Fachhochschule
Münster University of
Applied Sciences

Embedded Software

Vererbung in C emulieren

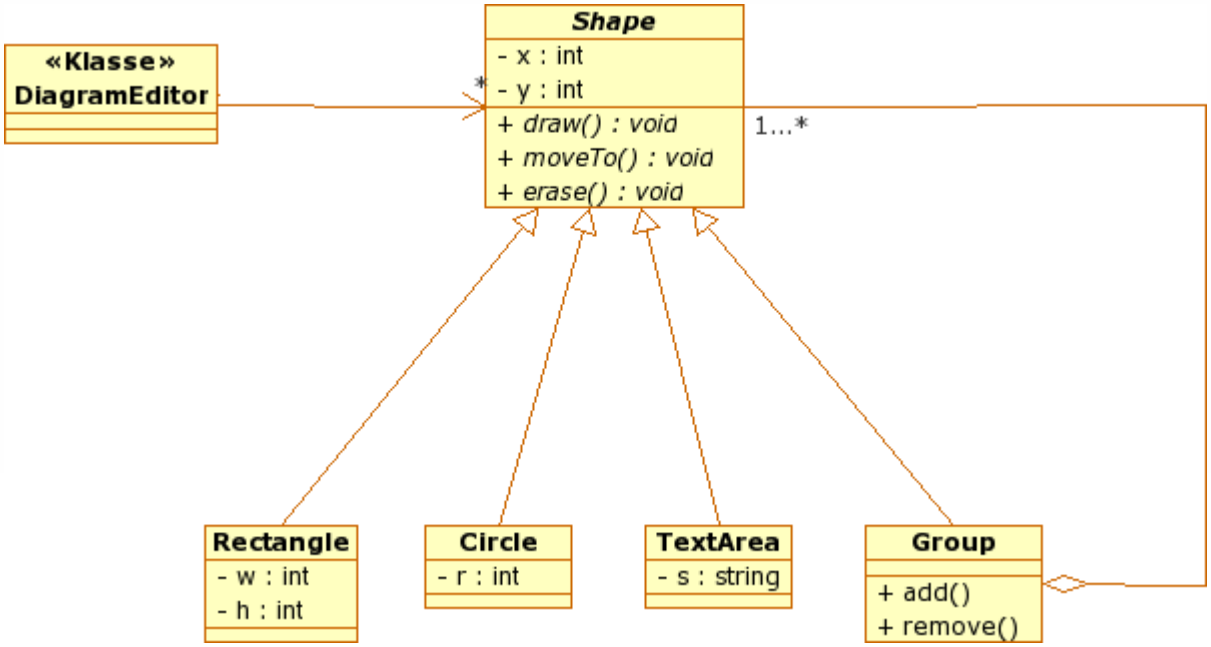
Vererbung in „C“ und gefährliches Casten

Prof. Dr. Nikolaus Wulff

Ein Diagramm Editor

- Ein Diagramm Editor soll generische 2D-Objekte zeichnen und verwalten können.
- Es gibt unterschiedliche 2D-Objekte, wie Rechtecke, Kreise, Textfelder, etc.
- Die 2D-Objekte werden durch den ADT *Shape* modelliert, der den abstrakten Oberbegriff und die gemeinsamen Attribute und Methoden bereitstellt.
- *Shapes* werden durch eine entsprechende C Struktur implementiert.

Diagramm Editor Design



Verkettung von Strukturen

- Die Struktur eines Shapes ist für alle geometrischen Objekte einheitlich:

Shape.h

```
#ifndef __SHAPE_H_
#define __SHAPE_H_
#include <assert.h>
#include <stdarg.h>

typedef struct shape_struct* Shape;

struct shape_struct {
      int x;
      int y;
      void (*draw) (Shape aShape);
};
```

Erweiterungen von Strukturen

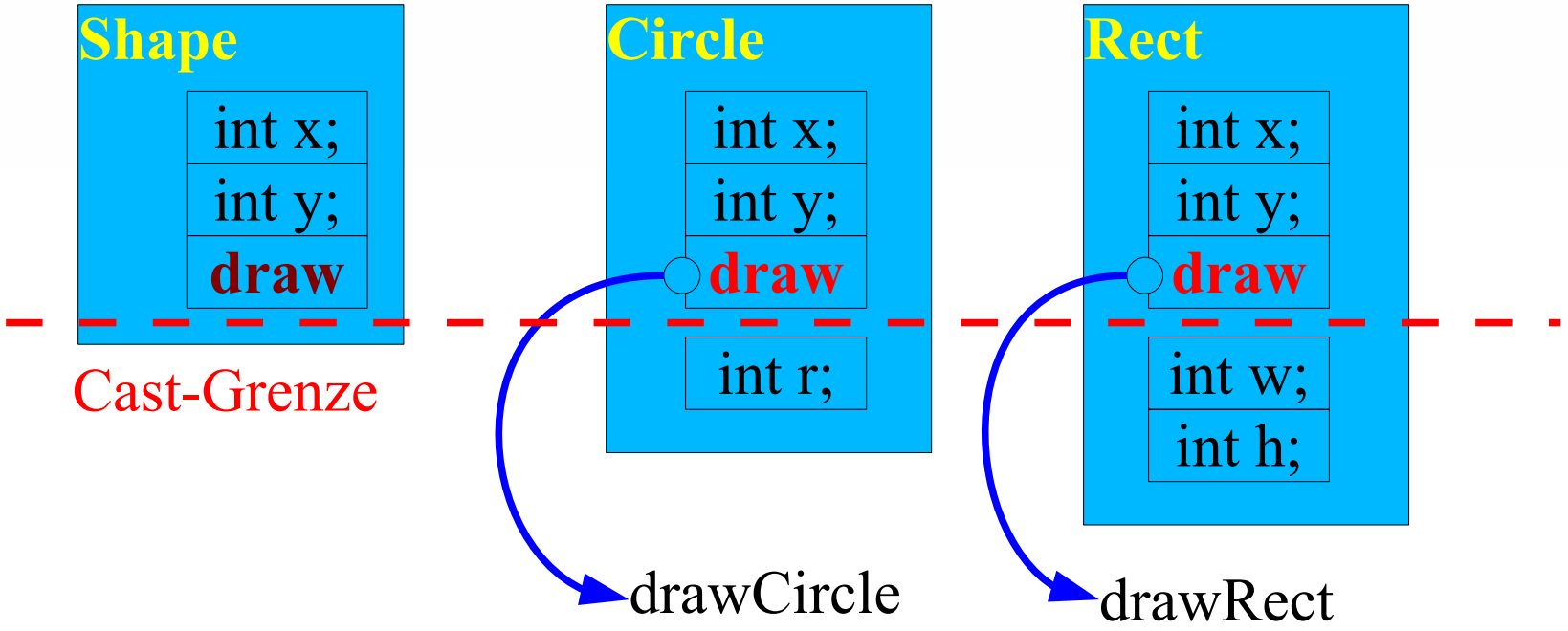
- Die zusätzlichen Argumente des Rechtecks werden dann „hinten angehängt“:

Rect.h

```
#ifndef __RECT_H_
#define __RECT_H_
#include "Shape.h"
typedef struct rect_struct* Rect;
struct rect_struct {
      int x;
      int y;
      void (*draw) (Shape aShape);
      int w;
      int h;
};
```

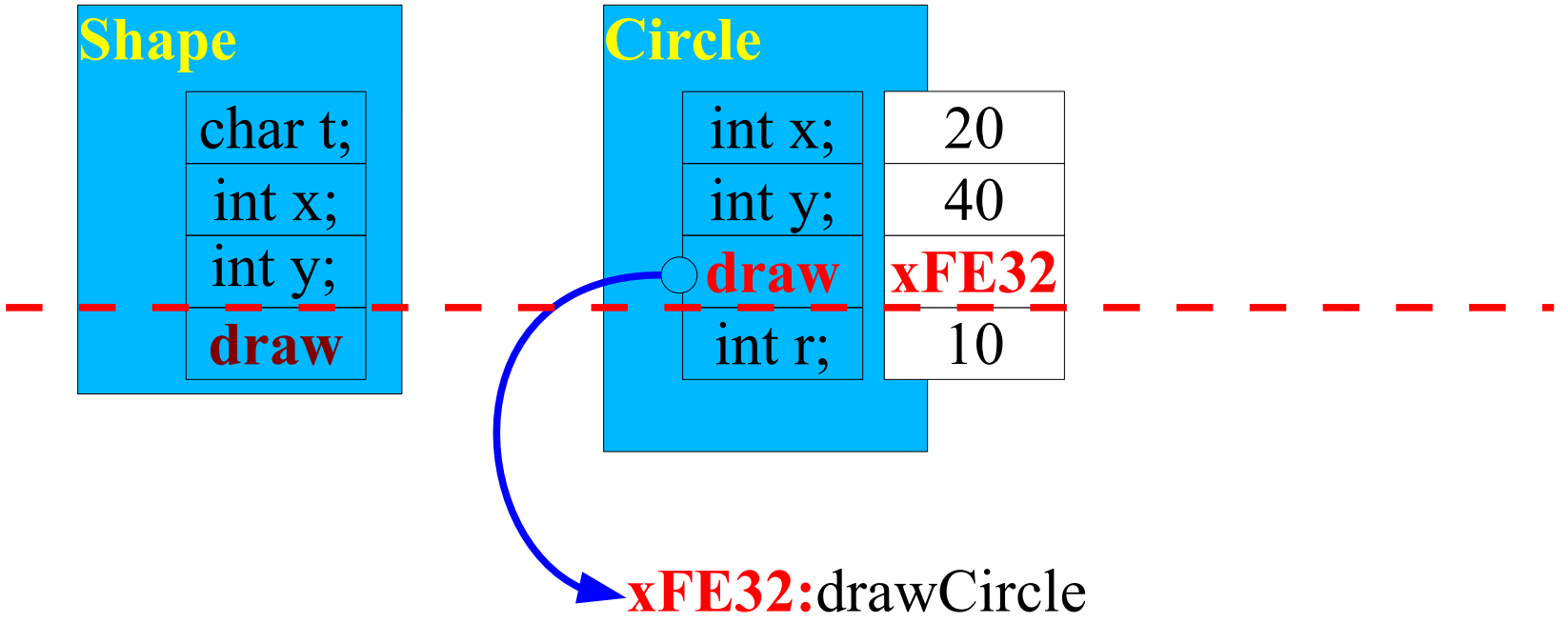
Casting von Strukturen

- Die unterschiedlichen Pointer von Shape Strukturen lassen sich gefahrlos aufeinander casten, solange die Strukturen zueinander kompatibel sind:



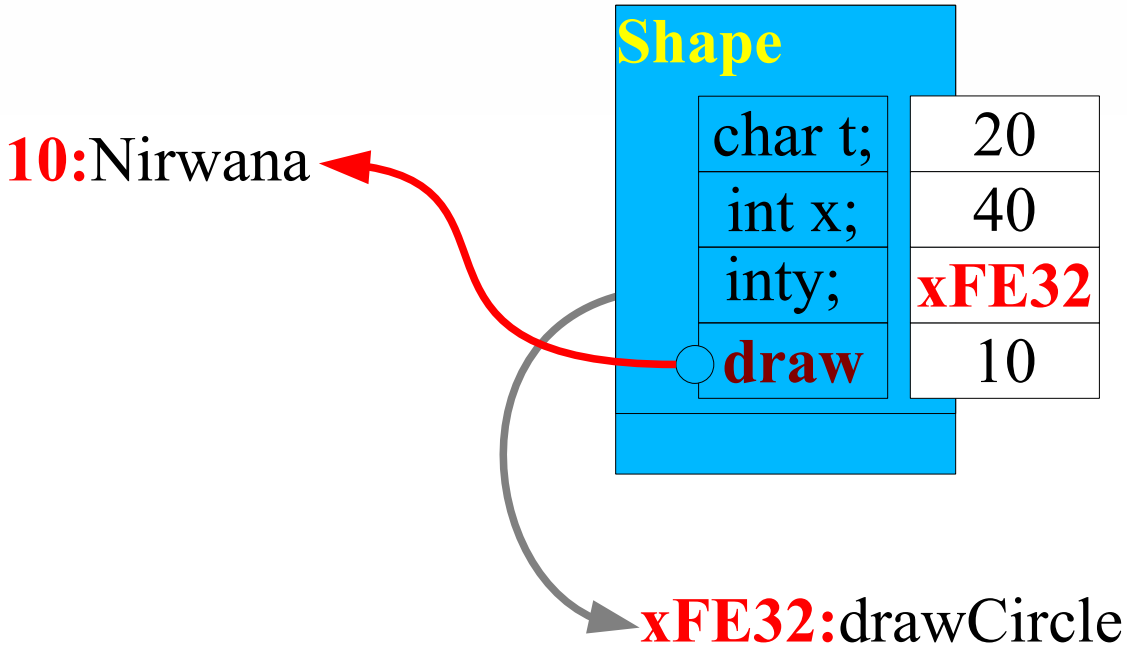
Gefährliches Casten

- Diese Art von Strukturvererbung ist zwar sehr elegant aber auch sehr gefährlich. Was ist wenn die Shape.h Struktur verändert wird ohne das Rect.h angepasst wird?



Gefährliches Casten

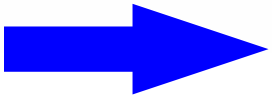
- Wird ein solches Rect Struct auf ein Shape gecasted so ist dies auf Grund der Strukturveränderung nicht mehr konform. Der Methoden Pointer zeigt in den Datenbereich und wird zu einem Absturz führen.



Automatisierung der „Vererbung“

- Gesucht wird eine Möglichkeit diese gefährlichen Seiteneffekte zu verhindern:

```
struct p2d {
    int x;
    int y;
};

struct p3d {
    int x;
    int y;
    int z;
};
```

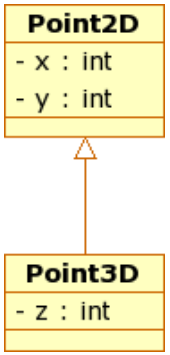


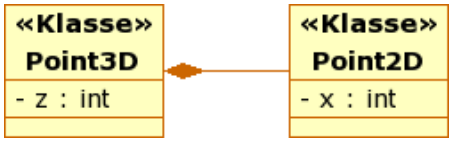
```
struct p2d {
    int x;
    int y;
};

struct p3d {
    struct p2d p2;
    int z;
};
```

- Die rechte Version garantiert, dass alle Änderungen von p2d automatisch auch für p3d gelten.

Vererbung versus Aggregation

- Leider drückt die rechte Seite nicht das Selbe wie die linke Seite aus. Wir versuchen mit C Vererbung zu emulieren.
- Die rechte Seite modelliert jedoch keine Vererbung sondern eine Aggregation!





Verwendung von Point3D

- Der Unterschied wird sehr schnell deutlich, wenn die rechte Variante instanziiert und verwendet wird:

```
struct p2d point2 = { 2, 4 };
struct p3d point3 = { {1, 4} , 5};
struct p3d another= { point2 , 5};

 point2.x = 1;
 point2.y = 0;

 point3.p2.x = 3;
 point3.p2.y = 4;
 point3.z = 3;
```

point3 enthält eine Instanz von p2d. Er ist jedoch kein Subtyp von p2d.

Automatisierung durch Macros

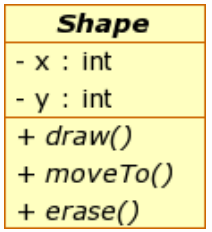
- Die einzige Chance hier weiter zu kommen ist es den C Präprozessor einzusetzen.

```
#define ATTRIBUTE_2D() \
        int x,y;

#define ATTRIBUTE_3D() \
        ATTRIBUTE_2D() \
        int z;

struct p2d {
    ATTRIBUTE_2D()
};

struct p3d {
    ATTRIBUTE_3D()
};
```

Änderungen an der 2D Struktur werden automatisch auch an die 3D Struktur propagiert.

Operationen

- Bis lang konzentrierte sich alles auf die Attribute der mit C-Structs modellierten Shapes. Was ist wenn weitere Methoden hinzukommen?




```
struct shape_struct {
    int x;
    int y;
    void (*draw)   (Shape aShape);
    void (*moveTo) (Shape aShape, int x, int y);
    void (*erase)  (Shape aShape);
};
```

- Die Shapes unterscheiden sich nur durch ihre Attribute. Alle Shapes vom selben Typ müssen die gleichen Operationen und Methodenpointer haben.

GTK Separationsansatz

- Das Linux Graphical Toolkit GTK ist ein gutes Beispiel für ein modulares Design.
- Es ist vollkommen in C programmiert, bietet jedoch eine objektorientierte API an.
- Jedes graphische Element ist als ein GtkWidget modelliert, dass entsprechende Methoden zur Verfügung stellt.
- Die Attribute und die zugehörigen Methoden werden in zwei getrennten Structs modelliert.
- => Der Speicherplatz für die Methoden wird nicht pro Instanz benötigt sondern geteilt.

Shapes revisited

- Die Aufteilung in Methoden und Attribute erfolgt in zwei Structs XXX_Instance und XXX_Class:

```
typedef struct shape_Instance* Shape;
typedef struct shape_Class {
    void (*draw)   (Shape aShape);
    void (*erase)  (Shape aShape);
    void (*moveTo) (Shape aShape, int x ,int y);

} ShapeClass;

struct shape_Instance {
    ShapeClass *class;
    int x;
    int y;
};
```

Instanz- und Class-Struct

- Die Class-Struktur wird pro ADT jeweils nur einmal benötigt und modelliert das gemeinsame Verhalten aller Instanzen der selben „Klasse“.
- Die Instanz-Struktur wird für jedes „Objekt einer Klasse“gesondert verwaltet und kapselt die Daten.
- Genau die selbe Aufteilung wird auch von der Java Virtuellen Maschine verwendet, die Instanzen und Klassen (mit den Methoden) getrennt verwaltet.