COEN 79: Object-Oriented Programming
Homework #3

Tamir Enkhjargal

February 6, 2020

Question #1

We create an array of `fruit` in the `main` function. How can we make sure that for all the items in array `fruit_ptr` the values of `weight` and `color` are equal to 1 and 2, respectively? Please show your solution. Do not modify the `main()` function.

```
class fruit {
private:
    int weight;
    int color;
public:
    fruit() {
        weight = 1;
        color = 2;
    }
}

main() {
    fruit* fruit_ptr;
    fruit_ptr = new fruit[100];
}
```

Question #2

How is *stack memory* used? And what are its pros and cons?

When functions declare new variables they are pushed onto the stack. They are freed when the functions ends. This means that we don't need to worry about memory management. Stack memory is also faster than heap memory. A consequence is that you could have stack overflow when you have too many nested function calls or you generate too large of objects.

Question #3

Explain why *heap* variables are essentially global in scope. Please present an example as well.

```
int* car_ptr; // pointer to heap object
car_ptr = new int[10]; // generates memory on heap
```

At this point, the car_ptr that points to an integer array in heap memory is now available to everyone, they just need to use the car_ptr.

Question #4

When do we call that a *resource is leaked*? Present a sample code and explain your answer.

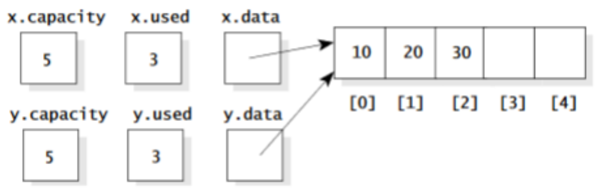
When you lose the pointer to memory on heap, and then are not able to access that object on heap anymore. This occurs when you do not delete the heap object before the object on stack gets destroyed at the end of the function.

```
main() {
    int* ptr;
    ptr = new int;  // creates int object on heap
//  delete ptr      // this should be included to delete the object on heap
}                   // this deletes ptr object on stack
```

Question #5

Explain why the *automatic assignment operator fails for the dynamic bag* (or for any other class that uses dynamic memory). Present your answer with a figure.

The automatic assignment operator fails because it fails to create a shallow copy of the pointers. The pointers that are copied over will just point to the same original object, instead of a copy of the object. Therefore we must overload the assignment operator when working with pointers.



The automatic assignment operator would make the new bag `y` point to the same array as bag `x`, instead of making a copy of the data.

Question #6

Is it possible to use the keyword "`this`" inside a friend function? Please explain your answer.

Friend functions are non member functions, even though they are allowed access to private functions. Because of this they are not activated by an object of that class. Therefore, using `this` would not work, because the compiler does not know what `this` points to.

Question #7

What is the output of this code? Explain your answer.

```
#include <iostream>
using namespace std;

class Book {
public:
    static int number;

    void bookInfo() {
        number++;
        delete this;
    }
};

int Book::number = 0;

int main() {
    Book* b = new Book();
    b->bookInfo();
    cout << "Book Information: " << Book::number << endl;
    Book book;
    book.bookInfo();
    cout << "Book Information: " << Book::number << endl;
    return 0;
}
```

```
Book Information: 1
```

And then the code will crash on the second one, because it runs the line `delete this;`, but the second book `book` doesn't point to anything.

Question #8

The definition of a bag class is as follow:

```
class bag {

public:
    typedef int value_type;
    typedef std::size_t size_type;
    static const size_type DEFAULT_CAPACITY = 30;

    bag(size_type initial_capacity = DEFAULT_CAPACITY);
    bag(const bag& source);
    ~bag();

    void reserve(size_type new_capacity);
    bool erase_one(const value_type& target);
    size_type erase(const value_type& target);
    void insert(const value_type& entry);
    void operator+=(const bag& addend);
    void operator=(const bag& source);

    size_type size() const { return used; }
    size_type count(const value_type& target) const;

private:
    value_type* data;
    size_type used;
    size_type capacity;
};
```

Write the full implementation of the following function:

```
void bag::reserve(size_type new_capacity)
// Postcondition: The bag's current capacity is changed to the
// new_capacity (but not less than the number of items already in the bag).
```

```
void bag::reserve(size_type new_capacity) {
    value_type temp;
    if(new_capacity == capacity)
        return;
    if(new_capacity < used)
        new_capacity = used;
    temp = new value_type [new_capacity];
    copy(data, data + used, temp);
    delete[] data;
    data = temp;
    capacity = new_capacity;
}
```

Question #9

What is the output of this code? Please explain your answer.

```
#include <iostream>
using namespace std;

class A {
    int val;
public:
    A() { this->val = 5; }
    void setValue(int a) { this->val = a; }
    int getValue() { return val; }
};

class B {
    A ob;
public:
    void foo(A ob) {
        this->ob = ob;
        ob.setValue(10);
    }
};

int main() {
    A ao;
    B bo;
    cout << "value = " << ao.getValue() << '\n';
    bo.foo(ao);
    cout << "value = " << ao.getValue() << '\n';
    return 0;
}
```

```
value = 5
value = 5
```

Because the function `foo(A ob)` uses a value parameter, which creates a copy of the object, and it gets destroyed afterwards. Therefore, the original value of the A object is not changed.

Question #10

Does the following code compile? Does it run? Is there any problem with the code? If yes, how do you fix it?

```
#include <iostream>
using namespace std;

class Computer {
    int Id;
public:
    Computer(int id) { this->Id = id; }
    void process() { cout << "Computer::process()"; }
};

class Employee {
    Computer* c;
public:
    Employee() { c = new Computer(123); }
    ~Employee() {}
    void foo() {
        cout << "Employee::foo()";
        c->process();
    }
};

int main() {
    Employee ob;
    ob.foo();
    return 0;
}
```

The code compiles and runs. It outputs:

```
Employee::foo()Computer::process()
```

There is still an issue in the code however, because the destructor has no actual code in there. To fix it, just include `delete c` in the implementation.

The code originally compiles and runs because it sees that a destructor exists, but the actual implementation has nothing in there. After running, you have the computer object on heap leaked.