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

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

Solution

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

Solution

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

---

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

---

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

---

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

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?

```
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

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

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