# Exercise Session

## Week 13

Adel Gavranović
agavranovic@student.ethz.ch

# Overview

▸ polybox for session material ▸ mail to TA

## Today’s Topics

Introduction

Self-Assessment

-tors

Exercise ”Box”

Vocabulary

# Introduction

- We had very little exercises in the past few *exercise* sessions. Today will be more exercise focused
- Be ready to answer a lot of tiny questions

# Comments on last [code]expert Exercises

- When giving ranges in PRE/POST-conditions, make sure to be precise: `c in [0,127]` or `0 <= c < 128`
- Use `vec.at(i)` instead of `vec[i]` whenever you can. It is a little slower, but much safer!
- Great job on last week's exercises, especially the quicksort and nonogram exercise!

# Questions or Comments re: Exercises?

# Learning Objectives Checklist

**Now I...**

- ☐ can trace code that uses `new`, `delete`, copy-constructors, and destructors
- ☐ can implement simple data structures that act as values, but are implemented internally by using dynamic memory
- ☐ know how to avoid common problems with dynamically allocated memory (dangling pointers, double-free, use-after-free)
- ☐ understand the difference between `new/delete` and `new[]/delete[]`

# Questions?

# Self-Assessment IV

- log into the Moodle page and wait
- do the Self-Assessment (be aware of the 20 minute time limit)
- the Master Solution will be available when you review your solutions
- this has **no** impact on your final grade
- we’ll discuss parts of it after you’re done

# Questions?

# Remember...

## Don’t forget

To each `new` a `delete`.

## Constructor, Copy-Constructor, Destructor

- Are just fancy functions that get called on specific occasions
- Must be in the public section of your class/struct

# Constructor

## Constructor

- gets called when an object of that class/struct gets created/constructed
- can be used to pass construction arguments, so you can initialize the object however you like
- you can define multiple constructors (e.g. for different types) and the compiler will choose which one to use `classname object1(6.0f)` or `classname object2('A')`
- excellent resource on this: ▸ cppreference link

# Constructor example in a class

Good looking way of writing a constructor

```
class classname {
    int a, b;
public:
    const int& r;

    classname(int i)
      : r(a)  // initializes X::r to refer to X::a
      , a(i)  // initializes X::a to the value of i
      , b(i+5) // initializes X::b to the value of i+5
    { } // <- if you want your constructor to do
        anything else, put it in there
};
```

# Destructor

## Destructor

- gets called when an object gets deleted/deconstructed (at the end of a scope or when using `delete`)
- used, to clean up memory when an object is no longer needed (`delete`)

# Destructor example in a class

A way of writing a destructor

```
class classname {
    int* value;
public:

   ...

    ~classname(){
      delete value; // that's how we clean up the value
          where the int-pointer is pointing to, instead
          of just deleting the int-pointer (avoiding
          "memory leaks")
    }
};
```

# Copy-constructor

## Copy-Constructor

- gets called when initalizing a object with another object of the same class/struct
- enables you to modify how *exactly* you want the compiler to copy another object of the same class/struct (instead of just a ”shallow copy”)
- not to be confuse with `operator=`, which does a very similar thing (more on that later)

# Shallow Copy

What the copy-constructor does. (We usually want *deep* copy)

# Assignment-operator (=)

## Assignment-operator (=)

- gets called when *assigning* an object of the same class/struct to an object
- gets called *after* initialization
- called ”assignment operator”, just like with regular types (=)
- rule of thumb: activates destructor and then copy-constructor
- *has* a return type (usually `classname&`) so one can use ”chained assigments” (e.g. `a = b = c = d`, all of them will be assigned `d`)

# Difference between Assignment-operator= and Copy-Constructor

```
// our class/struct is named "Box"

Box first;
   // ^ initialization by default constructor
Box second(first);
   // ^ initialization by copy constructor
Box third = first;
   // ^ Also initialization by copy constructor
second = third;
   // ^ assignment by copy assignment operator
```

# Questions?

# Exercise ”Box (copy)”

## Task

- Go to `[code]`expert and open the code example ”Box (copy)”
- Don’t worry about main.cpp yet, we’ll get to that
- Don’t worry about `std::cerr`, it’s just fancy `std::cout`
- Program Tracing!

# Members of ”Box”[1]

```
Box::Box(const Box& other) {
  ptr = new int(*other.ptr);
}

Box& Box::operator= (const Box& other) {
  *ptr = *other.ptr;
  return *this;
}
```

[1] with all `std::cerr` removed

# Members of ”Box”[2]

```
Box::~Box() {
  delete ptr;
  ptr = nullptr;
}

Box::Box(int* v) {
  ptr = v;
}

int& Box::value() {
    return *ptr;
}
```

[2] with all `std::cerr` removed

# Tracing `test_destructor1()`

```
void test_destructor1() {
    std::cerr << "[enter] test_destructor1" << std::endl;
    int a;
    {
        Box box(new int(1));
        a = 5;
    }
    std::cout << "a = " << a << std::endl;
    std::cerr << "[exit] test_destructor1" << std::endl;
}
```

# Tracing `test_destructor2()`

```
void test_destructor2() {
   std::cerr << "[enter] test_destructor2" << std::endl;
   {
      Box* box_ptr = new Box(new int(2));
      delete box_ptr;
   }
   std::cerr << "[exit] test_destructor2" << std::endl;
}
```

# Tracing `test_copy_constructor()`

```
void test_copy_constructor() {
   std::cerr << "[enter] test_copy_constructor" <<
       std::endl;
   {
      Box demo(new int(0));
      Box demo_copy = demo;
      // assert(demo.value() == 0);
      // assert(demo_copy.value() == 0);
      demo.value() = 4;
      // assert(demo.value() == 4);
      // assert(demo_copy.value() == 0);
      demo_copy.value() = 5;
      // assert(demo.value() == 4);
      // assert(demo_copy.value() == 5);
   }
   std::cerr << "[exit] test_copy_constructor" <<
       std::endl;
}
```

# Tracing `test_copy_constructor()`

# Tracing `test_assignment()`

```
void test_assignment() {
   std::cerr << "[enter] test_assignment" << std::endl;
   {
      Box demo(new int(0));
      demo.value() = 3;
      Box demo_copy(new int(0));
      demo_copy = demo;
      // assert(demo.value() == 3);
      // assert(demo_copy.value() == 3);
      demo.value() = 4;
      // assert(demo.value() == 4);
      // assert(demo_copy.value() == 3);
      demo_copy.value() = 5;
      // assert(demo.value() == 4);
      // assert(demo_copy.value() == 5);
   }
   std::cerr << "[exit] test_assignment" << std::endl;
}
```

# Tracing `test_assignment()`

# Questions?

# Dangling Pointer

# Double-Free

# Use-after-Free

# Memory Leak

# [] or not to []?

## When to use []

Actually quite easy:

- use `new[]` if allocating more than one variable at a time (simple values or object)
- use `delete[]` if deallocating more than one variable at a time (arrays of values/object)

Check out the Summary 11 on how to actually use them in your code

# Questions?