⚠ This page is not current. It is an archive from Winter Quarter 2022.

Section 6. Class Design and Hashing

Section materials curated by Neel Kishnani, drawing upon materials from previous quarters.

This week’s section exercises explore the ins and outs of content from weeks 6 and 7, focusing on class design and hash functions!

Each week, we will also be releasing a Qt Creator project containing starter code and testing infrastructure for that week's section problems. When a problem name is followed by the name of a `.cpp` file, that means you can practice writing the code for that problem in the named file of the Qt Creator project. Here is the zip of the section starter code:

📦 Starter code

Problem One: Shrinking the Stack

In class, we implemented the **`OurStack`** class, which represents a stack of integers. The class definition is shown here:

```
class OurStack {
public:
    OurStack();
    ~OurStack();

    void push(int value);
    int pop();
    int peek() const;

    int size() const;
    bool isEmpty() const;

private:
  void grow();

  int* elems;
  int logicalSize;
  int allocatedSize;
};
```

The implementation is on the next page, just as a refresher. Here’s the .cpp file:

```
#include "OurStack.h"
#include "error.h"
const int kInitialSize = 4;

OurStack::OurStack() {
  elems = new int[kInitialSize];
  logicalSize = 0;
  allocatedSize = kInitialSize;
}

OurStack::~OurStack() {
  delete[] elems;
}

void OurStack::push(int value) {
  if (logicalSize == allocatedSize) {
    grow();
  }
  elems[logicalSize] = value;
  logicalSize++;
}

int OurStack::peek() const {
  if (isEmpty()) {
    error("What is the sound of one hand clapping?");
  }
  return elems[logicalSize - 1];
}

int OurStack::pop() {
  int result = peek();
  logicalSize--; return result;
}

int OurStack::size() const {
  return logicalSize;
}

bool OurStack::isEmpty() const {
  return size() == 0;
}

void OurStack::grow() {
  int* newArr = new int[2 * allocatedSize];
  for (int i = 0; i < size(); i++) {
        newArr[i] = elems[i];
    }
  delete[] elems;
  elems = newArr;
  allocatedSize *= 2;
}
```

This problem consists of some questions about the OurStack type:

i. What is the meaning of the term “logical size?” How does it compare to the term “allocated size?” What’s the relationship between the two?

Solution

The logical size of the stack is the actual number of elements stored in the stack, whereas the allocated size is the length of the **`elems`** array. The logical length can be any number between 0 (inclusive) and **`allocatedSize`** (exclusive) depending on how many pushes and pops have been made.

ii. What is the `OurStack()` function? Why is each line in that function necessary?

Solution

This is the constructor. It's responsible for setting all the variables in the OurStack class to a reasonable set of defaults. The lines here set up the actual memory where the elements will be stored, configure the initial logical length (0 elements stored), and the initial allocated length (which is the default initial size given by the constant.)

iii*. What is the `~OurStack()` function? Why is each line in that function necessary? Why isn't there any code in there involving the logicalSize or allocatedSize data members?

Solution

This is the destructor. Its job is to deallocate any extra memory or resources that were allocated by the **`OurStack`** class and not otherwise automatically reclaimed. Here, we deallocate the memory we allocated earlier by using **`delete[]`**. We don't need to do anything to **`logicalSize`** or allocatedSize because there were no dynamic allocations performed to get space for them. Generally speaking, you only need to explicitly clean up resources that you explicitly allocated.

iv. In the `OurStack::grow()` function, one of the lines is `delete[] elems`, and in the next line we immediately write `elems = newArr;`. Why is it safe to do this? Doesn't deleting `elems` make it unusable?

Solution

The statement **`delete[] elems;`** means "destroy the memory *pointed at by* **`elems`**" rather than "destroy the **`elems`** *variable*." As a result, after the first line executes, elems is still a perfectly safe variable to reassign.

v. The `OurStack::pop()` function doesn't seem to have any error-checking code in it. What happens if you try to pop off an empty stack?

Solution

Notice that **`OurStack::pop`** calls **`OurStack::peek`**, which in turn has its own error-checking. As a result, trying to pop off an empty stack will trigger the custom error message from the **`OurStack::peek`** member function.

vi. What is the significance of placing the `OurStack::grow` function in the private section of the class? What does that mean? Why didn't we mark it public?

Solution

This is a private member function. This is a member function that's used as a part of the implementation of the class rather than the interface. Any clients of **OurStack** can't access this function, which is a Good Thing because we don't want rando's coming along and increasing our stack's capacity unnecessarily.

Right now, our stack doubles its allocated length whenever it runs out of space and needs to grow. The problem with this setup is that if we push a huge number of elements into our stack and then pop them all off, we'll still have a ton of memory used because we never shrink the array when it's mostly unused.

vii. Explain why it would not be a good idea to cut the array size in half whenever fewer than half the elements are in use.

Solution

Imagine we have a stack with logical size $n - 1$ and allocated size n. If we perform two pushes, we'll double our stack's allocated size to 2 n, which takes time O(n), and increase the logical size to n+1. If we now do two pops, we'll drop the logical size down to n-1, which is less than half the allocated size. If we now shrink the allocated size by half down to n, we'll have to do O(n) work transferring elements over, ending with a stack with logical size $n - 1$ and allocated size n. Overall, we've done two pushes and two pops, we're right back where we've started, and we've done O(n) work. That's a lot of work for very little payoff!

---

viii. Update the code for **OurStack** so that whenever the logical length drops below one quarter of the allocated length, the array is reallocated at half its former length. As an edge case, ensure that the allocated length is always at least equal to the initial allocated capacity. This setup maintains the (amortized) *O(1)* cost of each insertion and deletion and ensures that the memory usage is always *O(n)*.

Solution

There are many ways we can do this. We're going to do this by replacing the **OurStack::grow** function with a new **OurStack::setCapacity** function that lets us say what the new allocated size will be. Here's the updated class definition:

```
class OurStack {
public:
    OurStack();
    ~OurStack();

    void push(int value);
    int pop();
    int peek() const;

    int size() const;
    bool isEmpty() const;
private:
  void setCapacity(int capacity);

  int* elems;
  int logicalSize;
  int allocatedSize;
};
```

Here's the updated functions from the .cpp file:

```
void OurStack::push(int value) {
  if (logicalSize == allocatedSize) {
    setCapacity(allocatedSize * 2);
  }
  elems[logicalSize] = value;
  logicalSize++;
}
int OurStack::pop() {
  int result = peek();
  logicalSize--;
  if (logicalSize < allocatedSize / 4) {
    setCapacity(allocatedSize / 2);
  }
  return result;
}
void OurStack::setCapacity(int capacity) {
  /* If we're asking to reduce our size below the initial starting size,
   * instead leave the array at its current size.
   */
  if (capacity < kInitialSize) return;

  int* newArr = new int[capacity];
  for (int i = 0; i < size(); i++) {
    newArr[i] = elems[i];
  }
  delete[] elems;
  elems = newArr;
  allocatedSize = capacity;
}
```

Problem Two: Hash Functions

In CS106B, we'll use hash functions without spending too much time talking about their internal workings. (Building a good hash function is a challenging endeavor!) To give you a sense about why this is, we'd like you to investigate four different possible hash functions.

Each of the functions shown below is designed to take as input a string and then output a hash code in the range from 0 to 9, inclusive. For each proposed hash function, tell us whether it's

1. not even a valid hash function;
2. a valid hash function, but not a very good one; or
3. a very good hash function.

Of course, you should be sure to justify your answers.

```
int hashFunction1(const string& str) {
  return randomInteger(0, 9); // a value between 0 and 9, inclusive
}
int hashFunction2(const string& str) {
  return 0; // Who cares about str, anyway?
}
```

Solution

```
int hashFunction1(const string& str) {
  return randomInteger(0, 9); // a value between 0 and 9, inclusive
}
```

This is not a valid hash function. One of the main requirements for a hash function is that if you hash the same input multiple times, you always get the same output. However, this hash function doesn't do this, since hashing any string multiple times might produce different answers.

At a more intuitive level – imagine you're using a hash function to distribute items into drawers. The idea is that you'd compute the hash code for the item, then put the item in the given drawer. The advantage of doing this is that if you want to see whether you have that item, you just need to look in the drawer it hashes to. But if you get different hash codes back each time, you aren't guaranteed that you'll be able to find the item when you hash it, since you'll be sent to a random drawer each time!

```
int hashFunction2(const string& str) {
  return 0; // Who cares about str, anyway?
}
```

This hash code passes the first test of a hash function – if you hash the same input many times, you will get the same output. However, it absolutely fails the second test – since everything gets mapped to slot zero, this hash function doesn't distribute items nicely over the range, and in fact hashes of wildly different strings will always end up the same! Returning to the analogy above – this hash function corresponds to the strategy of "put everything in the same drawer." That'll work, but your drawer will get really full!

These next three functions use the fact that each char has an associated integer value in C++. For example, on most systems the character `'A'` has value 65, the character `'3'` has value 51, the character `'!'` has value 33, and the character `'~'` has value 126. We can use these numeric values to design hash functions for strings.

```
int hashFunction3(const string& str) {
  return str[0] / 10;
}
int hashFunction4(const string& str) {
  return str[0] % 10;
}
int hashFunction5(const string& str) {
  int result = 0;
  for (char ch: str) {
    result += ch; // Again, uses the numeric value of ch
  }
  /* Numeric values for characters can be positive or negative. */
  if (result < 0) {
    result = 0;
  }
  if (result >= 10) {
    result = 9;
  }
  return result;
}
```

Solution

```
int hashFunction3(const string& str) {
  return str[0] / 10;
}
```

There are two problems with this hash function. First, if the input string is empty, it reads off the end of the string – oops! That's no good. (It turns out that in C++, reading one step off the end of the string is technically a well-defined operation, but I'd wager that most C++ programmers don't know this and at a bare minimum it's a weird thing to do.)

But for now, let's ignore that. This hash function does indeed compute hash codes consistently (if you hash the same string many times, you'll always get the same result). However, it does not produce hash codes that are guaranteed to be in the range from 0 to 9, since hashing the string "~" would output 12. Therefore, this is not a legal hash function.

```
int hashFunction4(const string& str) {
  return str[0] % 10;
}
```

Ignoring the issue raised above about how str[0] might be an out-of-bounds read, this hash function is deterministic (hashing the same string many times always produces the same output), and the outputs are always in the range from 0 to 9, inclusive. In fact, taking an integer and modding it by the number n will always give you a value in the range from 0 to n – 1, inclusive.

But is this a *good* hash function? I'd say no. Good hash functions should have the property that tiny changes to the inputs result in large changes to the output. Here, taking a string and appending a new character to it, or changing any character but the first, or deleting any character but the first will not change the hash code. You'll get a huge number of collisions with this hash function.

```
int hashFunction5(const string& str) {
  int result = 0;
  for (char ch: str) {
    result += ch; // Again, uses the numeric value of ch
  }
  /* Numeric values for characters can be positive or negative. */
  if (result < 0) {
    result = 0;
  }
  if (result >= 10) {
    result = 9;
  }
  return result;
}
```

This hash function is deterministic (hashing the same string always produces the same result), and the way it's structured guarantees that the range of values produced is always between 0 and 9, inclusive. However, there are two major problems with this hash function:

1. The way that this hash function produces values in the range from 0 to 9 is biased toward producing the outputs 0 and 9. Specifically, any string made of purely positive-value characters (basically, most English text) will produce a result greater than 9, which clamps to 9. Similarly, any string made of purely negative-value characters (whether these exist depends on your OS) will clamp to 0. So in that sense, the values aren't nicely-distributed across the whole range, and you'd expect to get lots of collisions at 0 and 9.
2. The hash function doesn't care about the order of the characters in the string. The strings "tarragon" and "arrogant" are *anagrams* of one another – they're made of the same characters in a different order – and so they'll have the same hash code. The same is true of "senator," "treason,"

and “atoners.” We’d like it to be the case that if we make small changes to a string (say, swap- ping the order of the letters) that we get wildly different outputs, but that’s not the case here. We hope this little foray into the design of hash functions gives you a sense as to why it’s hard to design good hash functions and why this is something we’re going to leave to the experts. 😛

Problem Three: Salting and Hashing

In lecture, we talked about how when storing passwords, it’s common to not store the passwords themselves, but rather the hash code of the password using some hash function. This ensures that if someone manages to steal the password database, they can’t immediately read off all the passwords stored in the database.

This approach – storing the hashes of the passwords rather than the passwords themselves – is much better than just storing the passwords themselves, but it’s not what’s done in practice because it doesn’t work well if the users have weak passwords.

Specifically, suppose that you have a list of the 100,000 most commonly used passwords, which isn’t too hard to find with a bit of Googling. You also have, probably nefariously, stolen the password file from a website, which is represented as a `Map<string, int>` mapping each user name to the hash of their password. You could then do the following to figure out the passwords for anyone using weak accounts:

1. Compute the hash codes for the 100,000 commonly-used passwords that you know of.
2. Look at the `Map<string, int>` mapping website users to the hashes of their passwords. For each hash code that matches one of the hashes you’ve computed above, output their password.

Write a function

```
Map<string, string> breakWeakPasswords(const Map<string, int>& stolenPasswordFile,
                                       const Set<string>& knownWeakPasswords,
                                       HashFunction<string> hashFn);
```

that uses the above approach to figure out the passwords of each user who has a weak password, returning a `Map` associating each such user with their password. The final parameter, `hashFn`, represents the hash function used to compute password hashes. Then, answer the following question: if there are *n* users on the website and there are *m* weak passwords, what is the runtime of the code that you wrote, assuming hash codes can be computed in time O(1)?

In practice, it’s most common to use a strategy called **salting and hashing**, which works as follows, to prevent people from recovering passwords in the way you just did. We store for each user two pieces of information:

- a randomly-chosen string called the **salt**, and
- the hash code of the string `salt + password`, where password is the user’s password.

It’s important that the salt is chosen randomly for each user, rather than having a fixed salt used for all passwords.

First, explain why the introduction of a per-user salt means that you can’t break passwords using the code that you wrote above. Then, answer the following question: suppose you had the master password file for a website, including the salted, hashed passwords, and you wanted to find all users who have weak passwords. If there are *n* users and *m* weak passwords, how long would it take to find all users who have weak passwords? Represent your answer in big-O notation.

(A note on this problem: we’re not teaching this to you because we want you to do Nefarious Things like this. Rather, we wanted you to see what Roguish Scalawags could do to compromise a password system and to see why we take the protective steps that we take to stop them from doing so.)

Solution

Here's one possible implementation of the function:

```
Map<string, string>
breakWeakPasswords(const Map<string, int>& stolenPasswordFile,
const Set<string>& knownWeakPasswords,
                   HashFunction<string> hashFn) {
  /* Build a map from hash codes to weak passwords. This takes time O(m). */
  Map<int, string> weakHashes;
  for (string password: knownWeakPasswords) {
      weakHashes[hashFn(password)] = password;
  }
  /* A user has a bad password if the hash is known. We can very quickly check
   * this because hash table lookups are so fast. This takes time O(n).
   */
   Map<string, string> result;
   for (string username: stolenPasswordFile) {
     int hashedPassword = stolenPasswordFile[username];
     if (weakHashes.containsKey(hashedPassword)) {
          result[username] = weakHashes[hashedPassword];
      }
    }
  return result;
}
```

This code takes time O($m + n$), which means that it's a very efficient way to find everyone with a weak password. Think about how this function scales and what has to happen for its output to double.

The above attack works because we can precompute the hashes of common passwords fairly quickly. If we add in a per-user salt, we can't do this any more. For example, suppose someone has the very weak password password. If they have a random salt, it's as though their password really was some long random string of characters followed by password, and the hash code of that string is likely to be completely different than that of the hash code for password itself. We therefore can't precompute the hash codes of all the weak passwords and then see if anyone has those hash codes, because the hash codes stored per person will be quite different from the hash codes of the raw passwords.

We could still go one person at a time and try hashing all the weak passwords along with the salt to see if that matches the stored hash for the person, but that's going to be much slower than before. We have to now do O(m) work for each of the O(n) users for a total runtime of O(mn). If m is, say, 100,000 and n is, say, 100,000,000, it's going to take us a long time to find everyone with weak passwords, long enough for the company to announce that there's been a breach and urge everyone to change passwords.

Website programming by Julie Zelenski • Styles adapted from Chris Piech • This page last updated 2022-Feb-17