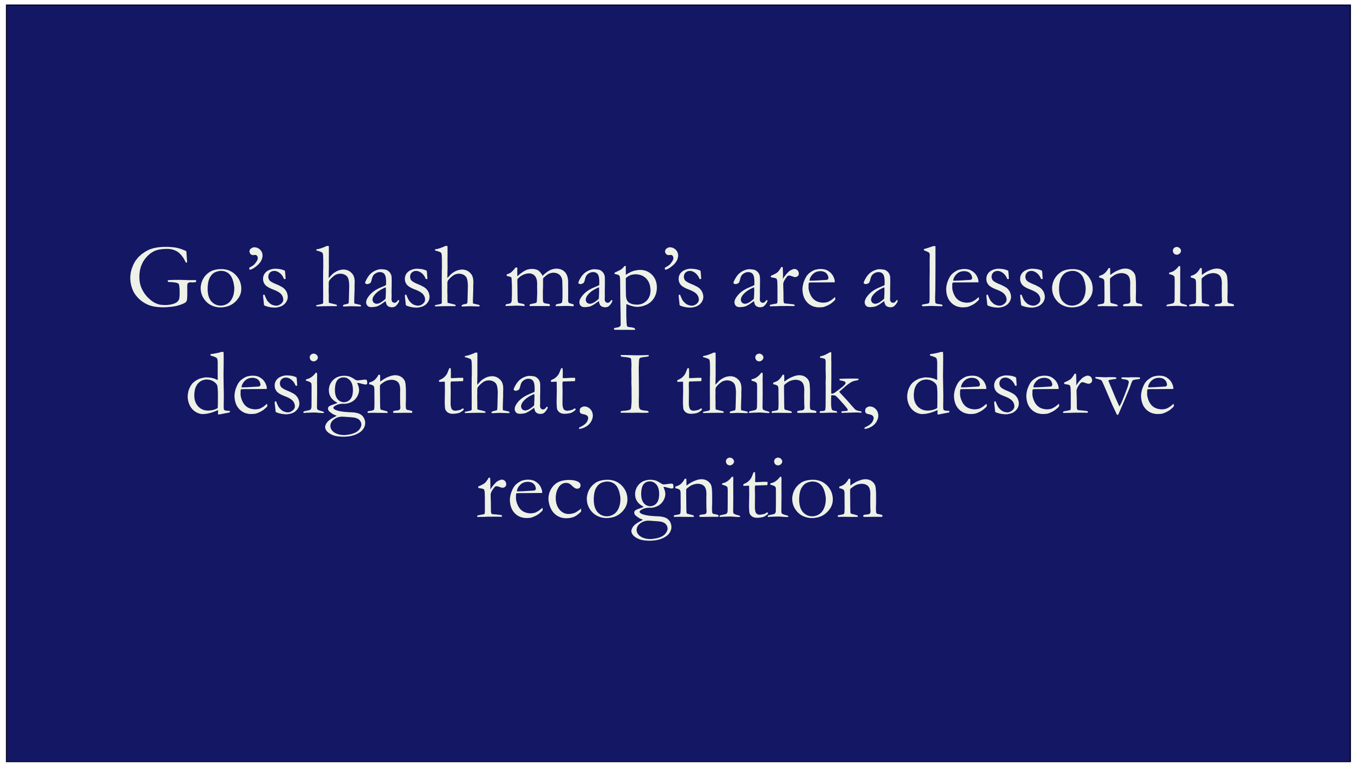
(I'm just talking about the situation we have today in Go 1.x)

Now I want to be clear that I am not trying to tell you that Go should not have generics.

My goal today was to describe the situation we have today in Go 1 and how the map type in Go works under the hood

However, without generics, Go has avoided many of the pitfalls of other implementations that use a “generic” or “templated” method

But, even though we don’t have genetics today, the go map implementation we have is very fast and provides most of the benefits of templated types, without the downsides of code generation and compile time bloat.



Go’s hash map’s are a lesson in design that, I think, deserve recognition

And so I see that as a lesson in design that deserves recognition.