Programmiertechnik 2

Unit 3b: Typen, Module, Klassen, Objekte

Ablauf

- Abstrakte Datentypen
- Konstruktion neuer Datentypen
- Charakteristika von Objekten
- Polymorphie und Vererbung
- Objekte in C
- Mehrfachvererbung, Interfaces, Exceptions

Abstrakte Datentypen (ADT)

- Beschreibung der Datentypen nicht auf Basis ihrer Repräsentation, sondern auf Basis ihrer Operationen und der Zusammenhänge zwischen diesen Operationen
- Algebraische Beschreibung: Typ wird durch Mengen, Konstruktoren, Terme und Axiome beschrieben
  - z.B. ACT ONE
- Bsp. Natürliche Zahlen in ACT ONE:
  - Operatoren: 0, succ(x), plus(x,y), minus(x,y)
    - Zahlen sind 0, succ(0), succ(succ(0)), ...
    - Terme sind z.B. plus(0, succ(minus(succ(0), 0)))
  - Axiome: Definition von äquivalenten Termen
    - succ(plus(x, y)) ≡ plus(x, succ(y))
    - plus(succ(x), y) ≡ plus(x, succ(y))
    - plus(0, y) ≡ y

Stack

- Stapelspeicher (LIFO)
  - Werte werden auf den Stapel gelegt und können wieder von ihm heruntergenommen werden
- Menge: Elementtype E, Stack SE
- Operatoren
  - emptyStack: → SE
  - push:SE × E→SE
  - pop:SE →SE
  - top:SE →E
  - empty: SE → boolean
- Axiome:∀e∈E∀s∈
  - top(push(s, e)) ≡ e
  - pop(push(s, e)) ≡ s
  - empty(emptyStack) ≡ true, empty(push(s, e)) ≡ false

Queue

- Warteschlange (FIFO):
  Werte werden am Ende an die Warteschlange angefügt;
  vorderste Elemente können entfernt werden
- Menge: Elemente E, Queue QE
- Operationen:
  - emptyQueue: → QE
  - enqueue:QE × E→QE
  - dequeue: QE → QE
  - head:QE →E
  - empty: QE → boolean

Objektorientierte Konstruktion neuer Datentypen

- Datentyp
  - Menge von Daten
  - Familie von Operationen
- Eingebaute Datentypen
  - Ganze Zahlen (char, short, int, long)
  - Reelle Zahlen (float, double)
  - Zeichenketten (char *, string)
- Klassen
  - Konstrukt in objektorientierten Sprachen
  - Repräsentiert Tupel gleicher Bauart
  - Ähnlich zu struct, record
  - Kann zusätzlich Operationen enthalten

Realisierung abstrakter Datentypen durch Klassen

- Klassen
  - Kapselung der Implementierungsdetails
- Operationen des ADT werden i.d.R. Methoden der Klasse
  - Ausnahme u.a. initiale Erzeugung des ADT
  - Instanzenmethoden vs. Klassenmethoden
- Objekte sind Instanzen von Klassen
  - Erzeugung
  - Initialisierung
  - Lebensdauer

Charakteristika von Objekten

- Zustand
  - Belegung von Objektattributen (members)
- Verhalten
  - Algorithmen, die auf Zustand operieren; Methoden
- Identität
  - “implizites Attribut”
  - Objekte, die ansonsten den gleichen Zustand haben (gleiche Attributwerte), sind evtl. trotzdem von verschiedener Identität

Darstellung von Methoden

- Imperative Programmierung:
  - Methoden sind Folgen von Aktionen
    - Darstellung der Aktionen in Quellcode, Zwischencode, Maschinencode
  - Methodenname
  - Eingabeparameter, Ergebnisse (Rückgabewerte, Ausgabewerte)
    - evtl. Definition von verschiedenen Methoden mit gleichem Namen, aber verschiedenen Parametertypen (Überladung)
    - impliziter Parameter “this”
- Offene Fragen:
  - virtuelle Methoden, Interfaces und spätes Binden
  - statische Methoden und Objektmethoden

Darstellung der Objektidentität

- Jede Objekterzeugung schafft neue Identität

```
new Foo(param1, param2);
```

- Objekte enthalten Verweise auf andere Objekte

```
class Bar {
  Foo f1;
  Foo f2;
};
```

- Variante 1: Objektidentität ist Adresse des Objekt (z.B. C++, Sun Hotspot JVM)
  - Erzeugung von Objekten alloziert freien Speicher; damit neuen Pointerwert
  - Annahme: Objektreferenzen beschränken sich auf einen Adressraum
- Variante 2: Objektidentität ist abstrakte "Kennung" (handle) (z.B. Sun Classic JVM)
  - Zugriff auf den Inhalt des Objekts erfordert Umwandlung der Kennung in tatsächlichen Pointer
  - Zugriff auf Objekte in anderen Adressräumen möglich

Abstrakte Datentypen in C vs. OOP

- Ein Datentyp ist eine Mengen von Daten zusammen mit einer Familie von Operationen
- abstrakter Datentyp:
  - Beschrieben wird lediglich die Menge und die Semantik der Operationen, nicht aber die interne Repräsentation der Daten oder die Implementierung der Operationen
  - In C: Trennung Deklaration (.h) und Implementierung (.c)
- Problem in C:
  - Viele Implementierungsdetails (Speicherlayout von Strukturen) offengelegt
  - Lösung: Zeiger auf echten Datentyp (opaque)
  - Verwendung von Zugriffsfunktionen
- Idee:
  - Daten und Zugriffsfunktionen bündeln → Objekte
  - Verwenden Funktionszeiger um Methoden an Objekte zu binden

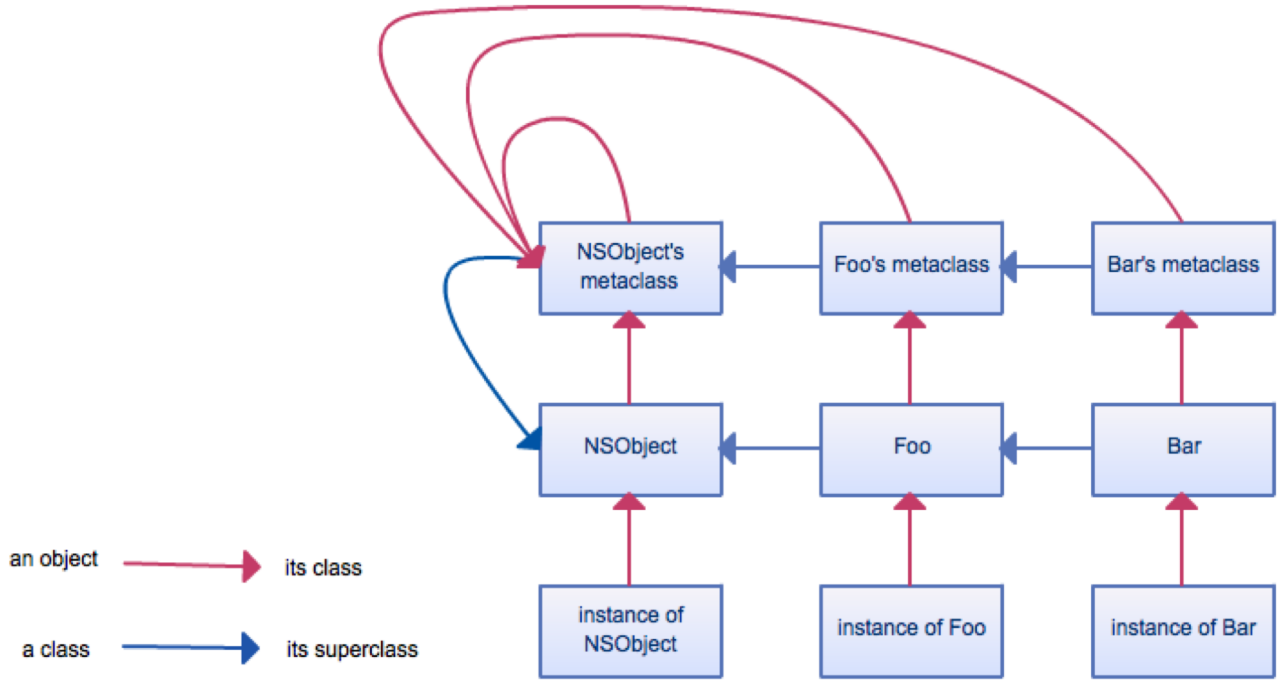
Datenkapselung
i2stack.h

Objektorientierung - Begriffe

- Klasse:
  - Beschreibung von Strukturen mit gleichen Eigenschaften (vergleichbar einem Datentyp mit wesentlich komplexeren Möglichkeiten);
  - eine Klasse zeichnet sich durch zwei Merkmale aus:
    - sie definiert eine Menge von Daten und kann Handlungen in Form von Methoden/Elementfunktionen (das sind Prozeduren und Funktionen, die der Klasse angehören) ausführen.
- Instanz: (einer Klasse)
  - anschaulich: eine Variable ,,vom Typ Klasse``
- Objekt:
  - Instanz einer Klasse; eine Menge von Daten (definiert in der Klasse) mit den Eigenschaften (Funktionen, Zugriffsrechte...) der Klasse, der es angehört
- Vererbung:
  - Weitergabe von Klasseneigenschaften an andere Klassen; ein Objekt ,,erbt`` zusätzlich zu seinen eigenen Eigenschaften die einer anderen Klasse

Objekte, Klassen, Metaklassen (ObjC)

- Jedes Objekt ist Instanz einer Klasse
- Klassen sind Objekte, welches ist deren Klasse?
  ...und welches ist deren Klasse...?



Klassenmethoden, Instanzenmethoden

- Klassenmethoden lassen sich ohne Übergabe einer Objektinstanz aufrufen
  - new()
  - Kein impliziter this oder self-Zeiger
  - Statische Methoden in C++
  - Sind in der Metaklasse implementiert
- Instanzenmethoden lassen sich nur mit einer Objektinstanz aufrufen
  - Objekt implizit übergeben
  - Bspw. zur Initialisierung von Objekten (isnew, Konstruktor)
    ...oder zur Zerstörung (Destruktor, finalizer)

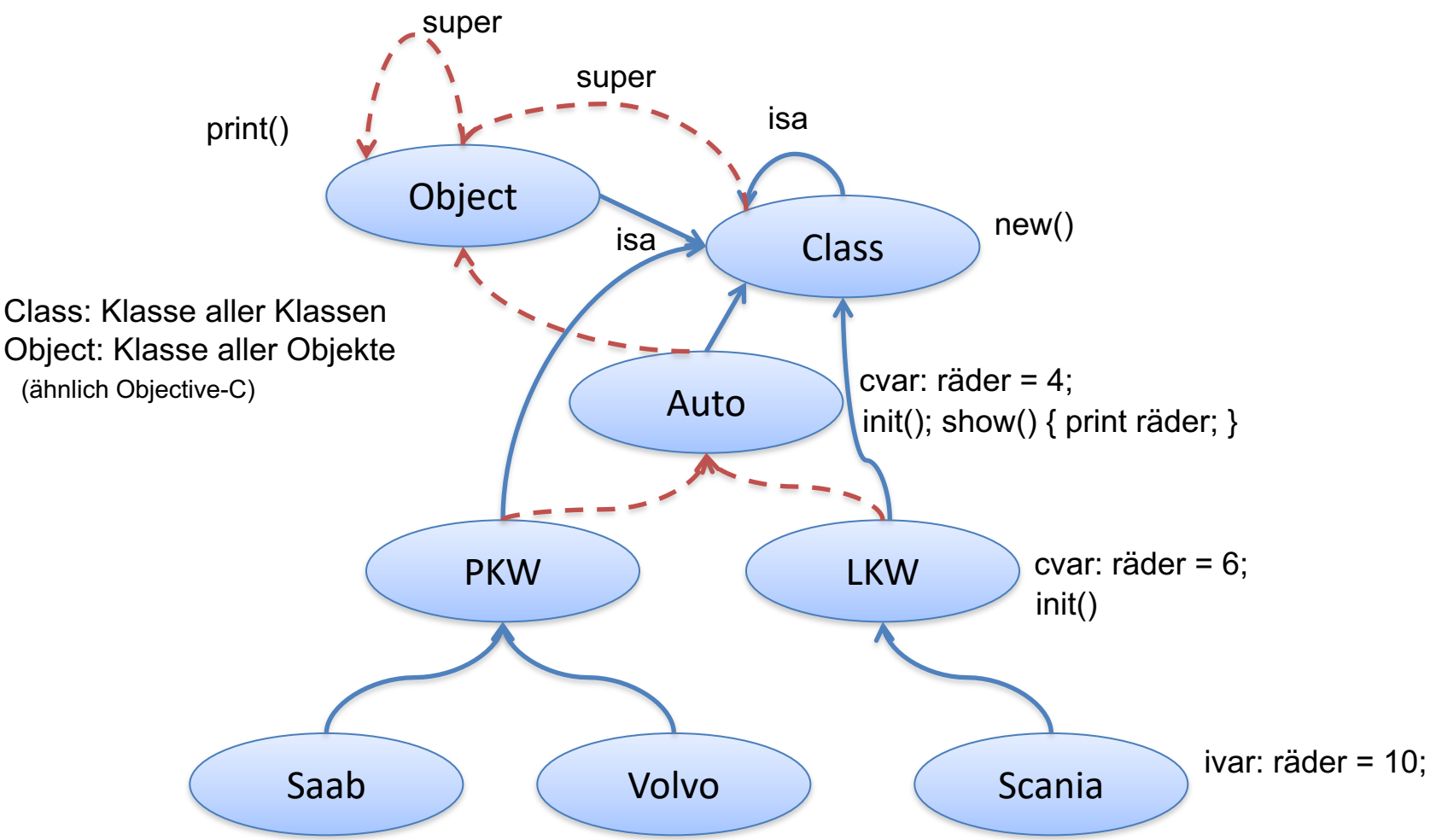
```
id obj = [ MyClass new ];
/* bewirkt impliziten Aufruf von  [ obj isnew ]; */
```

Klassenvariablen, Instanzenvariablen

- Klassenvariablen
  - Existieren einmal pro Klasse
  - Schlüsselwort static in C++ oder Java
  - Kein Objective-C-Sprachkonstrukt

- Instanzenvariablen
  - Existieren pro Objektinstanz
  - Können Klassenvariablen u.U. überdecken

Beispiel - Methodenaufruf



Polymorphie und spätes Binden

- Vielgestaltigkeit
- LSP: Liskov Substitution Principle
  - Barbara Liskov, Data Abstraction and Hierarchy, SIGPLAN Notices. 23(5), May 1988
  - Funktionen, die Referenzen auf die Basisklasse erwarten, sollen Exemplare der Ableitung verarbeiten können, ohne es zu wissen.
- If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

Spätes Binden

- Auffinden einer gerufenen Methode zum Zeitpunkt des Aufrufs, in Abhängigkeit des dynamischen Typs einer Variablen
- class Foo{
  public: virtual void bar();

```
};

// weitere Ableitungen
…
```

- // Verwendung

```
void foobar(Foo *foo){
    foo->bar();
}
```

Multi-Methoden

- Virtuelle Methoden: Ausgewählte Methode hängt nur von Zielobjekt des Aufrufs ab
- Multi-Methoden: Ausgewählte Methode hängt von den dynamischen Typen mehrerer Parameter ab
  - Number add(Number, Number); // abstrakt
  - Integer add(Integer, Integer);
  - Float add(Float, Float);
  - Float add(Float, Integer);
  - Float add(Integer, Float);

  - Number x = new Float(1.0);
  - Number y = new Integer(2);
  - Number z = add(x, y);
  - implementiert in CLOS (Common Lisp Object System), SDL-2000, Dylan

Realisierung von Vererbung

- Objekt-Layout: Festlegung der Offsets von Exemplarattributen innerhalb des Objektzustands
- Vererbung: neue Attribute werden zu der Klasse hinzugefügt
- LSP: Methoden der Basisklasse müssen auf Objekten so operieren können, als wären es Exemplare der Basisklasse
  - alle Attribute müssen in der Ableitung die gleichen Offsets haben wie in der Basisklasse
  - neue Attribute folgen im Layout den alten
  - Pointer auf Objekte sind dann gleichermaßen Pointer auf die Basisklasse und Pointer auf die Ableitung
- Mehrfachvererbung?

Realisierung von Methoden

- Problem:
  - Methoden operieren implizit auf “aktuellem” Objekt (C++, Java: zugänglich über “this”)

```
class X{
    int item;
    void foo(){
            item = 4;  // equivalent zu this->item = 4
    }
};
```

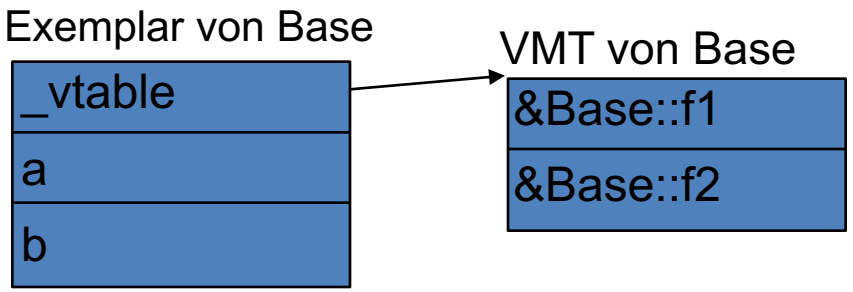
- Lösungsstrategie: Jede Methode hat impliziten ersten Parameter (Referenz auf das Objekt)
  - alle anderen Parameter verändern ihre Parameterposition

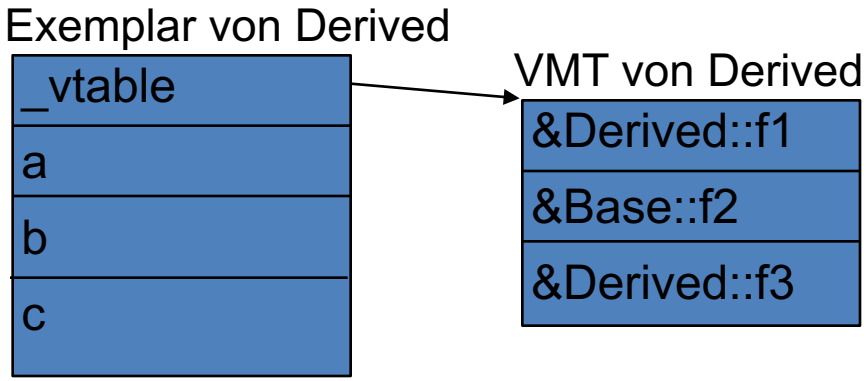
Realisierung von virtuellen Methoden

- Spätes Binden: zu rufende Methode ergibt sich aus dem Typ des Objekts
  - Jede Methode ist pro Klasse höchstens einmal vorhanden
- Funktionspointer: Adresse einer Funktion
- Virtuelle Methodentable (virtual method table, VMT): Feld/Struktur von Funktionspointern
  - eine VMT pro Klasse
  - virtuelle Methoden haben Nummern/Offsets in der Klasse

Virtuelle Methodentabellen

```
struct Base { // C++
    int a;
    double b;
    virtual void f1();
    virtual int f2(int);
};

struct Derived:public Base{
    char* c;
    virtual void f1();
    virtual void f3();
};
```





OOP in C

- Objekte sind Zeiger auf Strukturen
  - (void *) – generische Repräsentation für alle Objekte
  - Objekte besitzen Zeiger auf ihre Klassen (-objekte)
- Instanzenvariablen
  - Sind Elemente der Objektstrukturen
- Klassenobjekte
  - Enthalten Zeiger auf Methoden
    - Organisiert als Zeiger auf Methodentabelle (mit Funktionszeigern)
  - Zugriffsfunktionen für Objekten
    - Instanzenmethoden – funktionieren nur bei Existenz einer Objektinstanz
    - Klassenmethoden – zur Objekterzeugung

Beispiel: String-Objekte

```
/*     ( str_oo.h )        */

    #ifndef __STR_OO_H
    #define __STR_OO_H

    /* String-Definition: abstrakte Objekte */

    void * newString    (const char * text);
    void   deleteString (void * _string);
    void   setvalue     (void * _string, const char * text);
    char * getvalue     (const void * _string);
    void   cout         (const void * _string);

    #endif
```

Deklaration verrät keine Details über Implementierung

Vererbung in C

```
struct Base{ /* C */
    int a;
    double b;
};

struct Derived{
    struct Base _base;
    char* c;
};
```

Methoden in C

- Kein impliziter “this”-Parameter
  - this muss explizit sein
- Methode mit gleichem Namen in mehreren Klassen
  - Funktionen in C dürfen nur einmal definiert werden
  - Klassenname wird Teil des Funktionsnamens
  - Optional: Überladung; Parametertypen werden ebenfalls Teil des Funktionsnamens

```
void Base_f1(Base *this) /* C */
{
   //code
}
```

Funktionszeiger

	Deklaration	Typedef	Cast	Zuweisung
int	int a;	typedef int Count;	(int)wert	a=3;
Zeiger auf int	int *b;	typedef int* PCount;	(int*)wert	b=&a;
Funktion, die int zurückgibt	int c();	-	-	-
Funktion, die int erwartet und Zeiger auf int zurückgibt	int* d(int);	-		-
Zeiger auf Funktion, die int zurückgibt	int (*e)();	typedef int (*intfun)();	(int (*) ())wert	e = &c;
Zeiger auf Funktion, die int erwartet, und Zeiger auf int zurückgibt	int* (*f)(int);	typedef int* (*intfun2)(int);	(int* (*)(int))wert	f = &d;

Implementierung String-Objekte

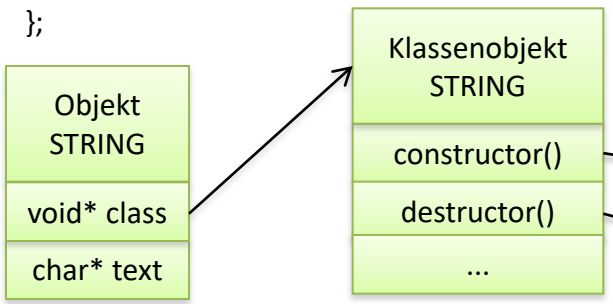
```
/* str_oo.c */
   #include <stdio.h>
   #include "str_oo.h"
   #include "error.h"

/* String-Implementation abstrakte Objekte */
   static const char STR_Rep;
/* die Repraesentation der Klasse STRING, einfach eine Adresse */
   static const void * STRING = & STR_Rep;

/* die Definition der Objektstruktur */
   struct String {
     void * class;
     char * text;
   };
```

Das Klassenobjekt soll alle Methodenzeiger enthalten

Implementierung Klasse STRING

```
/* Definition der Klasse */
struct Class_Str {

/* die grundlegenden Methoden, fuer alle Klassen gleich */
    size_t size;
    void * (*constructor) (void * self, const int argnr, va_list
      * app);
    void * (*destructor)  (void *self);
    void * (*copy)       (const void * self);
    void   (*cout)       (const void *self);
/* ab hier stehen die benutzerdefinierten Methoden */

    void   (*setvalue) (void * self, char * chars);
    char * (*getvalue) (void * self );
    void   (*setindex) (void * self, int index, const char c);
    char   (*getindex) (void * self, int index)
};
```

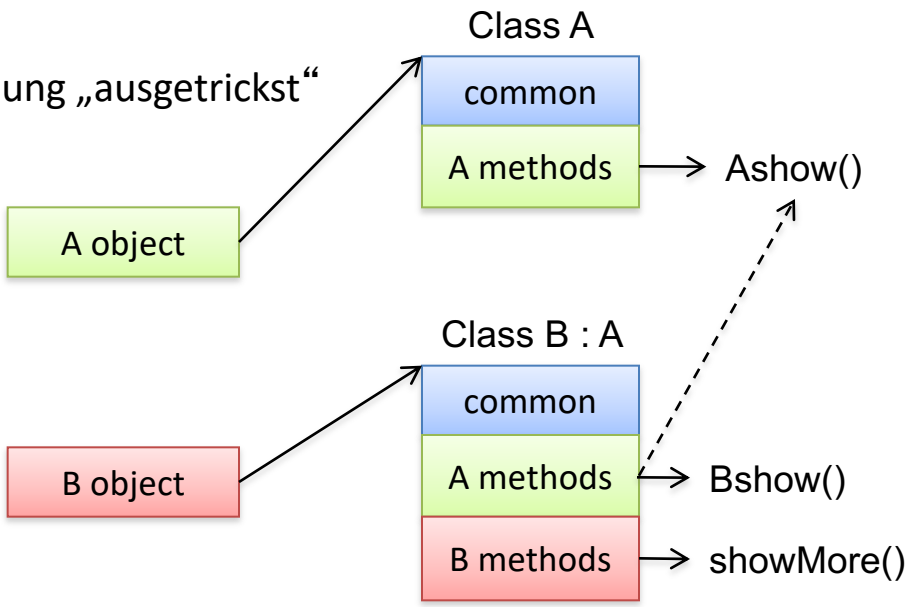
```
/* Deklarationen der Methoden von STRING */
static void * String_constructor (void * self, const int argnr, va_list *
      app);
static void * String_destructor  (void * self);
static void * String_copy       (const void * self);
static void   String_setvalue    (void * self , char * chars);
static char * String_getvalue    (void * self);
static void   String_setindex    (void * self, int index, const char c);
static char   String_getindex    (void * self, int index);
static void   String_cout       (const void * self);

/* Initialisierung des Klassenobjekts */
static const struct Class_Str _String = {
    sizeof(struct String), String_constructor,
    String_destructor, String_copy,
    String_cout, String_setvalue,
    String_getvalue, String_setindex,
    String_getindex
};
 const void * STRING = & _String;
```



Klassenhierarchien

- Zugriff auf Methoden erfolgt über Klassenobjekte
- Idee:
  - Klassenobjekte beginnen immer gleich
    - (Konstruktor, Destruktor, etc....)
  - ...können aber durch weitere Methoden erweitert werden
    - Strukturen werden „vergrößert“
    - C Compiler wird durch Typumwandlung „ausgetrickst“

p->show()



- Viele Details:
  - Vererbung vs. Überschreiben
  - Kopieren von Objekten
  - Referenz-Typen
- All das führt letztlich zu C++

Virtuelle Methodentabellen in C

- Struktur von Funktionszeigern:

```
struct Base_vtable_layout{ /* C */
    void (*f1)(struct Base*);
    int (*f2)(struct Base*, int);
};
struct Base_vtable_layout Base_vtable = {
    &Base_f1, &Base_f2
};
```

  – Problem: wie kann man vtable um neue Methoden verlängern?
- Feld von Funktionszeigern
  – Datentyp des Feldelements? void*

```
void *Base_vtable[] = {
    &Base_f1, &Base_f2
};
```

Makros

- Zwei Arten von Makros: parameterlos, mit Parameter

```
#define M1                          wert
#define M2(param1, param2)          param1 + param2
```

- Makro endet am Ende der Zeile
  - Backslash am Zeilenende bedeutet Fortsetzung des Makros

```
#define M3(param1, param2, param3) \
        param1 = param3;\
        param2 = param3;
```

- Verwendung von Makros: Text der Makrodefinition wird eingesetzt
  - Parameterbehaftete Makros müssen mit Klammern aufgerufen werden

```
M3(a, b, M1) // a=wert; b=wert;
```

  - Ersetzung auch in andern Makros und in #if

```
#if M1 > 100
  bedingtes Fragment
#endif
```

  - Ausnahme: “#ifdef M1” testet nur, ob Makro definiert ist; keine Ersetzung

Parameter in Makros

- Direkte Verwendung: Ersetzung

```
#define A(B, C)    B *x = new B[C]
A(int, 10);    // int *x = new int[10];
```

- Verwendung mit #: *Stringification*

```
#define      A(S)   char *S = #S
A(hello);      // char *hello = “hello”;
```

- Verwendung mit ##: Token-Verkettung

```
#define      A(S)   char *S##_text = #S
A(hello);      // char* hello_text = “hello”;
```

- Kann zur Synthese beispielsweise von VMT-Namen verwendet werden:
  - #define VMT(S) struct VTable S##_vtable = { &S##_f1, &S##_f2 }

Mehrfachvererbung

- Mehrfachvererbung: mehrere Basisklassen
  - Vereinigung von Zustand und Verhalten
  - Was passiert bei doppelten Methoden oder Attributen?
    - C++: bei Zugriff muss mittels der Basisklasse qualifiziert werden
    - o->Base1::member = value1;
    - o->Base2::member = value2;
  - Und was ist, wenn die Methoden virtuell sind?
    - C++: Für jede virtuelle Methode muss es einen "unique final overrider" geben
  - Implementierung: Layout der Basisklassen wird hintereinander gespeichert
    - Viele Probleme:
      - Konvertierung zwischen Klasse und Basisklasse erfordert Pointer-Korrektur
      - Virtuelle Basisklassen: Zustand soll nur einmal vorkommen
      - …

Finalisierung

- Bevor ein Objekt freigegeben wird, wird eine Finalisierungsroutine aufgerufen
  - explizite Speicherverwaltung (C++): Freigabe referenzierter Objekte
  - automatische Speicherverwaltung: Freigabe lediglich externer Ressourcen (etwa Schließen von Fenstern, Dateien, Datenbankverbindungen, …)
- erfordert Kenntnis aller freigegebenen Objekte
  - Copying GC: Traversierung des alten Halbraums
- Wiederherstellung der Erreichbarkeit in Finalisierung (*resurrection*)
  - Java: Finalisierung wird nur einmal ausgeführt

Interfaces

- Motivation: Ein Objekt soll in verschiedenen Rollen auftreten
  - Mehrfachvererbung gilt aber als problematisch
- Lösung: Eine Klasse darf eine Basisklasse besitzen, aber beliebig viele Schnittstellen (Interfaces) implementieren
  - Polymorphie: Konvertierung von Klassen in eine implementierte Schnittstelle ist typrichtig
- Implementierung: wie virtuelle Methoden
  - Exemplare verweisen auf mehrere virtuelle Methodentabellen (eine pro Schnittstelle)

Properties

- Kapselung: Implementationsdetails des Objekts sollen versteckt und Invarianten bewahrt werden
  - Zugang zu Zustand nur über Zugriffsmethoden
  - aber: Objekte haben oft “öffentlich bekannten Zustand”
- Lösung 1: Zugriffsmethoden (getter/setter)
- Lösung 2: Properties
  - Nutzer greifen auf Properties zu wie auf Attribute
    - p.name = p.name + “(Jr.)”
  - Property-Zugriff wird automatisch in Methodenruf umgesetzt
- Implementierungsstrategie:
  - C#: Kurznotation: Compiler ersetzt Property-Zugriffe “sofort”
  - Python: Dynamisch: bei Zugriff wird Property-Methode gesucht

Templates/Parametrierte Typen (*generic types*)

- Kein OO-Konzept im eigentlichen Sinne
- Implementierungsstrategie:
  - C++: Duplizieren des Template-Körpers für jeden Parametertypen
  - Java, C# (?): Parametrisierung zur Laufzeit (Erweiterung des Byte-Code-Formats um "generische" Byte-Codes)

Ausnahmebehandlung

- Kein OO-Konzept im eigentlichen Sinne
- Kernkonzepte:
  - Auslösen und Abfangen (Behandeln) von Ausnahmen
  - Hierarchien von Ausnahmen
- Weitere Konzepte
  - Wiederaufnahme nach beendeter Behandlung (Dylan)
  - Stack-Unwinding
    - Automatische Destruktorrufe für lokale Variablen (C++)
    - finally-Blöcke (Java, Python, C#)
  - Deklaration möglicher Ausnahmen für Funktionen
    - C++: optional
    - Java: vorgeschrieben

Ausnahmebehandlung (2)

- Exception-Sicherheit (safety): Alle Operationen funktionieren "richtig" auch falls eine Ausnahme auftritt
  - Commit or rollback: Operationen gelingen entweder vollständig oder scheitern vollständig
    - Rollback ist nicht immer implementierbar
  - Einhaltung von Invarianten
    - Temporär allozierte Resourcen werden auch bei einer Ausnahme wieder freigegeben (Invariante: Menge allozierter Resourcen wird von Methode nicht verändert)

Zusammenfassung

- Abstrakte Datentypen
- Konstruktion neuer Datentypen
- Charakteristika von Objekten
- Polymorphie und Vererbung
- Objekte in C
- Mehrfachvererbung, Interfaces, Exceptions