COEN 79

Object-Oriented Programming and Advanced Data Structures

Assignment #3 - Due: October 22, 2018 – 11:59PM

Name: Jordan Murtiff **Date:** 10-22-18

- Number of questions: 10
- Points per question: 0.4
- Total: 4 points

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;
}

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

In the public section of the fruit class, add a default constructor that is called when you initialize a dynamic array of 100 fruit objects:

```
public:
fruit()
{
   this->weight = 1;
   this->color = 2;
}
```

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

Stack memory is used to hold local/temporary variables that are created by each function. Once a function ends, all variables on the stack are freed and the memory becomes available to other stack variables. So stack variables only exist while a function that has created them is running. The pros of stack memory is that the program is faster (since it doesn't have to allocate dynamic memory) and the user/programmer does not have to manage memory at all and can let the system take care of allocating and freeing variables. The cons however, is that the stack is limited in size and can only hold so many variables. If we try to allocate a very large array or too many variables in our code, we might get a stack overflow and cause our program to crash. Additionally, another con is that stack variables cannot change their size during runtime.Once delared as a stack variable, we can only change the size of the data structure/variable during compile time. If we wish to change the size of the data structure while the program is running, we have to declare it as a dynamic variable.

An Example of how stack variables are declared:

```
void stack_function()
{
   int a;
   double b;
}
```

Both variables a and b are stack variables that are deallocated once the stack_function has finished completing all its tasks. Once deallocated the space used by the variables a and b can be used for other stack variables that are created in other functions.

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

Heap variables are global in scope as long as you have a pointer that points to the heap variable. If we are able to pass the pointer that points to the heap variable to a function (whether as a value parameter or as a reference parameter) we are able to access the value of a heap variable and even change what that value is.

Example:
```
main()
{
 int *a;
 function_1(a);
 function_2(a);
 delete a;
}

 function_1(int *&b)
{
  b = new int (3);
}

 function_2(int *c)
{
  *c = 5
}
```

So first we create a new integer pointer “a”. This pointer is passed as a reference parameter to function_1. This function allocates a dynamic integer variable and makes it so that “a” is pointing to this dynamically allocated integer. Finally, integer pointer “a” is passed to function_2 as a value parameter (which calls the copy constructor and makes a new integer pointer called “c” to point to the same memory location as “a”). The memory location that “c” is pointing to (which is the same memory location that “a” is pointing to) is now changed from a value of 3 to a value of 5. When the main function finishes, a new dynamic integer variable is created, and its value has been changed from a value of 3 to a value of 5 within two different functions. Unlike stack variables, as long as you pass the pointer pointing to a heap variable to another function, you can manipulate and/or access the values of the heap variables in any function. Stack variables can only be accessed by the functions in which they were created and we cannot create pointers that point to stack variables, thereby being unable to pass a pointer of a stack variable and access/change the value in another function.

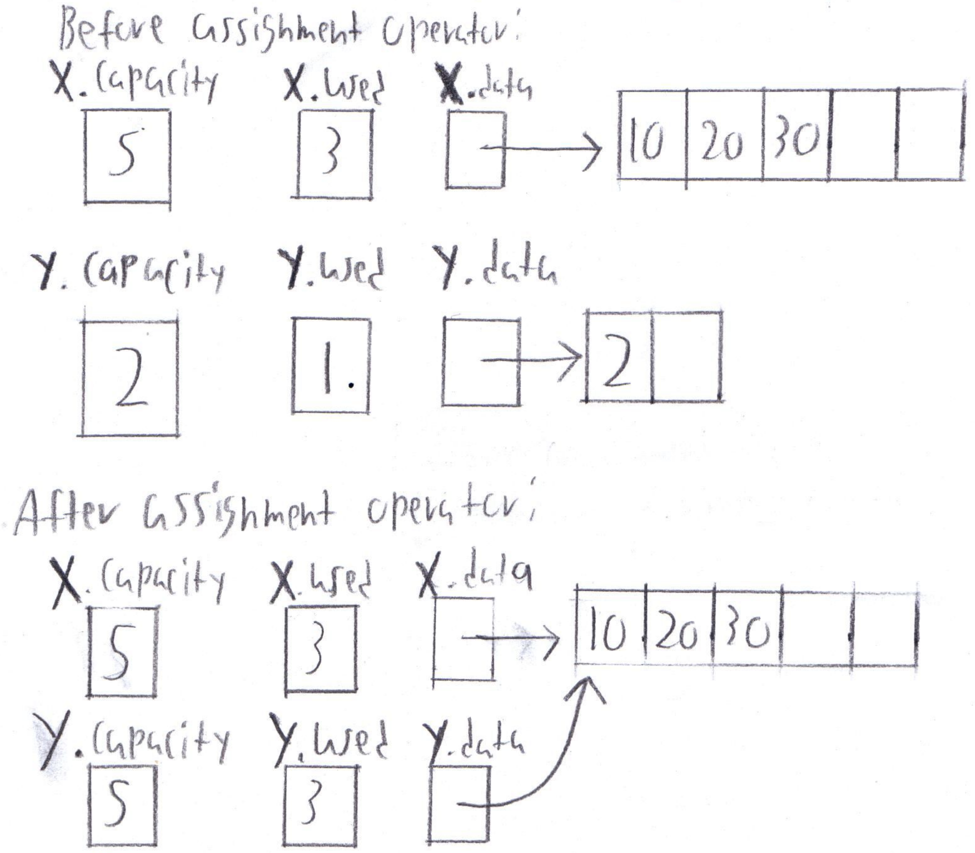
4. When we call that a *resource is leaked*? Present an example and explain your answer.

A resource is leaked when we allocate dynamic memory that is not deallocated at the end of a function/program. When a programmer allocates dynamic memory (heap variables) it is the responsibility of the programmer to deallocate these variables when they are no longer of any use. If the dynamic memory is not deallocated, then a memory leak has occurred.

Example:
```
int main()
{
 int *test;
 test = new int (7);
}
```

At the end of the main function, the integer pointer “test” is deallocated (since it is a stack variable), however, the heap variable that “test” was pointing to is not automatically deallocated. Now that there are no pointers pointing to the memory location that make up the integer variable "test" the memory has become inaccessible to other functions/variables of the program. Since it is being used up, it cannot be allocated for other variables to use and it is therefore a space of unusable memory. Only by using the "delete" keyword either within a function or as a destructor can we ensure that heap memory is deallocated when not in use.

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 ("=") can only be used for classes that do not contain pointers for their private member variables. In the case of our dynamic bag class we have a value of used (the current number of elements in the bag) a value for capacity (the total number elements that can fit in the bag) as well as a "data" pointer that points to the array that holds all the elements of our bag.

So the problem is that even though the values of capacity and used have changed for the bag "y", now both bag "x" and bag "y" are pointing to the same dynamically allocated array. In fact, since we have changed where y.data points to, we cause a memory leak since we never deallocate the dynamic memory before changing what the "data" pointers points to for the "y" bag. If we change one bag, it changes the other bag as well. If we add something to bag "x", it adds the same to bag "y". Either way, we only have one bag to work with instead of both bag "x" and bag "y" pointing to separate dynamically allocated arrays. Changing where a pointer points is not the equivalent of creating a new data structure to hold the elements of a bag.

6. Is it possible to use keyword "`this`" inside a friend function? Why?

No, you cannot use the keyword "this" inside a friend function. The "this" keyword acts as a pointer to the object that calls a member function. Since a friend function is not a member function (and therefore cannot be called by an object) it cannot use the "this" keyword to access or change the private member variables of an object. If you wish to use the "this" keyword inside a function, it must be a member function.

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

```
1.  #include < iostream >
2.  using namespace std;
3.
4.  class Book {
5.  public:
6.      static int number;
7.
8.      void bookInfo()   {
9.          number++;
10.         delete this;
11.     }
12. };
13.
14. int Book::number = 0;
15.
16. int main() {
17.     Book* b = new Book();
18.     b -> bookInfo();
19.     cout << "Book Information: " << Book::number << endl;
20.     Book book;
21.     book.bookInfo();
22.     cout << "Book Information: " << Book::number << endl;
23.     return 0;
24. }
```

This code does compile, however when the code is run it should crash on line 21 after trying to deallocate the stack variable “book”. Since the “book” object is a local variable (on the stack) it cannot be deallocated using the “delete” keyword, but can only be deallocated after the main function ends. This code should print out “Book Information: 1” and then afterwards try to free the allocated memory for the book variable. In doing so it is trying to free a pointer that has not been allocated and will end up crashing the program.

Lines 17-18 work fine, Lines 20-21 will cause the program to crash due to trying to deallocate a stack variable that is handled by the system.

8. The definition of a bag class is as follows:

```
class bag {

public:
    // TYPEDEFS and MEMBER CONSTANTS
    typedef int value_type;
    typedef std::size_t size_type;
    static const size_type DEFAULT_CAPACITY = 30;

    // CONSTRUCTORS and DESTRUCTOR
    bag(size_type initial_capacity = DEFAULT_CAPACITY);
    bag(const bag & source);
    ~bag();

    // MODIFICATION MEMBER FUNCTIONS
    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);

    // CONSTANT MEMBER FUNCTIONS
    size_type size() const {  return used;  }
    size_type count(const value_type & target) const;

private:
    value_type* data;   // Pointer to partially filled dynamic array
    size_type used;     // How much of array is being used
    size_type capacity; // Current capacity of the bag
};
```

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 *larger array;
 if (new_capacity == used)
 {
  return;
 }

 if (new_capacity < used)
 {
  new_capacity = used;
 }

 larger_array = new value_type[new_capacity];
 copy(data, data+used, larger_array);
 delete [ ] data;
 data = larger_array;
 capacity = new_capacity;
}
```

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

```cpp
#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; //Line 1

    B bo;

    cout << "value = " << ao.getValue() << '\n';

    bo.foo(ao); //Line 2

    cout << "value = " << ao.getValue() << '\n';
    return 0;
}
```

The two values being printed out would both have the value of 5. Since the default constructor of an "A" object sets the value of “val” to be 5, that is the first value that is printed out when the member function getValue() is called. When you call the foo function with the parameter of "ao" you are passing the object named "ao" as a value parameter and therefore creating a copy of it (using the copy constructor). This copy then has its value changed from 5 to 10, but then the copy is deallocated once the foo function ends. This means that once we print out the value of "val" for the object "ao" it will again by 5 since we never changed the original object but instead we changed a copy of the "ao" object. If we wanted to change the value of the "ao" object, we would have had to pass it as a refrence parameter to the "foo" function.

So the output should be:
5
5

Because the "foo" function uses value parameters, we cannot explicity change the object being passed to it, instead we are just changing a copy of the object.

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 does compile and run, however, the Computer variable that is dynamically allocated when a new Employee object is created is not deallocated when the main function finishes. Since the Computer object is a dynamic variable, it must be deallocated at the end of the main function to ensure a memory leak doesn't happen.

To fix the problem we should implement the destructor for the Employee class, shown here:

```
~Employee()
{
 delete c;
}
```

Now at the end of the main function, when the Computer pointer "c" is deallocated, the destructor will be called and deallocate the heap memory that the pointer "c" is pointing to before deallocating itself. This will ensure that a memory leak does not occur.