Schöne Visualisierung zu our_list von Lily

Recap: caller vs callee

A caller is a function that calls another function; a callee is a function that was called.

```c
int function(){
    int some_variable = 1;
    return some_variable;
}

int main(){
    int another_variable = 5;
    int nommal_öppis = function();
    //...
    return 0;
}
```

→ func callee

main ist caller

Recap: operators als member oder non-member und this

```
struct Complex{
    double r, i;
    // ...
}

// als non-member:
Complex operator+(const Complex &a, const Complex& b){
    Complex result;
    result.r = a.r + b.r;
    result.i = a.i + b.i;
    return result;
}

// als member:
Complex Complex::operator+(const Complex& other){
    Complex result;
    result.r = this->r + other.r;
    result.i = this->i + other.i;
    return result;
}
```

z1 + z2
↓
operator+(z1, z2)

z1 + z2
↓
z1.operator+(z2)

Dynamic Memory Allocation: new

The new operator denotes a request for memory allocation. If sufficient memory is available, the new operator default-initializes the memory and returns the address of the newly allocated memory.

- Heisst: `new` gibt uns immer einen Pointer zurück!

Weiterführende Links:

Allg. zu new: `https://www.geeksforgeeks.org/new-and-delete-operators-in-cpp-for-dynamic-memory/`

Bzgl. Initialisierung des Speichers: `https://stackoverflow.com/questions/7546620/operator-new-initializes-memory-to-zero`

Dynamic Memory Allocation: new

```
// runde Klammern:
// - wir wollen Speicher für einen einzigen Integer
// - p zeigt auf diesen int
int *p = new int(int_value);

// eckige Klammern:
// - wir wollen Speicher für num_ints Integer
// - p zeigt auf den ersten int
int *p = new int[num_ints];
```

Dynamic Memory Allocation: delete

```
// Speicher für ein Integer allozieren:
int *p = new int(int_value);
// Speicher für ein Integer löschen:
delete p;

// Speicher für einen Block allozieren:
int *p = new int[num_ints];
// Speicher für einen Block löschen:
delete[] p;
```

Dynamic Memory Allocation: delete

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

Grobe Regel:

- Wenn wir Variablen/Objekte mit `new` deklarieren, dann haben wir die ganze Kontrolle darüber, wann das Objekt kreiert und zerstört wird. Insbesondere gehen sie nicht "out-of-scope" so wie wir es sonst gesehen haben. (Heisst aber auch wir müssen uns darum kümmern.)
- Der ganze Sinn von dynamic memory ist ja, die volle Kontrolle über die Lebenszeit der Objekte zu haben.
- Für jedes `new` ein `delete` und für jedes `new[]` ein `delete[]`.

Constructor, Copy Constructor und operator=

```
int a = 6;
Complex z1(2,5);
```

- Constructor:
  - Wird aufgerufen, wenn ein neues Objekt einer Klasse initialisiert wird.

```
int b = a;
Complex z2(z1);
```

- Copy Constructor:
  - Wird aufgerufen, wenn ein neues Objekt mit einem schon existierenden Objekt der selben Klasse initialisiert wird.
- (Copy) Assignment Operator:
  - Wird aufgerufen, wenn ein bereits existierendes Objekt einem anderen bereits existierenden Objekt der selben Klasse zugewiesen wird.
  - Wird nur nach, nie bei einer Initialisierung gerufen.

```
int c;
c = a;
Complex z3;
z3 = z1;
```

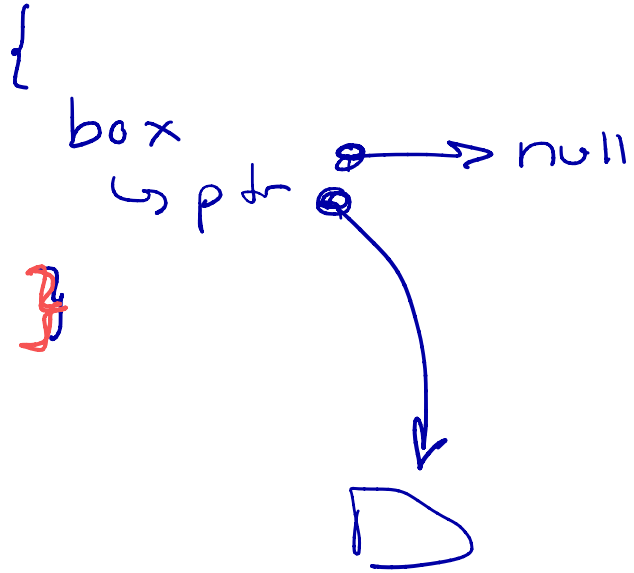
Constructor, Copy Constructor und operator=

```
Box::Box(int* v) {
  ptr = v;
  std::cerr << "constructor: value=" << *ptr;
  std::cerr << " at " << ptr << std::endl;
}
```

existiert bereits

```
Box::Box(const Box& other) {
  ptr = new int(*other.ptr);
  std::cerr << "copy constructor: value=" << *ptr;
  std::cerr << " at " << ptr << std::endl;
}

Box& Box::operator=(const Box& other) {
  *ptr = *other.ptr;
  std::cerr << "assignment operator: value=" << *ptr;
  std::cerr << " at " << ptr << std::endl;
  return *this;
}
```



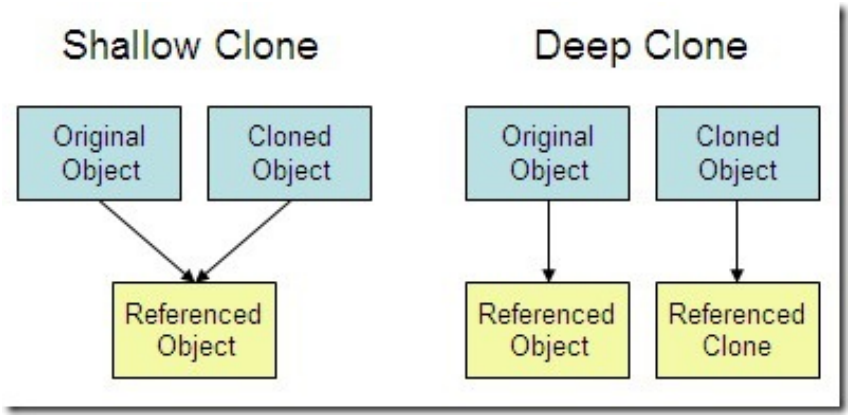

```
Box::~Box() {
  std::cerr << "destructor: value=" << *ptr << " at " << ptr << std::endl;
  delete ptr;
  ptr = nullptr;
}
```

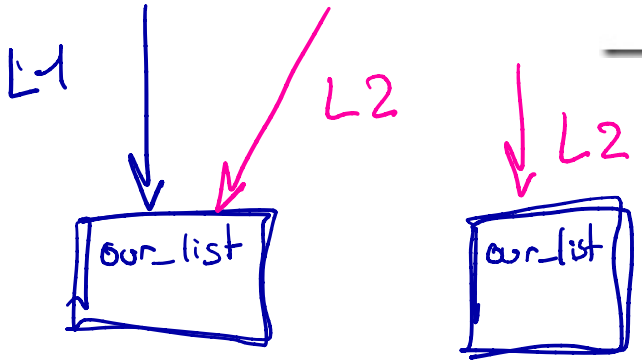
FYI: Shallow vs. Deep Copy

```
int a = 6;
int b = a;
```

Hier findet ihr gute Erklärungen, sowie schöne Bilder dazu:

`https://stackoverflow.com/questions/184710/what-is-the-difference-between-a-deep-copy-and-a-shallow-copy`





FYI: Rule of 3/5/0

↳ nur default constructors

"If a class defines any of the following then it should probably explicitly define all three: constructor, copy constructor, (copy) assignment operator."
`https://www.codementor.io/@sandesh87/the-rule-of-five-in-c-1pdgpzb04f`

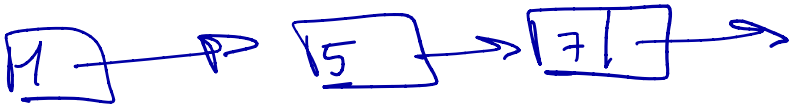
Wie gut fühlt Ihr Euch mit dem Stoff?

Siehe grobe Themenübersicht

- recursion
- references vs. pointers
- classes and structs
- function- and operator-overloading

Allg. Hinweis im Hinblick auf die Prüfung: Versucht wirklich möglichst nah an der Aufgabe zu bleiben. E.g. wenn eine Funktion etwas returnen soll und dann in der main dieser return Wert geprinted wird: versucht wirklich auch das richtige zu returnen und nicht einfach selber irgendwas zu printen.

Kleine Hinweise

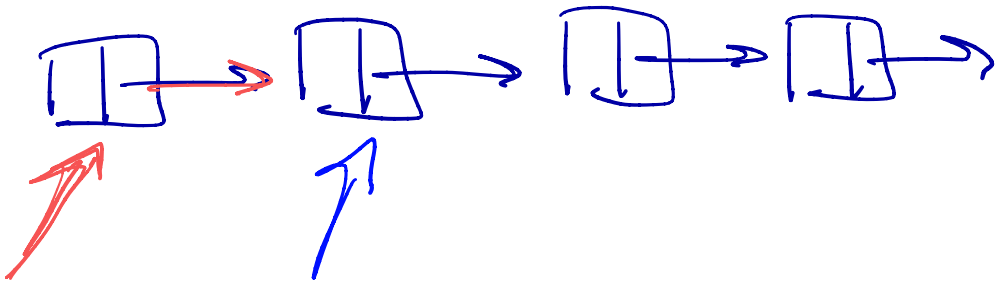



```
void sorted_list::add(int value){
  // this creates a new node in dynamic memory, wrapped in a shared ptr
  // analogously to new node(value) for normal pointers
  node_ptr newNode = std::make_shared<node>(value);

  node_ptr prev = nullptr;
  node_ptr n = first;
  while (n != nullptr && n->value < value){
    prev = n;
    n = n->next;
  }

  if (prev == nullptr){
    // TODO
  } else {
    // TODO
  }
}
```

Kleine Hinweise

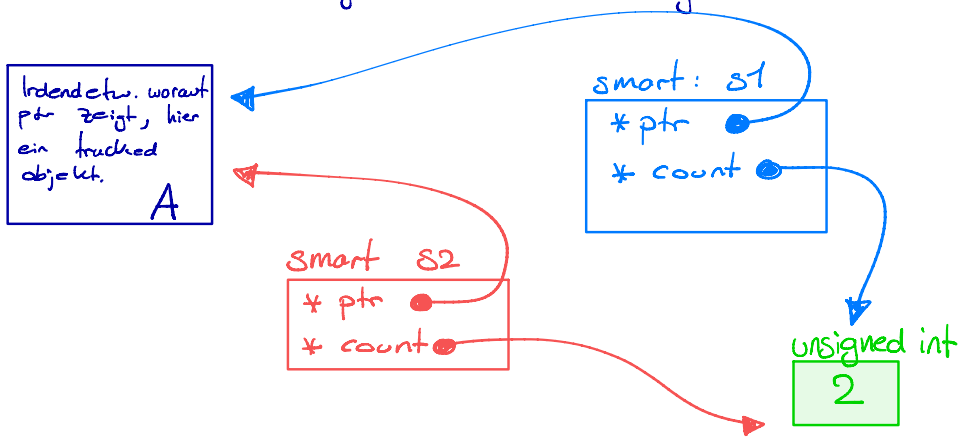


```
~List
{
    ListNode* current = startNode;
    ListNode* next;

    while (current != NULL) {
        next = current->next;
        delete current;
        current = next;
    }
}
```

https://codereview.stackexchange.com/questions/20472/destructor-for-a-linked-list

Generelle Idee hinter smart ptr: sie merken sich wieviele andere ptr auf das Gleiche zeigen wie sie selber. e.g.

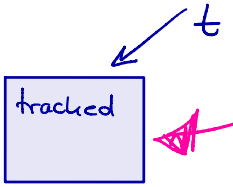


wenn mehr smart ptr auf A zeigen müssen wir das hier festhalten. Wenn ein ptr (mittels destr.) gelöscht wird auch.

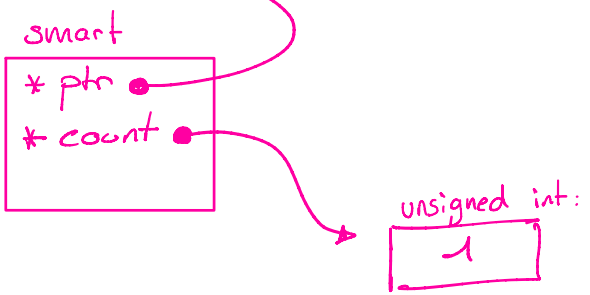
⚠ Falls ein ptr merkt, dass er der letzte ptr auf A ist, soll er auch A deallozieren (mit delete)

smart::smart(tracked * t) ist ein constructor i.e. hier wird ein neuer smart ptr gemacht. * ptr dieses neuen ptr soll auf das tracked zeigen, auf welches *t zeigt.
Ausserdem wollen wir uns merken, dass ein ptr auf ein Obj zeigt. Heisst: count soll auf neu alloziertes memory zeigen und dort im memory soll 1 stehen weil nur ein smart ptr zur Zeit auf dieses tracked zeigt.

wir kriegen:



wir wollen machen:



Wichtig: im constructor muss kein neuer smart ptr gemacht werden e.g. mit ... = new smart() weil dadurch, dass wir im constructor sind, sind wir ja dabei einen neuen smart ptr zu machen.

Eine Art, den copy constructor für den (copy) assignment operator zu benutzen:

① Ausgangslage

② → eine Kopie erstellen

③ → Pointer so anpassen, dass this auf das Objekt zeigt, auf welches src schon zeigt, hier also t