# Programmiertechnik II

## Polymorphie und spätes Binden

# Polymorphie

- 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
   }
};
```

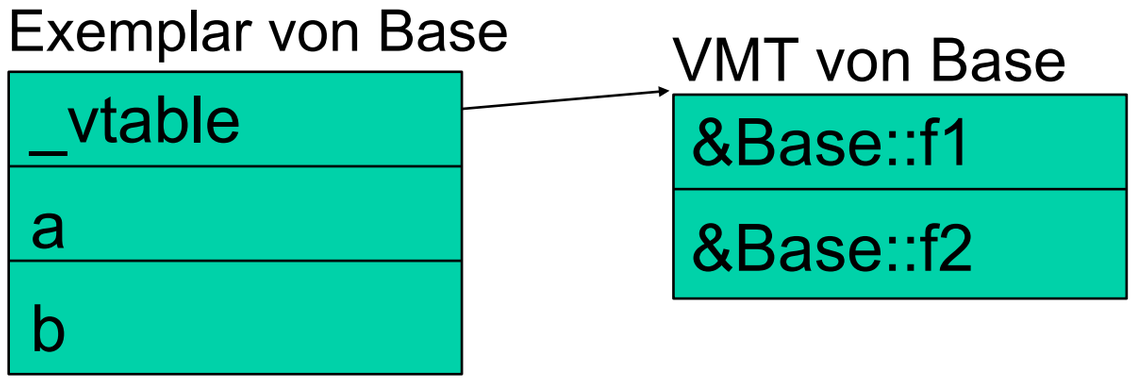
- Lösungsstrategie: Jede Methode hat impliziten ersten Parameter (Referenz auf das Objekt)
  - alle anderen Parameter verändern ihre Parameterposition

# Realisierung von virtuellen Methoden

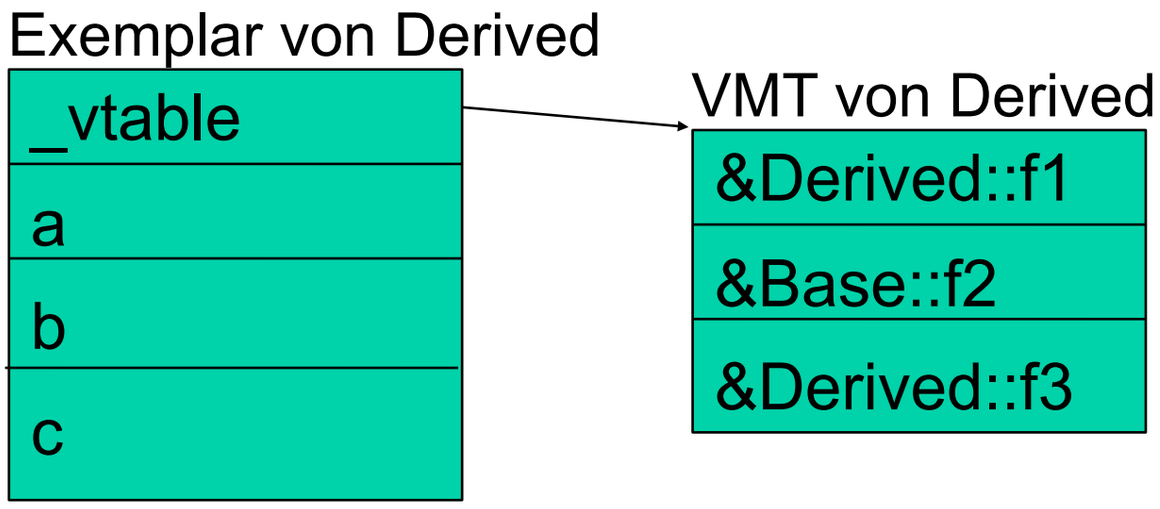
- Spätes Binden: zu rufende Methode ergibt sich aus dem Typ des Objekts
  - Jede Methode ist pro Klasse höchstens einmal vorhanden
- Funktionspointer: Adresse einer Funktion
- Virtuelle Methodentable (virtual method table, VMT): Feld/ Struktur von Funktionspointern
  - eine VMT pro Klasse
  - virtuelle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



# 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; |

# 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 }
```