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

Self-Assessment IV

- log into the Moodle page and wait
- do the Self-Assessment (be aware of the 20 minute time limit)

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

Self-Assessment IV

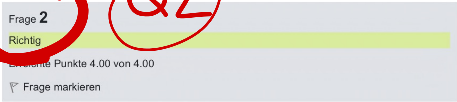
- log into the Moodle page and wait
- do the Self-Assessment (be aware of the 20 minute time limit)
- the Master Solution will be available when you review your solutions
- this has **no** impact on your final grade

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?

Q2

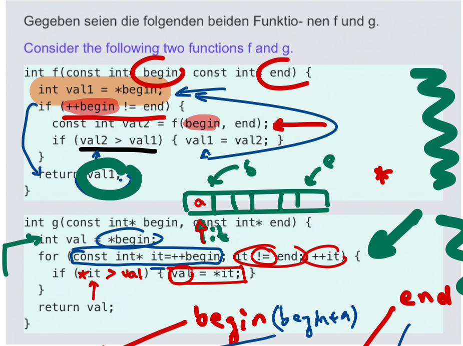


Frage **2**

Richtig

Erreichte Punkte 4.00 von 4.00

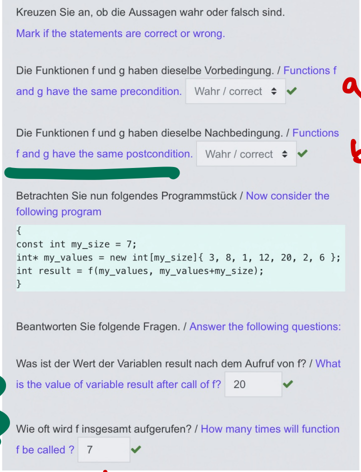
Frage markieren



Gegeben seien die folgenden beiden Funktio- nen f und g.

Consider the following two functions f and g.

```
int f(const int* begin, const int* end) {
  int val1 = *begin;
  if (++begin != end) {
    const int val2 = f(begin, end);
    if (val2 > val1) { val1 = val2; }
  }
  return val1;
}

int g(const int* begin, const int* end) {
  int val = *begin;
  for (const int* it=++begin; it != end; ++it) {
    if (*it > val) { val = *it; }
  }
  return val;
}
```



Kreuzen Sie an, ob die Aussagen wahr oder falsch sind.

Mark if the statements are correct or wrong.

Die Funktionen f und g haben dieselbe Vorbedingung. / Functions f and g have the same precondition. Wahr / correct ✓ a

Die Funktionen f und g haben dieselbe Nachbedingung. / Functions f and g have the same postcondition. Wahr / correct ✓ b

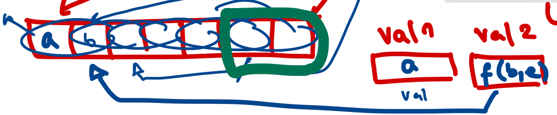
Betrachten Sie nun folgendes Programmstück / Now consider the following program

```
{
const int my_size = 7;
int∗ my_values = new int[my_size]{ 3, 8, 1, 12, 20, 2, 6 };
int result = f(my_values, my_values+my_size);
}
```

Beantworten Sie folgende Fragen. / Answer the following questions:

Was ist der Wert der Variablen result nach dem Aufruf von f? / What is the value of variable result after call of f? 20 ✓ c

Wie oft wird f insgesamt aufgerufen? / How many times will function f be called ? 7 ✓ d

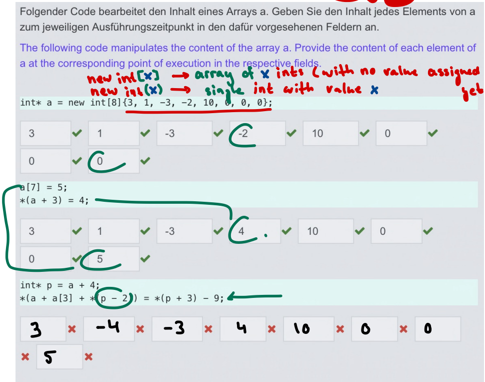


↳ 6 rec.

+1 init. call

Questions?

Q3



Folgender Code bearbeitet den Inhalt eines Arrays a. Geben Sie den Inhalt jedes Elements von a zum jeweiligen Ausführungszeitpunkt in den dafür vorgesehenen Feldern an.

The following code manipulates the content of the array a. Provide the content of each element of a at the corresponding point of execution in the respective fields.

new int[x] → array of x ints (with no value assigned yet.)
new int(x) → single int with value x

```
int* a = new int[8]{3, 1, -3, -2, 10, 0, 0, 0};
```

3 ✓ 1 ✓ -3 ✓ -2 ✓ 10 ✓ 0 ✓
0 ✓ 0 ✓

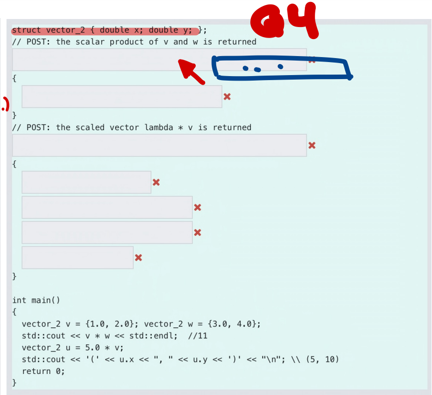
```
a[7] = 5;
*(a + 3) = 4;
```

3 ✓ 1 ✓ -3 ✓ 4 ✓ 10 ✓ 0 ✓
0 ✓ 5 ✓

```
int* p = a + 4;
*(a + a[3] + *(p - 2)) = *(p + 3) - 9;
```

3 ✖ -4 ✖ -3 ✖ 4 ✖ 10 ✖ 0 ✖ 0
✖ 5 ✖

Q4



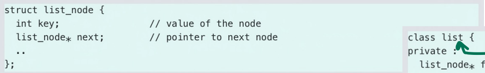
```
struct vector_2 { double x; double y; };
// POST: the scalar product of v and w is returned
                                                    ... ✖
{
                                                    ✖
}
// POST: the scaled vector lambda * v is returned
                                                    ✖
{
                                                    ✖
                                                    ✖
                                                    ✖
                                                    ✖
}

int main()
{
  vector_2 v = {1.0, 2.0}; vector_2 w = {3.0, 4.0};
  std::cout << v * w << std::endl;  //11
  vector_2 u = 5.0 * v;
  std::cout << '(' << u.x << ", " << u.y << ')' << "\n"; \\ (5, 10)
  return 0;
}
```

* a[x] is deref'd

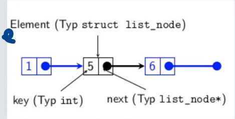
↑ nope

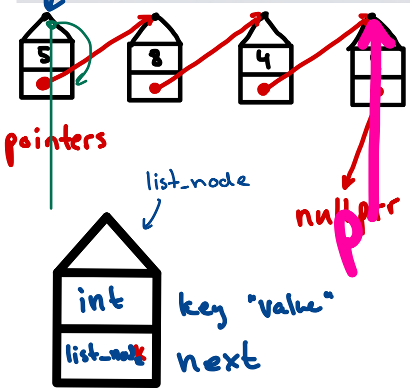
.at() ← std::vector thing

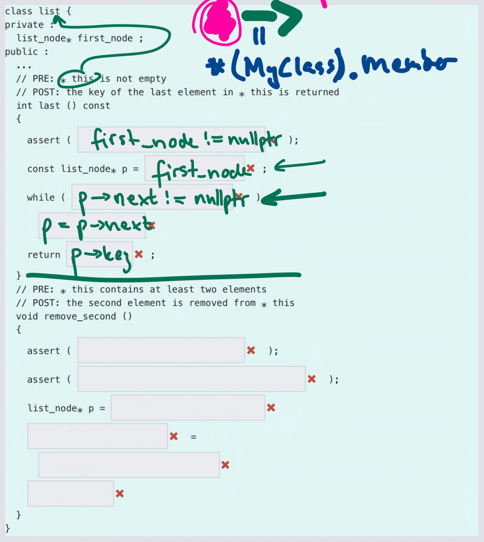
Questions?



```
struct list_node {
  int key;                // value of the node
  list_node* next;        // pointer to next node
  ..
};
```







```
class list {
private :
  list_node* first_node ;
public :
  ...
  // PRE: * this is not empty
  // POST: the key of the last element in * this is returned
  int last () const
  {
    assert ( first_node != nullptr );
    const list_node* p = first_node ;
    while ( p->next != nullptr )
      p = p->next
    return p->key ;
  }
  // PRE: * this contains at least two elements
  // POST: the second element is removed from * this
  void remove_second ()
  {
    assert (                    );
    assert (                    );
    list_node* p =
                      =


  }
}
```

must be a ptr

→ || *(MyClass).member

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

Constructor

Constructor

- gets called when an object of that class/struct gets created/constructed

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

Classname a(5)

"object"

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')`

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 main(){

```
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
};
```

class a(5);

a = 5

b = i+5 = 5+5 = 10

}

Destructor

Destructor

- gets called when

Destructor

Destructor

- gets called when an object gets deleted/deconstructed (at the end of a scope or when using `delete`)

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")
    }
};
```

"tilde"

Copy-constructor

Copy-Constructor

- gets called when

Copy-constructor

```
main(){
classname a(5);
classname b(a);
}
```

Copy-Constructor

- gets called when initalizing a object with another object of the same class/struct

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”)

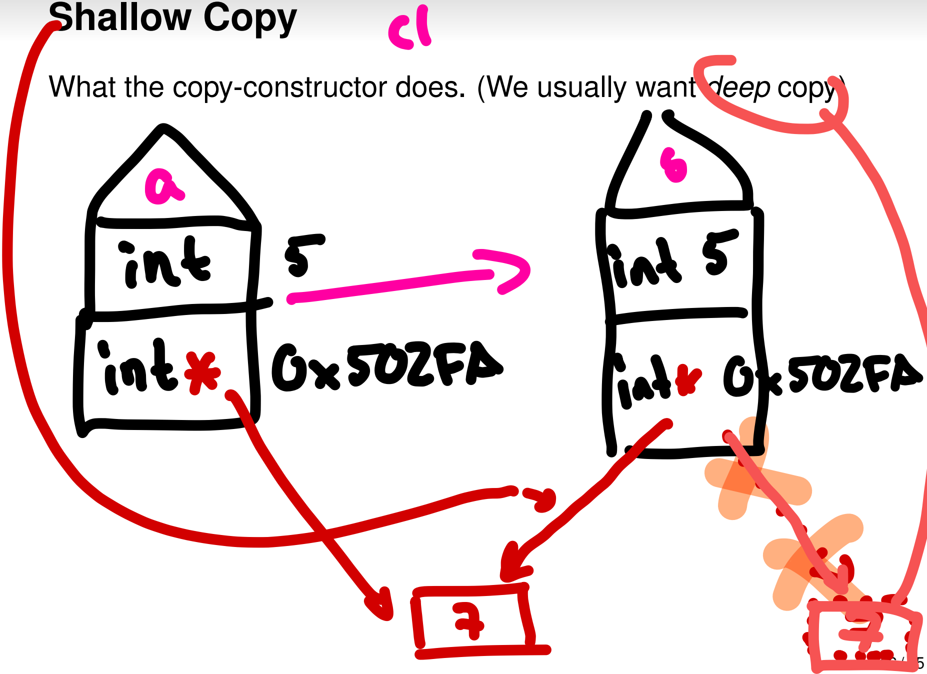
Copy-constructor

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 (=)

operator =

Assignment-operator (=)

- gets called when

Assignment-operator (=)

a = b;

same class

Assignment-operator (=)

- gets called when *assigning* an object of the same class/struct to an object

Assignment-operator (=)

Assignment-operator (=)

- gets called when *assigning* an object of the same class/struct to an object
- gets called *after* initialization (on init. stuff)

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 (=)

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

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&`)

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. $\underbrace{\texttt{a = }\underbrace{\texttt{b = }\underbrace{\texttt{c = d}}_{\&d}}_{\&d}}_{\&d}$ 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
```

() no arguments

(Ch7 init.)

Questions?

Box (copy)

Exercise ”Box (copy)”

Task

- Go to [code]expert and open the code example ”Box (copy)”

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

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`

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()`

```cpp
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()`

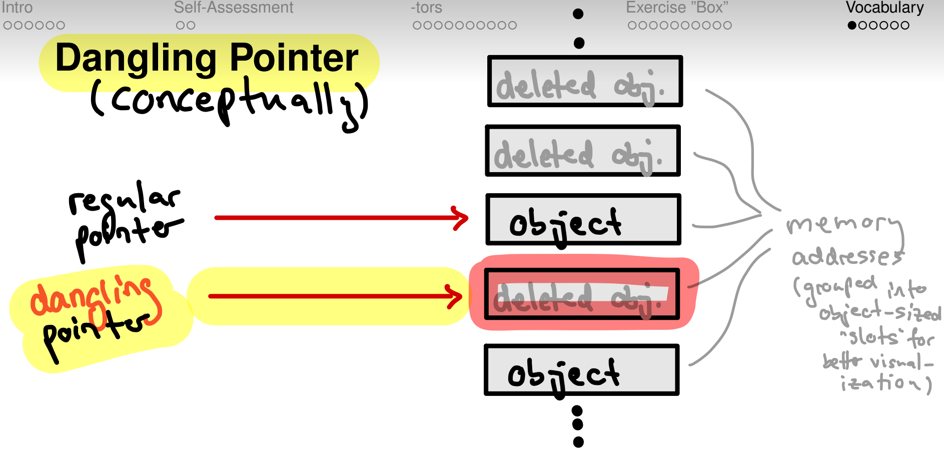
```
void test_assignment() {
   std::cerr << "[enter] test_assignment" << std::endl;
   {
      Box demo(new int(0));
      demo.value() = 3;
      Box demo_copy(new int(0));
      demo_copy = demo;
      // assert(demo.value() == 3);
      // assert(demo_copy.value() == 3);
      demo.value() = 4;
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

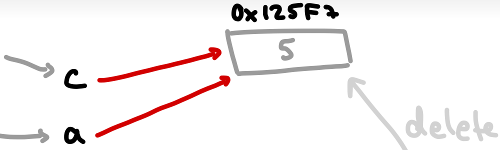
(conceptually)



dangling pointer occurs when the memory address that the pointer is pointing to got freed (with delete) but the pointer wasn't deleted

Double-Free

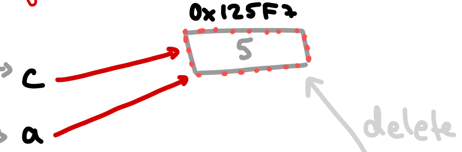
```
int* c = new int(5);
int* a = c;
```



```
delete a; // deletes ("frees") the memory
             address saved in a

delete c; // ERROR! Address was already
             freed (by the line before).
             Can't "free it again".
```

Use-after-Free (very closely linked to dangling pointers!)

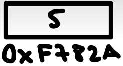
```
int* c = new int(5);
int* a = c;
```



```
delete a; // deletes ("frees") the memory
          //   address saved in a

// c is now a dangling pointer

std::cout << *c << std::endl; // will usually run,
                              //   but god knows
                              //   what's in that
                              //   address now.
                              //   Could be 0, could
                              //   be ¯\_(' ',)_/¯
```

⚠ no warnings or errors...

Memory Leak



```
{
  int* p = new int(5);
}
```

← here, our pointer will be deleted (like any "normally" (not dynamically) allocated value/object), because the scope ended.

wait... how can we now free the memory we allocated for our int? Well shit, we can't! We lost the address (our int* p which contained the address of where our int-value was stored.) That's a memory leak!

We allocated memory but can't free it again :(

`[]` or not to `[]`?

new[] or new

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?~~ // TODO:

□ read the (short) wikipedia article on "dangling pointers" (DE ≥ EN)

https://de.wikipedia.org/wiki/Hängender_Zeiger
https://en.wikipedia.org/wiki/Dangling_pointer

↳ Don't worry about "heap" and "stack" yet, those are just regions in our memory (band).

□ after delete-ing a pointer, it's good practice to set it to p = nullptr. (just to be safe)

□ Seriously, try the "Box (copy)" code example

□ check out the SEGFAULT.pdf on the Polybox. (pretty tricky!)