Informatik für Mathematiker und Physiker - AS18

# Exercise 13: Memory Management with Classes

*Handout: 11. Dez. 2018 06:00*

*Due: 17. Dez. 2018 23:59*

---

## Task 1: operator delete

Open Task (/solve/sZ6pigkCQGJSf3s7s)

*This task is a text based task. You do not need to write any program/C++ file: the answer should be written in* main.txt *(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, which of the two cases may occur, and in the case of an error, 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 metho
    ...
private:
    int* array;
};

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

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

Open Task (/solve/pb5Y8Zn3JNjdv6f4n)

# 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 copy constructor, assignment operator and destructor for class `avec`, in file `avec.cpp`.

**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 much memory/deallocation error as possible upon occurrence.

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

# Task 3: Smart Pointers

Open Task (/solve/EREmFysmM9s4jptHs)

# 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, except that when the last smart pointer to an object is destroyed, the pointed-to memory is deallocated as well. Reference-count smart pointer achieve this by allocating and maintaining an extra counter in memory together with the actual pointed-to object, which represents the number of smart pointers currently pointing to the pointed-to object. In particular, the destruction of the last smart pointer to an object is detected because the counter become $0$.

**Remark:** As one of the tests shows, using smart pointers instead of regular pointers is not *always* a suitable solution in order to prevent memory leaks.

**Locations:** The declarations of 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 pointer encapsulate pointer to object of class `tracked`, which is declared in file `smart.h`.

**Structure:** In class `Smart`, member variable `ptr` represent the pointer (potentially shared by several object of class Smart) to the underlying pointed-to object. Member variable `count` represent 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 and does not manage any memory.

Object 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 list with shared nodes, and check at the end that everything was correctly deallocated. To that end, every objects of class `tracked` are tracked behind the scenes.

**Steps:**

1. Implement default constructor for class `Smart`. Default constructor should create a null smart pointer.
2. Implement constructor `Smart(tracked* t)`. If `t==nullptr`, this should return a null smart pointer. Otherwise, the caller must enforces that `t` points to memory allocated by `new`. This constructor then makes smart pointers responsible for eventually deallocating the memory stored in `t` when it can no longer be used. In particular, no other smart pointer should be responsible for deallocation of `t` before this constructor is called.
3. Implement copy constructor, assignment operator and destructor for class

`Smart`.

**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.