Section 5. Class Design and Dynamic Memory Allocation

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

Problem One: Random Bag Grab Bag

The very first container class we implemented was the **random bag**, which supported two operations:

- **add**, which adds an element to the random bag, and
- **removeRandom**, which chooses a random element from the bag and returns it.

Below is the code for the **RandomBag** class. First, **RandomBag.h**:

```
#pragma once

#include "vector.h"

class RandomBag {
public:
    void add(int value);
    int removeRandom();

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

private:
    Vector<int> elems;
};
```

Next, **RandomBag.cpp**:

```
#include "RandomBag.h"
#include "random.h"

int RandomBag::size() const {
    return elems.size();
}

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

void RandomBag::add(int value) {
    elems += value;
}

int RandomBag::removeRandom() {
    if (isEmpty()) {
        error("That which is not cannot be!");
    }

    int index = randomInteger(0, size() - 1);
    int result = elems[index];

    elems.remove(index);
    return result;
}
```

Let’s begin by reviewing some aspects of this code.

i. What do the **`public`** and **`private`** keywords mean in **`RandomBag.h`**?

Solution

The **`public` keyword** indicates that the member functions listed underneath it are publicly accessible by anyone using the **RandomBag** class. This essentially means that they form the public in-terface for the class.

The **`private`** keyword indicates that the data members listed underneath it are private and only accessible by the class itself. This means that those data members are part of the private implementation of the class and aren’t something that clients should be touching.

ii. What does the **`::`** notation mean in C++?

Solution

It’s the **scope resolution operator**. It’s used to indicate what logical part of the program a given name belongs to. The case we’ll primarily see it used is in the context of defining member func-tions in a .cpp file, where we need to indicate that the functions we’re implementing are actually member functions of a class, not freestanding functions.

iii. What does the **`const`** keyword that appears in the declarations of the **`RandomBag::size()`** and **`RandomBag::isEmpty()`** member functions mean?

Solution

It indicates that those member functions aren't allowed to change the data members of the class. Only **const** member functions can be called on an object when in a function that accepts an object of that class by const reference.

Now that you're a bit more acclimated to the syntax, let's look at this code in a bit more detail.

First, some notes. The **Vector** type's **+=** operator takes time O(1) to run. The square brackets on the **Vector** type select a single element out of its underlying array, which takes time O(1) as well.

The **Vector::remove()** function, on the other hand, takes time proportional to the distance between the element being removed and the last element of the **Vector**. The reason for this is that whenever a gap opens up in the **Vector**, it has to shift all the elements after the gap down one spot. As a result, removing from the very end of a **Vector** is quite fast – it runs in time O(1) – but removing from the very front of a **Vector** takes time O(n).

iv. Look at the implementation of our **RandomBag::removeRandom** function. What is its worst-case time complexity? How about its best-case time complexity? Its average-case time complexity?

Solution

The worst-case time complexity of an operation is O(n), which happens when we remove the very first element from the Vector. Our best-case complexity is O(1) if we remove from the end of the Vector. On average, the runtime is O(n), since on average n/2 elements need to get shifted over.

---

v. Based on your answer to the previous part of this question, what is the worst-case time complexity of removing all the elements of an n-element **RandomBag**? What's the best-case time complexity? How about its average case?

Solution

The worst-case complexity would be O(n^2), which would happen if we get very unlucky and always remove the very first element of the Vector, making the total work done roughly equal to(n-1) + (n-2) + (n-3) + … + 2 + 1 = O(n^2). The best-case complexity would be O(n), which happens if we always remove the last element of the Vector (n × O(1) = O(n). The average-casetime complexity is O(n^2), since on average each removal does O(n) work and we do it n times.

---

vi. In the preceding discussion, we mentioned that removing the very last element of a **Vector** is much more efficient that removing an element closer to the middle. Rewrite the member function **RandomBag::removeRandom** so that it always runs in worst-case O(1) time.

Solution

There a couple of different ways to do this. One option is based on the insight that removing from the very end a Vector is an O(1)-time operation, so if we remove the last element of the Vector at each step we'll get an O(1) worst-case bound. The problem is that the last element is decidedly not a random element, since it always holds the last thing added. However, we can easily fix this by simply choosing a random element of the array and swapping it with the one at the end. This makes the element at the end randomly-chosen, which we still need to do, but makes deletions run in time O(1). Here's some code for this.

```
int RandomBag::removeRandom() {
    if (isEmpty()) {
        error("That which is not cannot be!");
    }
    int index = randomInteger(0, size() - 1);
    int result = elems[index];
    swap(elems[index], elems[elems.size() - 1]);
    elems.remove(elems.size() - 1);
    return result;
}
```

vii. The **Stack** and **Queue** types each have **peek** member functions that let you see what element would be removed next without actually removing anything. How might you write a member function **RandomBag::peek** that works in the same way? Make sure that the answer you give back is actually consistent with what gets removed next and that calling the member function multiple times without any intervening additions always gives the same answer.

Solution

Part of the challenge here is that we need to find a way to determine what the next element to be removed is going to be, but we have to do so in way that gives consistent results from call to call.

There are many different ways we can do this. One option, which guarantees a uniformly-random selection of elements from the RandomBag, is to store an extra variable keeping track of the index of the nextelement to remove. Every time we add or remove an element, we'll update this value to hold a new ran-dom value. Here's one way we can do this. First, the changes in the header:

```
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    int peek() const; // <-- Don't forget to make this const!
    int size() const;
    bool isEmpty() const;
private:
    Vector<int> elems;
    int nextIndex;
};
```

Next, the changes to the **.cpp** file. We've moved a lot of the logic out of **RandomBag::removeRandom** into **RandomBag::peek** in order to unify the code paths and avoid duplicating our logic.

```
void RandomBag::add(int value) {
    elems += value;

    /* Stage a new element for removal. */
    nextIndex = randomInteger(0, elems.size() - 1);
}
int RandomBag::peek() const {
    if (isEmpty()) {
        error("That which is not cannot be!");
    }
    return elems[nextIndex];
}
int RandomBag::removeRandom() {
    int result = peek();

    swap(elems[nextIndex], elems[elems.size() - 1]);
    elems.remove(elems.size() - 1);

    /* Stage a new element for removal. */
    nextIndex = randomInteger(0, elems.size() - 1);
    return result;
}
```

Problem Two: Pointed Points about Pointers

Pointers to arrays are different in many ways from Vector or Map in how they interact with pass-by-value and the = operator. To better understand how they work, trace through the following program. What is its output?

```cpp
void print(int* first, int* second) {
    for (int i = 0; i < 5; i++) {
        cout << i << ": " << first[i] << ", " << second[i] << endl;
    }
}

void transmogrify(int* first, int* second) {
    for (int i = 0; i < 5; i++) {
        first[i] = 137;
    }
}

void mutate(int* first, int* second) {
    first = second;
    second[0] = first[0];
}

void change(int* first, int* second) {
    first = new int[5];
    second = new int[5];

    for (int i = 0; i < 5; i++) {
        first[i] = second[i] = 271;
    }
}

int main() {
    int* one = new int[5];
    int* two = new int[5];

    for (int i = 0; i < 5; i++) {
        one[i] = i;
        two[i] = 10 * i;
    }

    transmogrify(one, two);
    print(one, two);

    mutate(one, two);
    print(one, two);

    change(one, two);
    print(one, two);

    delete[] one;
    delete[] two;
    return 0;
}
```

Solution

The output of the program is shown here:

```
0: 137, 0
1: 137, 10
2: 137, 20
3: 137, 30
4: 137, 40
0: 137, 0
1: 137, 10
2: 137, 20
3: 137, 30
4: 137, 40
0: 137, 0
1: 137, 10
2: 137, 20
3: 137, 30
4: 137, 40
```

Remember that when passing a pointer to a function, the **pointer is passed by value**! This means that you can change the contents of the array being pointed at, because those elements aren't copied when the function is called. On the other hand, if you change **which** array is pointed at, the change does not persistoutside the function because you have only changed the copy of the pointer, not the original pointer it-self

Problem Three: Cleaning Up Your Messes

Whenever you allocate an array with **`new[]`**, you need to deallocate it using **`delete[]`**. It's important when you do so that you only deallocate the array exactly once – deallocating an array zero times causes a memory leak, and deallocating an array multiple times usually causes the program to crash. (Fun fact – deallocating memory twice is called a double free and can lead to security vulnerabilities in your code! Take CS155 for details.)

Below are three code snippets. Trace through each snippet and determine whether all memory allocated with **`new[]`** is correctly deallocated exactly once. If there are any other errors in the program, make sure to report them as well.

```
int main() {
    int* baratheon = new int[3];
    int* targaryen = new int[5];

    baratheon = targaryen;
    targaryen = baratheon;

    delete[] baratheon;
    delete[] targaryen;

    return 0;
}
```

```
int main() {
    int* stark     = new int[6];
    int* lannister = new int[3];

    delete[] stark;
    stark = lannister;

    delete[] stark;

    return 0;
}
```

```
int main() {
    int* tyrell = new int[137];
    int* arryn  = tyrell;

    delete[] tyrell;
    delete[] arryn;

    return 0;
}
```

Solution

The first piece of code has two errors in it. First, the line

`baratheon = targaryen;`

causes a memory leak, because there is no longer a way to deallocate the array of three elements allocated in the first line. Second, since both **`baratheon`** and **`targaryen`** point to the same array, the last two lines will cause an error. The second piece of code is perfectly fine. Even though we execute

`delete[] stark;`

twice, the array referred to each time is different. Remember that you delete *arrays*, not *pointers*.

Finally, the last piece of code has a double-delete in it, because the pointers referred to in the last two lines point to the same array.

Problem Four: Creative Destruction

Constructors and destructors are unusual functions in that they're called automatically in many contexts and usually aren't written explicitly. To help build an intuition for when constructors and destructors are called, trace through the execution of this program and list all times when a constructor or destructor are called.

```
/* Prints the elements of a stack from the bottom of the stack up to the top
 * of the stack. To do this, we transfer the elements from the stack to a
 * second stack (reversing the order of the elements), then print out the
 * contents of that stack.
 */
void printStack(Stack<int>& toPrint) {
    Stack<int> temp;
    while (!toPrint.isEmpty()) {
        temp.push(toPrint.pop());
    }

    while (!temp.isEmpty()) {
        cout << temp.pop() << endl;
    }
}

int main() {
    Stack<int> elems;
    for (int i = 0; i < 10; i++) {
        elems.push(i);
    }

    printStack(elems);
    return 0;
}
```

Solution

The ordering is as follows:

- A constructor is called when **`elem`** is declared in **`main`**.
- A constructor is then called to initialize the **`temp`** variable in **`printStack`**.
- When **`printStackexits`**, a destructor is called to clean up the **`temp`** variable.
- When **`main`** exits, a destructor is called to clean up the **`elems`** variable.