Informatik - AS19

# Exercise 13: Memory Management with Classes

*Handout: 9. Dez. 2019 06:00*

*Due: 16. Dez. 2019 18:00*

---

## Task 1: Operator delete

Open Task

*This task is a text based task. You do not need to write any program/C++ file: the answer should be written in* main.md *(and might include code fragments if questions ask for them).*

# Task

All the following code fragments use operator `delete` and `delete[]` to deallocate memory, but not appropriately. This can either lead to an error or to a memory leak. Find the mistake in each code fragment, explain whether it results in a memory leak or an error, and in the case of an error, point out the location at which it occurs.

1. ```
   class A {
   public:
       A(unsigned int sz) {
           ptr = new int[sz];
       }
       ~A() {
           delete ptr;
       }
       /* copy constructor, assigmnent operator, public methods
       ...
   private:
       int* ptr;
   };
   ```

2. ```
   struct llnode {
       int value;
       llnode* next;
   ```

```
};

void recursive_delete_linked_list(llnode* n) {
    if (n != nullptr) {
        delete n;
        recursive_delete_linked_list(n->next);
    }
}
```

3.
```
class A {
public:
    A() {
        c = new Cell;
        c->subcell = new int(0);
    }
    ~A() {
        delete c;
    }
    /* copy constructor, assignment operator, public methods
    ...
private:
    struct Cell {
        int* subcell;
    };
    Cell* c;
};
```

4.
```
void do_something(int* p) {
  /* Do something */
  ...
}
void f() {
    int v;
    int* w = &v;
    do_something(w);
    delete w;
}
```

5.
```
class Vec {
public:
    Vec(unsigned int sz) {
        array = new int[sz];
    }
```

```
    ~Vec() {
        delete[] array;
    }
    int& operator[](int l) {
        return array[l];
    }
    /* copy constructor, assignment operator, other public m
    ...
private:
    int* array;
};

void f() {
    Vec v(5);
    delete[] &v[0];
}
```

---

## Task 2: Array-based Vector, Rule of Three

Open Task

# Task

You are provided a partial implementation of an array-based vector class `avec`. Declarations are given in file `avec.h`, member functions that are already implemented are in file `avec_locked.cpp`. Your task is to implement the copy constructor, assignment operator and destructor for class `avec`, in file `avec.cpp`. Note that the vectors store values of type `tracked` and not `int` as in the lecture, but you can treat them in the same fashion; the `tracked` values are used behind the scenes for checking your solution.

**Steps:**

1. Implement the copy constructor for class `avec` so that it creates a copy of the internal array.
2. Implement the assignment operator so that it creates a copy of the provided `avec` and destroys the vector that is currently assigned to the left-hand-side of the assignment (e.g., by swapping its current contents to a local copy, as described in the lecture).
3. Implement the destructor so that it deallocates the internal array.

**Memory tracking:** The internal elements of class `avec` are objects of class `tracked` (see file `tracker.h`). Such objects encapsulate a single integer location which is

tracked by an internal memory manager. This is used internally to catch as many memory/deallocation errors as possible upon occurrence.

**Testing:** Tests are already provided in file `main.cpp`. If you want to carry out further testing yourself, you may do so within function `your_own_tests()`, which is called by `main()` when encountering an unknown test identifier. You can edit this function in file `avec.cpp`.

---

## Task 3: Smart Pointers

Open Task

# Task

The objective of this problem is to implement a *reference-count smart pointer*, with functionality similar to that of a `std::shared_ptr`. Smart pointers implement the same functionality as regular pointers, but additionally they automatically take care of deallocating the object they point to when it is no longer needed. Reference-count smart pointers achieve this by allocating and maintaining a counter in memory together with the actual pointed-to object, which represents the number of smart pointers currently referencing the object. Any time a new smart pointer to an object is created or one is destroyed, this counter has to be incremented or decremented, respectively. When the last smart pointer to an object is destroyed, the counter will be decremented to $0$ and the smart pointer will know that the object is no longer referenced; it can then deallocate the object.

The following example illustrates the process:

```
Smart a, b, c;
a = Smart(new tracked);
// The smart pointer 'a' now points to the new
// tracked object with a reference count of 1.
c = Smart(new tracked);
// Another smart pointer 'c' now points to the
// second new tracked object with count 1.
b = a;
// 'a' and 'b' now both point to the first object,
// their shared counter is incremented to 2.
c = Smart();
// A null smart pointer is assigned to 'c'; the
// smart pointer previously stored in 'c' is destroyed
// and its counter is decremented. Since the count
// is now zero, the second tracked object is deallocated.
return;
```

```
// At the end of the function, the smart
// pointers 'a' and 'b' are both destroyed, their
// counter is decremented twice, and the first
// tracked object is deallocated.
```

**Locations:** The declarations of the smart pointer class (`Smart`) and member functions is provided in file `smart.h`. The implementation of member functions should be done in file `smart.cpp`. Smart pointers encapsulate pointers to objects of class `tracked`, which is declared in file `smart.h`.

**Structure:** In class `Smart`, member variable `ptr` represents the pointer (potentially shared by several object of class Smart) to the underlying pointed-to object. Member variable `count` represents the (shared) location containing the number of objects of class Smart currently holding the pointed-to object. Alternatively, both pointers may be `nullptr`, which corresponds to the notion of a *null* smart pointer. A null smart pointer does not manage any memory.

Objects pointed by smart pointers belong to class `tracked`, which is a linked list node where the next pointer is represented using a smart pointer. In particular, tests will use this structure to build linked lists with shared nodes, and check at the end that everything was correctly deallocated. To that end, every object of class `tracked` is tracked behind the scenes.

**Steps:**

1. Implement the default constructor for class `Smart`. The default constructor should create a null smart pointer.

2. Implement the constructor `Smart(tracked* t)`. If `t==nullptr`, this should return a null smart pointer, otherwise, this should create a smart pointer to `t` with a reference count of 1. You may assume that `t` points to memory newly allocated by `new` that is not already being managed by another smart pointer.

3. Implement the copy constructor `Smart(const Smart& src)` that returns a new smart pointer to the memory pointed to by `src` and increments the shared reference counter.

4. Implement the assignment operator `Smart& operator=(const Smart& src)` that creates a copy of the provided smart pointer (incrementing its reference counter) and decrements the counter of the smart pointer of the left hand side of the assignment (and potentially deallocates the memory it points to).

5. Implement the destructor `~Smart()`. For non-null pointers, it should decrement the reference counter and deallocate the pointed-to object if the resulting count is zero.

**Optional:** Figure out the situations in which smart pointers are not suitable for

memory management, in the sense that they may lead to memory leaks. You may look at the tests which leak memory for inspiration.