Objektorientierung in C

Alexander Droste

28. November 2013

Agenda

1. Motivation
   - Eigenschaften der Objektorientierung
   - Warum OOP in C?
2. Kapselung
3. Erstellen von Klassen
   - Abstrahieren neuer Datentypen
   - Erweiterung um Funktionen
4. Fortgeschrittene OO Techniken
   - Vererbung
   - Generische Funktionen

Eigenschaften der Objektorientierung

- Vererbung
- Polymorphie
- Kapselung
- Reduktion der Komplexität

- Produktivität?

International Software Benchmarking Standards Group:

Language	Hours Per Function Point
ASP*	6.1
Visual Basic	8.5
Java	10.6
SQL	10.8
C++	12.4
C	13.0
C#	15.5
COBOL	16.8
ABAP	19.9

Quellen: [1] [2] [3]

Warum OOP in C?

Es existieren viele OOP C-Sprachen.

- C++
- Objective-C
- C-Sharp

Gründe für OOP in C:

- Sprache auf Plattform nicht verfügbar
- Hardwarenah Programmieren
- Performance

Kapselung

onefile.c

```
typedef struct{int w, x;}
object_a;

typedef struct{long y, z;}
object_b;

object_a a = {0,1};
object_b b = {2,3};
```

- Alles in einer Datei -> keine Kapselung
- Alle Funktionen sowie Member der Structs sichtbar

onefile.c

```
int getSum(){
    return (a.w + a.x);
}

long getSub(){
    return (b.y - b.z);
}

long getValue(){
    return getSub();
}

void full_acces(){
    //a.w, b.z
    //getSub()
```

Kapselung

onefile.c

```
typedef struct{int w, x;}
object_a;

object_a a = {0,1};

int getSum(){
    return (a.w + a.x);
}
```

secondfile.c

```
typedef struct{long y, z;}
object_b;

object_b b = {2,3};

long getSub(){
    return (b.y - b.z);
}

long getValue(){
    return getSub();
}
```

- Auteilen in mehrere Dateien
- Kein Zugriff mehr

Kapselung

onefile.c

```
#include "secondfile.h"

//...

void acces(){
    long l = getValue();
}
```

- Schnittstelle im Header def.
- Zugriff über include

secondfile.h

```
long getValue();
```

secondfile.c

```
typedef struct{long y, z;}
object_b;

object_b b = {2,3};

long getSub(){
    return (b.y - b.z);
}

long getValue(){
    return getSub();
}
```

Kapselung

onefile.c

```
#include "secondfile.h"

//...

void acces(){
    extern long getSub();
    long l = getSub();
}
```

- Zugriff über extern
- Alle Variablen, Funktion von secondfile.c von außen erreichbar

secondfile.h

```
long getValue();
```

secondfile.c

```
typedef struct{long y, z;}
object_b;

object_b b = {2,3};

long getSub(){
    return (b.y - b.z);
}

long getValue(){
    return getSub();
}
```

Kapselung

onefile.c

```
#include "secondfile.h"

//...

void acces(){
//can only acces getValue()
}
```

- Static schützt vor Zugriff von außerhalb
- Schnittstelle beschränkt sich auf Header

secondfile.h

```
long getValue();
```

secondfile.c

```
typedef struct{long y, z;}
object_b;

static object_b b = {2,3};

static long getSub(){
    return (b.y - b.z);
}

long getValue(){
    return getSub();
}
```

Abstrahieren neuer Datentypen

- Eigene Datentypen durch typedef
- Schachtelung von eigenen Typen zu wieder Neuen

```
typedef struct{
    int x;
    int y;
}Position;
```

```
typedef struct{
    unsigned int width;
    unsigned int height;
}Size;
```

```
typedef struct{
    Position pos;
    Size size;
}Box;
```

```
typedef struct{
    Box box1;
    Box box2;
    Box box3;
}Lager;
```

Erweiterung um Funktionen

Deklarieren der Klasse:

```
typedef struct st StringType;

struct st{
    StringType* (* new) (char* text);
    void (* delete) (StringType * self);
    char* text;
};

StringType *String; //class pointer
```

- Struct werden Funktionspointer zugewiesen
- Verbund von Verhalten und Attributen

Erweiterung um Funktionen

Zuweisen der Funktionen:

```
void initClass(){
    String = calloc(1, sizeof(StringType));
    String->new = strCtor;
    String->delete = strDtor;
}
```

Implementation der Funktionen:

```
StringType* strCtor(char* text) {
    StringType* self = calloc(1, sizeof(StringType));
    self = String;
    //alloc text, copy string, return self
}
void strDtor(StringType* self) {
    free(self->text);
    free(self);
}
```

Erweiterung um Funktionen

Erzeugen der Objekte

- Speicher wird zunächst wie gewohnt reserviert
- Initialisierung durch Zuweisung des Klassenpointers
- Objektspezifische Initialisierung

Verwendung der Funktionen:

```
initClass();
StringType* newString = String->new("test");
```

- Klasse wird dynamisch initialisiert
- Funktion wird am Klassenpointer aufgerufen

Vererbung

Zieleigenschaften:

- Übertragen von Funktionen, Attributen auf Subklassen

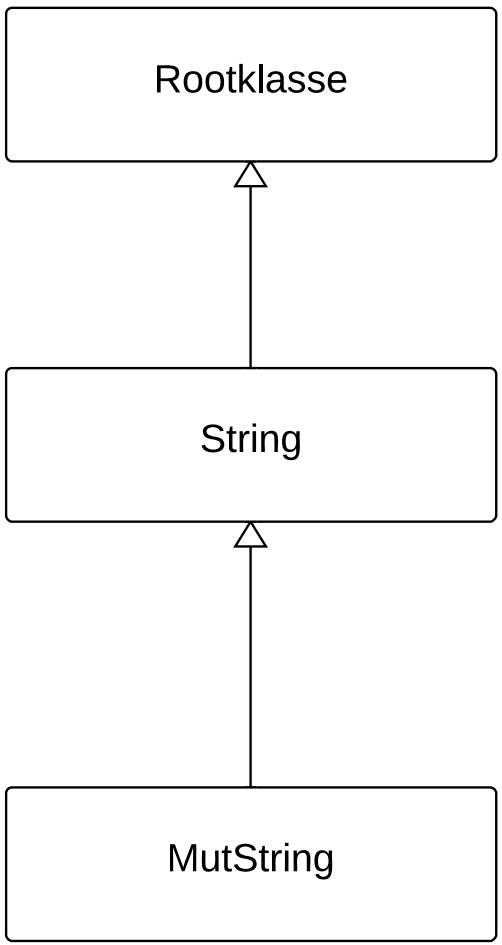
Ansatz:

- Funktionen den Struct-Typen als Pointer zuweisen

Problem:

- Abhängigkeiten zwischen Klassen innerhalb einer Hierachie

Vererbung - Strukturdiagramm



Rootklasse:

- Abstrakte Klasse
- Struct dient als Vorgabe für Subklassen

String:

- Fügt Attribut hinzu (char* text)
- Implementiert Kon-/Destruktor

MutString:

- Fügt Methode zum ändern des Strings hinzu
- Verwendet super-Aufruf in Kon-/Destruktor

Vererbung - Rootklasse

root.h

```
#define _self self->super_class

typedef struct cl Class;
struct cl{
/* all subclasses must define
   these values! */

    Class* super_class;
    void * (* new) (void* arg);
    void (* delete) (void* self);
};

//class-var
extern Class RootClass;
```

root.c

```
#include "Class.h"

Class RootClass = {
    NULL,
    NULL,
    NULL,
};
```

- Keine Superklasse, Kon-/Destruktoren
- Terminiert Hierachie
- Statisch erzeugt

Vererbung - Stringklasse

String.h

```
typedef struct{
    Class* super_class;
    void * (* new) (void* arg);
    void (* delete) (void* self);

    //additional attributes or functions
    char* text;
}StringClass;

extern StringClass String;
```

- Allgmeine Vorgabe für Klassen
- Superklasse, Funktionspointer angeben
  -> Statische Erzeugung, Zuweisung

Vererbung - Stringklasse

String.c

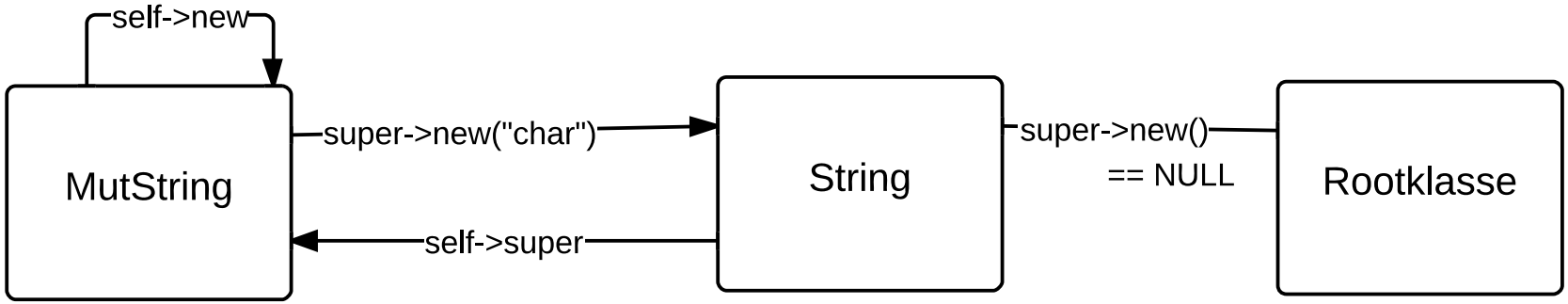
```
#define super String.super_class

static void strDtor(StringClass* self);
static StringClass * strCtor(char* text);

StringClass String = {
    &RootClass,      //assign SuperClass
    (void*)strCtor,  //concrete functions of
    (void*)strDtor,  //de-constructors
    NULL             //char* text
};
static StringClass * strCtor(char* text) {
    StringClass* self = NULL;
    if (super->new) _self = super->new(NULL);

    self = calloc(1, sizeof(StringClass));
    *self = String;//assign class values
    //init attributes... return self
}
```

Vererbung - SubStringklasse



MutString.c

```
//#define _self self->super_class =>Class.h

static void strDtor(MutStringClass* self) {
    if (super && super->delete) {
        super->delete(_self);
    }
    free(self);
}
```

Vererbung - Aufruf der Funktionen

client.c

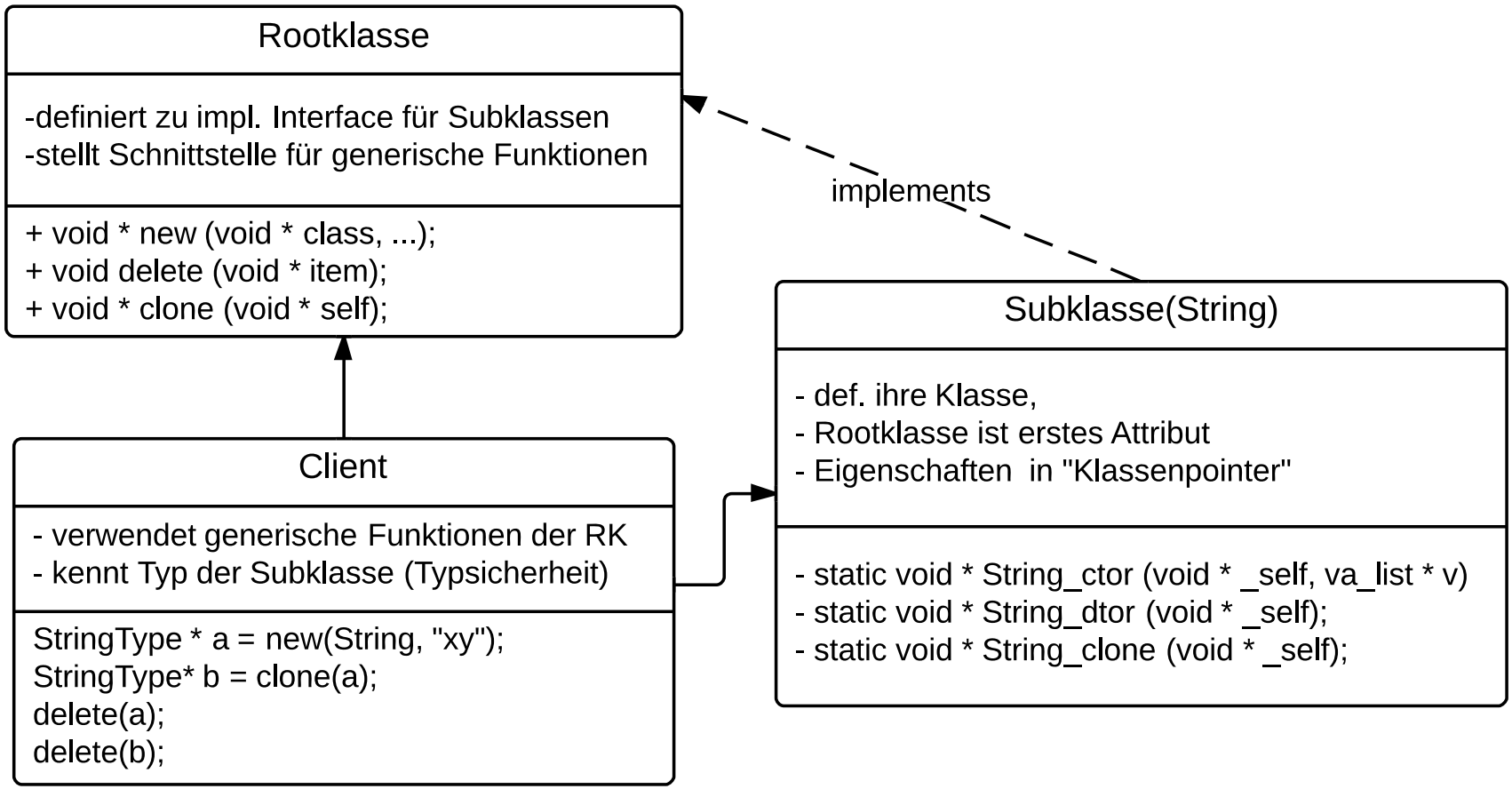
```
#include "MutString.h"

int main() {
    MutStringClass* c =  MutString.new("initial");
    MutString.changeString(c);
    MutString.delete(c);
}
```

- Öffentliche Schnittstelle durch Klassenpointer
- Funktionen werden am Klassenpointer aufgerufen
- Mut-String erbt De/Konstruktoren von String
  -> super->new(), super->delete() wird gerufen

Generische Funktionen - Beziehungsstruktur

Eine Schnittstelle für verschiedene Klassen:



Generische Funktionen - Interface/Schnittstelle

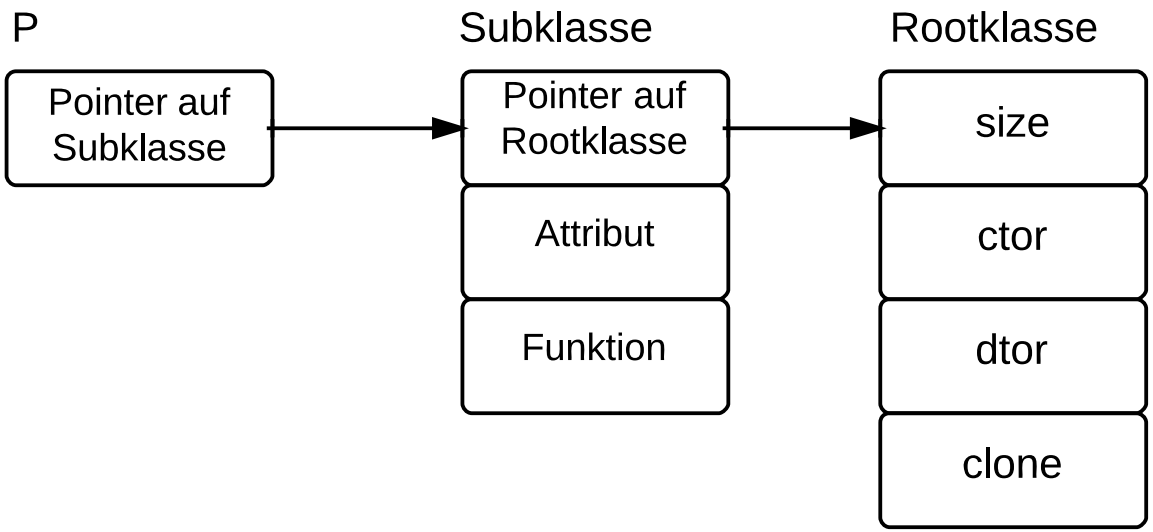
root.h

```
//zu implementierendes Interface
typedef struct {
    size_t size;
    void * (* ctor) (void * self, va_list * app);
    void * (* dtor) (void * self);
    void * (* clone) (void * self);
}Class;

//Schnittstelle zum Verwenden der Funktionen
void * new (void * class, ...);
void delete (void * item);
void * clone (void * self);
```

- Root-Struct definiert das zu impl. Interface
- An (Sub-)Klassen soll new(), delete(), clone() aufgerufen werden können.

Generische Funktionen - Pointerstruktur



- Erster Eintrag einer (Sub-)Klasse muss Pointer auf "Interface-Struct" (Rootklasse) sein

Generische Funktionen - Implementation

root.c - Generische new-Funktion:

```
void* new (void * _class, ...) {
    Class * class = _class;
    void *p = calloc(1, class -> size);
    * (Class **)p = class;

    if (class -> ctor) {
        va_list ap; va_start(ap, _class);
        p = class -> ctor(p, & ap);
        va_end(ap);
    }
    return p;
}
```

- Am Rootpointer werden size, ctor, etc. aufgerufen
- Funktionen werden in new(), delete() zur Laufzeit dynamisch gebunden
- Konstruktor mit variabler Argumentliste

Impl. der klassenspezifischen Funktionen

String.c - Implementation des Interfaces:

```
static void * String_ctor (void * _self, va_list* al){...}
static void * String_dtor (void * _self){...}
static void * String_clone (void * _self){...}

static Class _String = {
    sizeof(StringType),
    String_ctor,
    String_dtor,
    String_clone
};

//class pointer
void * String = & _String;
```

Generische Funktionen - Verwendung

String.h - Klasse sichtbar machen und Typ zur Verfügung stellen

```
extern void *String; //String-Class

typedef struct  {
    const void * class; //Root-Class
    char * text;
}StringType;
```

Client.c - verwendet die Generischen Funktionen

```
#include "String.h"
#include "root.h"

int useGenericFunctions() {
    StringType * a = new(String, "xy");
    StringType* b = clone(a);
    delete(a); delete(b);
    //...
}
```

Generische Funktionen - Zusammenfassung

- Rootklasse definiert Interface
- Subklassen implementieren das Interface
- Klienten verwenden Schnittstelle der Rootklasse, sowie Typ der spezifischen Klasse
  - Benutzung Generischer Funktionen + Typsicherheit
- jedes Objekt hat einen Pointer auf seine Klasse
  - Klasse beschreibt das spezifische Verhalten
- Unterschiedliches Verhalten bei verschiedenen Klassen
- Funktionalität wird zur Laufzeit gebunden (dynamisches Binden)

Zusammenfassung

- Öffentliche Schnittstelle immer im Header angeben
- Private Funktionen, Variablen durch static schützen

- Structs als Verbund von Attributen, Verhalten
- Verhalten wird durch Funktionspointer zugewiesen
- Klassen lassen sich statisch initialisieren
- Vorgaben zwischen Klassen innerhalb einer Hierachie müssen eingehalten werden
  -> Insbesondere Superklasse, Konstruktor, Destruktor
- Strenge Interfaces lassen sich nicht realisieren
- Aufruf der Funktionen am Klassenpointer
- Generische Funktionen durch einheitliches RootStruct

Quellen

Essentials of Programming Languages - D. Friedman, M. Wand

Programming Language Pragmatics - M. Scott

Object-Oriented Thought Process - M. Weisfeld

Concepts of Programming Languages – R. Sebesta

OOP with C – A. Schreiner

Learn C The Hard Way – http://c.learncodethehardway.org/book/ex19.html

Embedded –
http://www.embedded.com/electronics-blogs/object-oriented-c/4397794/Object-oriented-C-is-simple-

http://www.drdobbs.com/jvm/the-comparative-productivity-of-programm/240005881 [1]

https://de.wikipedia.org/wiki/Objektorientierte_Programmierung [2]

https://de.wikipedia.org/wiki/Objektorientierung [3]