2. Klassen in C++

Um das Klassenkonzept ranken sich
alle wichtigen (oo) Konzepte:

- abstrakte Datentypen (Daten & Operationen)
- Zugriffsschutz
- nutzerdefinierte Operatoren
- Vererbung, Polymorphie & Virtualität
- generische Typen (Templates)

```
// Stack.h
#ifndef STACK_H
#define STACK_H
class Stack {
protected:
        int *data;
        int top, max;
public:
        Stack(int = 100);
        Stack(const Stack&);
        ~Stack();
        void push (int);
        int pop();
        int full() const;
        int empty() const;
};
#endif
```

prevents multiple inclusion !

ein neuer Typ !

Memberdaten

Memberfunktionen

Konstruktoren (u.u. viele)

Destruktor (einer !)

const Memberfunktion: Zusage, das Objekt nicht zu verändern

Zugriffsmodi

```
// Stack.cc
#include "Stack.h"
#include <cstdlib>

Stack::Stack(int dim): max(dim), top(0), data(new int[dim]) { }
Stack::Stack(const Stack & other) // Copy-Konstruktor
: max(other.max), top(other.top), data(new int[other.max]) {
        for (int i=0; i<top; ++i)
                data[i]=other.data[i];
}
Stack::~Stack() {
        delete [] data; // Feld statt einzelnem Objekt !
}
void Stack::push (int i) {
        if (!full()) data[top++]=i;
        else std::exit(-1);
} // if (!this->full()) this->data[this->top++]=i;
```

initializer list

NICHT: max

Scope resolution

```
int Stack::pop () {
        if (!empty()) return data[--top];
        else std::exit(-1);
}
int Stack::full() const { return top == max; }
int Stack::empty() const { return top == 0; }
```

- alternativ Memberfunktionen im Klassenkörper: dann implizit inline
- oder außerhalb des Klassenkörpers mit expliziter inline-Spezifikation (im Headerfile !)

```
// Nutzung:
#include "Stack.h"
void foo() {
Stack s1 (1000); s1.push(123);
Stack *sp = new Stack; sp->push(321);
delete sp; // ansonsten memory leak !
}
```

- Wann immer Objekte entstehen, läuft automatisch ein (passender) Konstruktor !
- Wann immer Objekte verschwinden, läuft automatisch der Destruktor !
- Klassen ohne nutzerdefinierten Konstruktor/Destruktor besitzen implizit
  - den sog. *default constructor* `X (){}`
  - den sog. *default copy-constructor* `X (const X&){ ... }` und
  - den sog. *default destruktor* `~X() {}`

memberweise Kopie !

- sobald nutzerdefinierte Konstruktor-Varianten vorliegen, gibt es nur noch den impliziten Copy-Konstruktor (wenn dieser nicht auch explizit definiert wird)

- Jedes Objekt enthält seine eigene Realisierung der Memberdaten (NICHT der Memberfunktionen!)
- Die Identität eines Objektes ist mit seiner Adresse verbunden !

```
bool Any::same(Any& other){
        return this == &other;
}
```

?

Beispiel: Jedes `Stack`-Objekt hat das folgende Layout unabhängig davon, wie es entstanden ist !

kein overhead durch Meta-Daten !

- es sind auch sog. unvollständige Klassendeklarationen erlaubt, von einer solchen Klasse können jedoch bis zu ihrer vollständigen Deklaration lediglich Zeiger & Referenzen benutzt werden:

```
class B;
class A {B * my_B; ....}; // oder ... class B* my_B;
class B {A * my_A; ....};
```

- strukturell identische Klassen mit verschiedenen Namen bilden verschiedene Typen (es gibt jedoch die Möglichkeit, nutzerdefiniert Kompatibilität herbeizuführen s.u.):

```
class X { public: int i; } x0;
class Y { public: int i; } y0;
X x1 = y0; Y y1 = x0; // beides falsch !!!
x0 = y0; y0 = x;      // beides falsch !!!
```

- Konstruktorparameter sind beim Anlegen von Objekten (geeignet) anzugeben, d.h es muss einen entsprechenden Konstruktor geben

```
// direct initialization:
X x0;                    // needs X::X();
X x1(1);                 // needs X::X(int);
X x2 = X(2,0);           // needs X::X(int,int);
X *pb = new X (5,true);  // needs X::X(int, bool);
X x3(1, "zwei", '3'); // needs X::X(int,[const] char*,char);

// copy initialization:
X x3 = 3;    // X tmp(3); X x3(tmp); ggf. elision
```

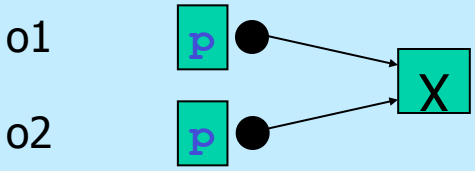
Copy-Konstruktoren

```
X::X(const X&); // kanonische Form !
```

shallow copy

(default copy ctor)

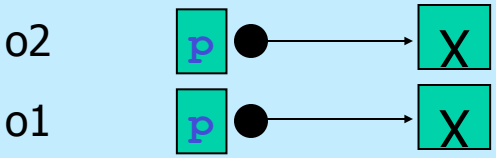
```
class SC {
   X* p;
public:
   SC(): p(new X) {}
};
SC o1;
SC o2=o1;
```



deep copy

(nutzerdefinierter copy ctor)

```
class DC {
   X* p;
public:
        DC(): p(new X) {}
        DC(const DC& src)
        : p(new X) {...copy X };
};
DC o1;
DC o2=o1;
```



- Konstruktoren können auch mit einem function try block implementiert werden, auch wenn ein passender handler vorliegt, wird die Ausnahme immer re-thrown !!!

```
struct Y {
   X* p;
   Y(int i) try : p(new X)
   { if (i) throw "huhh"; }
   catch(...)
   { /* delete p; NOT ALLOWED !!! */
      /* throw "huhh"; implicitly */}
   ~Y() { delete p; }
};
```

15.3 (10): Referring to any non-static member or base class of an object in the handler for a function-try-block of a constructor or destructor for that object results in undefined behavior.

http://www.gotw.ca/gotw/066.htm

```
{
  Parrot parrot;
}
```

In the above code, when does the object's lifetime begin? When does it end? Outside the object's lifetime, what is the status of the object? Finally, what would it mean if the constructor threw an exception? Take a moment to think about these questions before reading on.

Q: When does an object's lifetime begin?

A: When its constructor completes successfully and returns normally. That is, control reaches the end of the constructor body or an earlier return statement.

Q: When does an object's lifetime end?

A: When its destructor begins. That is, control reaches the beginning of the destructor body.

Q: What is the state of the object after its lifetime has ended?

A: As a well-known software guru ☺ once put it, speaking about a similar code fragment and anthropomorphically referring to the local object as a "he":

He's not pining! He's passed on! This parrot is no more! He has ceased to be! He's expired and gone to meet his maker! He's a stiff! Bereft of life, he rests in peace! If you hadn't nailed him to the perch he'd be pushing up the daisies! His metabolic processes are now history! He's off the twig! He's kicked the bucket, he's shuffled off his mortal coil, run down the curtain and joined the bleedin' choir invisible! THIS IS AN EX-PARROT! - Dr. M. Python, BMath, MASc, PhD (CompSci)

Referring to any non-static member or base class of an object in the handler for a function-try-block of a constructor or destructor for that object results in undefined behavior.

Leider wird dies von vielen Compilern dennoch erlaubt ☹ : g++, VisualStudio2005/2008

Initialisierung vs. Zuweisung:

= im Kontext einer Objektdeklaration: Initialisierung

```
X x = something; // initialize
```

= nicht im Kontext einer Objektdeklaration: Zuweisung

```
x = something; // assign !
```

```
class X {
  const int c;
public:
  X(int i): c(i) {} // ok, aber
  // X(int i) {c=i;} // falsch
};
```

Prefer initialization !