

Forum:[Hack Forums](#)

Thread: [A Challenge for the bored C++'ers](#)

saltycracker - April 30, 2020, 10:00 am

Here's a good challenge for a keen C++'er.

Create a dictionary in C++(keys are string's, and values are int's) but you can only use functions, function closures, and lambda functions as the dictionary.

Good luck and have fun!

A hint right way. Here's the header files I used in my solution:

Code `#include <iostream>`

`#include <string>`

`#include <functional>`

Hint number 2.

I don't expect anything fancy.

Just create a data store that you can add string keys and int values that you can later retrieve.

example ->

you should have a add function to add key value pairs to your dictionary.

You should have a find function to lookup keys and return values from your dictionary.

jumpprop - April 30, 2020, 3:15 pm

Hi, this seems like a fun challenge, but I'm a bit confused by the wording of the problem.

Would you mind clarifying what you mean when you say "but you can only use functions, function closures, and lambda functions as the dictionary."?

Thanks

saltycracker - April 30, 2020, 9:30 pm

(Apr 30, 2020 03:15 PM)jumpprop Wrote: [\[CITE\]](#) Hi, this seems like a fun challenge, but I'm a bit confused by the wording of the problem.

Would you mind clarifying what you mean when you say "but you can only use functions, function closures, and lambda functions as the dictionary."?

Thanks

You will store the key and value pairs using function closures and lambda functions.

Please note: The solution isn't complicated or large. My solution weights in at 34 lines of code.

Dostoevsky - April 30, 2020, 10:59 pm

PHP Code

```
<?php
using map = std::function<std::optional<int>(const std::string &);>;
```

```
map remap(const map &m, const std::string &k, int v) {
    return [=](const std::string &key) -> std::optional<int> {
```

```

    if (key == k)
        return {v};

    return m(key);
};
}

template<typename T>
std::ostream &operator<<(std::ostream &o, const std::optional<T> &opt) {
    if (opt.has_value())
        return o << opt.value();

    return o << "nothing";
}

int main() {
    auto empty = [] (const std::string &k) -> std::optional<int> { return {}; };
    auto map = remap(empty, "one", 1);
    map = remap(map, "two", 2);

    std::cout << map("one") << ' ' << map("two") << ' ' << map("fun") << '\n';
}

```

This isn't a challenge if you've ever formalised heaps in Lean, Coq, or Agda, etc. where this is like the simplest way to do a purely functional map (the heap is defined as a mapping from keys to values and to "set a key to a value" is just to create a shallower if-statement in a recursive decision tree using higher order functions).

saltycracker - April 30, 2020, 11:25 pm

(Apr 30, 2020 10:59 PM)Dostoevsky Wrote: [\[CITE\]](#) PHP Code
 <?php
 using map = std::function<std::optional<int>(const std::string &)>;

```

map remap(const map &m, const std::string &k, int v) {
    return [=](const std::string &key) -> std::optional<int> {
        if (key == k)
            return {v};

        return m(key);
    };
}

template<typename T>
std::ostream &operator<<(std::ostream &o, const std::optional<T> &opt) {
    if (opt.has_value())
        return o << opt.value();

    return o << "nothing";
}

int main() {
    auto empty = [] (const std::string &k) -> std::optional<int> { return {}; };
    auto map = remap(empty, "one", 1);

```

```
map = remap(map, "two", 2);

std::cout << map("one") << ' ' << map("two") << ' ' << map("fun") << "\n";
}
```

This isn't a challenge if you've ever formalised heaps in Lean, Coq, or Agda, etc. where this is like the simplest way to do a purely functional map (the heap is defined as a mapping from keys to values and to "set a key to a value" is just to create a shallower if-statement in a recursive decision tree using higher order functions).

I didn't use the optional type for my returned values.. I just used 0 as an invalid or end of search flag.

```
Code #include <iostream>
#include <string>
#include <functional>
```

```
using dict = std::function <int (std::string)>;
```

```
int empty (std::string key) {
    return 0;
}
```

```
int find(dict f, std::string key) {
    return f(key);
}
```

```
const dict add(const dict f, std::string key, int value){
    return
        [key, value, f] (std::string k) -> int
        {
            if (k == key) {
                return value;
            }else {
                return f(k);
            }
        };
}
```

```
int main () {
    auto ans = add(add(add(empty, "Salty", 4143), "Emily", 1234), "Angela", 4321);

    std::cout << find(ans, "Salty") << std::endl;
    std::cout << find(ans, "Emily") << std::endl;
    std::cout << find(ans, "Angela") << std::endl;
    std::cout << find(ans, "NoName") << std::endl;
}
```

Dostoevsky - April 30, 2020, 11:45 pm

(Apr 30, 2020 11:25 PM)saltcracker Wrote: [\[CITE\]](#) I didn't use the optional type for my returned values.. I just used 0 as an invalid or end of search flag.

```
Code #include <iostream>
#include <string>
```

```
#include <functional>
```

```
using dict = std::function <int (std::string)>;
```

```
int empty (std::string key) {  
    return 0;  
}
```

```
int find(dict f, std::string key) {  
    return f(key);  
}
```

```
const dict add(const dict f, std::string key, int value){  
    return  
        [key, value, f] (std::string k) -> int  
        {  
            if (k == key) {  
                return value;  
            }else {  
                return f(k);  
            }  
        };  
}
```

```
int main () {  
    auto ans = add(add(add(empty, "Salty", 4143), "Emily", 1234), "Angela", 4321);  
  
    std::cout << find(ans, "Salty") << std::endl;  
    std::cout << find(ans, "Emily") << std::endl;  
    std::cout << find(ans, "Angela") << std::endl;  
    std::cout << find(ans, "NoName") << std::endl;  
}
```

I used optional because, of course, the next step would be to generalise key and value types to produce a templated version.

saltycracker - May 1, 2020, 12:15 am

(Apr 30, 2020 11:45 PM)Dostoevsky Wrote: [\[CITE\]](#) I used optional because, of course, the next step would be to generalise key and value types to produce a templated version.

Yeah! Something like.

Code type ('a, 'b) dict = 'a -> 'b option

Dostoevsky - May 1, 2020, 12:26 am

(May 1, 2020 12:15 AM)saltycracker Wrote: [\[CITE\]](#) Yeah! Something like.

Code type ('a, 'b) dict = 'a -> 'b option

Perhaps a bit more like:

Code Eq a => a -> Maybe b(:

saltycracker - May 1, 2020, 12:50 am

(May 1, 2020 12:26 AM)Dostoevsky Wrote: [\[CITE\]](#) Perhaps a bit more like:

Code Eq a => a -> Maybe b(:

Different language... I'm more comfortable with OCaml

Code type ('a, 'b) dict = 'a -> 'b option

```
let empty _ = None
```

```
let find d key = d key
```

```
let add d key value =  
  fun k ->  
    match (k = key) with  
    | true -> Some value  
    | false -> d k
```

```
let d =  
  List.fold_left  
    (fun a (k, v) -> add a k v)  
    empty  
    [("Salty", 4143); ("Emily", 1234); ("Angela", 4321)]
```

```
let () = Option.iter (fun v -> Printf.printf "Found: %d\n" v) (find d "Emily")
```

Dostoevsky - May 1, 2020, 12:54 am

(May 1, 2020 12:50 AM)saltycracker Wrote: [\[CITE\]](#) Different language... I'm more comfortable with OCaml

Code type ('a, 'b) dict = 'a -> 'b option

```
let empty _ = None
```

```
let find d key = d key
```

```
let add d key value =  
  fun k ->  
    match (k = key) with  
    | true -> Some value  
    | false -> d k
```

```
let d =  
  List.fold_left  
    (fun a (k, v) -> add a k v)  
    empty  
    [("Salty", 4143); ("Emily", 1234); ("Angela", 4321)]
```

```
let () = Option.iter (fun v -> Printf.printf "Found: %d\n" v) (find d "Emily")
```

Yeah, I like OCaml. Just the lack of typeclasses, curried constructors (at least without Jane street extensions), etc. is a deal-breaker for quite a lot of things for me (although, admittedly, it has some of the best tooling for other stuff; menhir).

saltycracker - May 1, 2020, 1:03 am

(May 1, 2020 12:54 AM)Dostoevsky Wrote: [\[CITE\]](#) Yeah, I like OCaml. Just the lack of typeclasses, curried constructors (at least without Jane street extensions), etc. is a deal-breaker for quite a lot of things for me (although, admittedly, it has some of the best tooling for other stuff; menhir).

I'm just glad that OCaml takes a practical approach to functional programming.. I don't think I'd like OCaml if it embraced monads. I like an easy escape hatch out of the functional paradigm to address the world.

Dostoevsky - May 1, 2020, 1:21 am

(May 1, 2020 01:03 AM)saltycracker Wrote: [\[CITE\]](#) I'm just glad that OCaml takes a practical approach to functional programming.. I don't think I'd like OCaml if it embraced monads. I like an easy escape hatch out of the functional paradigm to address the world.

Yeah, it's a "get sh*t done" functional language. My favourite OCaml project is Coq, the theorem prover (that's the workhorse theorem prover).