Fachhochschule
Münster University of
Applied Sciences

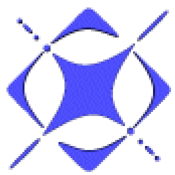
Embedded Software

Objektbasierte Entwicklung

Objektorientierung in „C“?

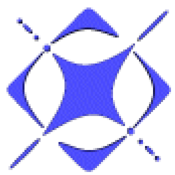
Prof. Dr. Nikolaus Wulff

Objektbasiert entwickeln

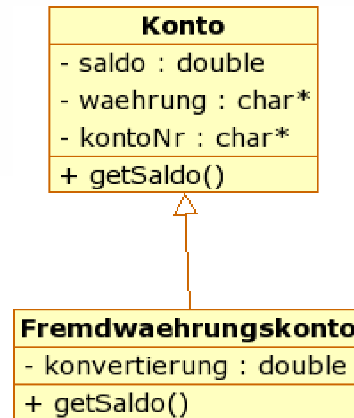


- Ohne C++ wird meist C im alten Stil programmiert.
  - => Ein endlose ***while***-Schleife mit C Funktionen.
- Eine Software Architektur in C erlaubt dennoch Module mit „einem Geheimnisprinzip“, private und öffentliche Methoden und die Abstraktion einer „Klasse“ mit instanziierten „Objekten“.
- Bei C handelt es sich nicht um echte Klassen, statt dessen werden diese mit C-***structs*** emuliert.
- Dieses Vorgehen ermöglicht es eine objektbasierte Anwendung mit entsprechenden Design Mustern und Abstraktionen zu entwickeln.

ADT in C

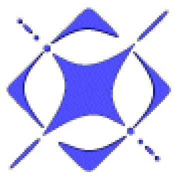


- Objekte mit gemeinsamen Attributen und Verhalten werden als Instanzen einer Klasse modelliert.
  - Beispiele: Konto, Kunde, Buchung etc.



- C kennt zwar keine Klassen, es lassen sich jedoch mit Strukturen entsprechende Abstrakte Datentypen definieren.

„Klassen“ in C



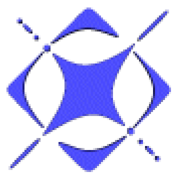
- Gemeinsame Attribute lassen sich in C mit Hilfe einer Struktur in einer *.h Datei definieren:

```c
struct konto_struct {
        char*  waehrung;
        char*  kontoNr;
        double saldo;
};
```

- Das gemeinsame Verhalten aller Konto Instanzen wird durch entsprechende Methoden abgebildet:

```c
extern double getSaldo(struct konto_struct* konto);
```

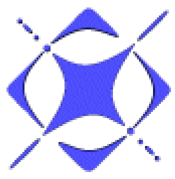
C Strukturen



- Die im Header deklarierten Strukturen und Methoden stellen die öffentliche Schnittstelle der modellierten Klasse da:
  - Zwei Attribute kontoNr, saldo sowie die Methode getSaldo.
  - Damit die Methode auf dem jeweiligen Konto operieren kann wird eine Referenz auf die Struktur des Kontos übergeben.
  - Die Implementierung in Konto.c ist trivial:

```
#include "Konto.h"
double getSaldo(struct konto_struct* konto) {
    return konto->saldo;
}
```

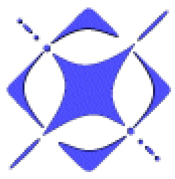
Verwendung der C „Klasse“



- Die Verwendung dieses ADT könnte so aussehen:

```
#include <stdio.h>
#include "Konto.h"

int main(int argc, char* argv[]) {

    struct konto_struct konto = {"Euro","08-a",10000};

    printf("Konto %s hat ein Saldo von %f %s\n",
            getKontoNr(&konto),
            getSaldo(&konto),
            getWaehrung(&konto)  );

    return 0;
}
```

Eigene Typen definieren



- Lästig ist die Pointer Arithmetik und das Offenlegen der Struktur. Hier erweist sich die Definition eines eigenen Typen per typedef als nützlich:

```
typedef struct konto_struct* Konto;
```

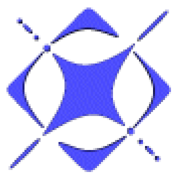
- Die Methode

```
double getSaldo(struct konto_struct* konto);
```

- kann dann prägnanter geschrieben werden als:

```
double getSaldo(Konto konto);
```

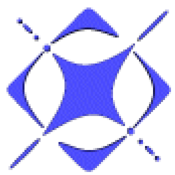
Verwendung des ADTs



```c
#include <stdio.h>
#include <stdlib.h>
#include "Konto.h"


int main(int argc, char* argv[]) {
   Konto konto = newKonto(EURO, "0815-567", 10000);

   printf("Konto %s hat ein Saldo von %f %s\n",
          getKontoNr(konto),
          getSaldo(konto),
          getWaehrung(konto) );

   free(konto);
   return 0;
}
```

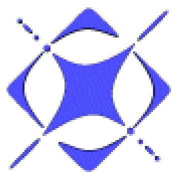
Konten sind jetzt Pointer und erfordern malloc und free!

„Klassen“ erweitern



- Die Methode getSaldo erscheint auf dem ersten Blick überflüssig, das Attribut saldo kann auch direkt der Struktur entnommen werden...
- Was ist jedoch bei unterschiedlichen Ausprägungen von Konto, z.B. einem Fremdwährungskonto bei dem der aktuelle Wert berechnet werden muss?
- Hierzu muss die Kontostruktur entsprechen erweitert werden.

Vererbung in C

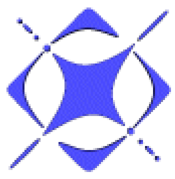


- Das Fremdwährungskonto ist eine Spezialisierung eines Kontos mit zusätzlichen Attributen und nach außen gleichem Verhalten:

```
typedef struct fremdWaehrungs_konto_struct {
    char*  waehrung;
    char*  kontoNr;
    double saldo;
    double konvertierung;
} *FremdKonto;
```

- Ein solches Fremdkonto kann wie ein Konto verwendet werden. Es muß jedoch die getSaldo Methode anders darauf reagieren.

Methoden erweitern

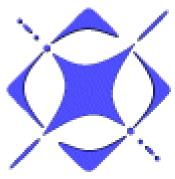


- Die Methode getSaldo reagiert je nach Kontoausprägung unterschiedlich:

```
double getSaldo(Konto konto) {
    double saldo = konto->saldo;
    if ( !strcmp(EURO, konto->waehrung))  {

       saldo *= ((FremdKonto)konto)->konvertierung;

     }
    return saldo;
}
```

- Handelt es sich um ein Fremdwährungskonto wird gecasted und entsprechend umgerechnet.

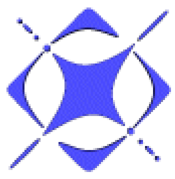
Weitere Verbesserungen



- Es ist sicherlich ungeschickt die Verzweigung ob es ein Fremdwaehrungskonto ist oder nicht an einem Stringvergleich fest zu machen.
- Besser ist hier z.B. ein Enum Type:

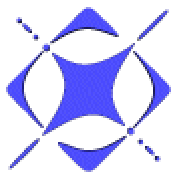
```
typedef enum {euro, foreign} Currency;
```

Das Methoden Problem



- Es stört, das die Methode getSaldo für jede mögliche Kontoart anders reagieren muss.
- Die Alternative zwei unterschiedliche Methoden getSaldo und *getSaldoFremdWaehrung* zu implementieren klingt auch nicht gut, denn der aufrufende Client sieht dann nicht mehr eine einheitliche Konto Schnittstelle.
- Hierzu bietet es sich an die Methoden zu überladen und an den jeweiligen Kontostrukturen zu verankern.
- C kennt hierzu das Konstrukt eines Pointers auf eine Methode.

Function Pointer

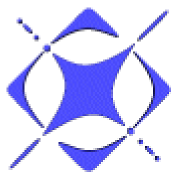


- Die aufzurufende Methode wird direkt an der Struktur notiert:

```
struct konto_struct {
        char*  waehrung;
        char*  kontoNr;
        double saldo;
        double (*getSaldo)(Konto konto);
};
```

- Beim Erzeugen der unterschiedlichen Konto Typen muß im newKonto Konstruktor der Pointer auf eine entsprechende Methode gesetzt werden.

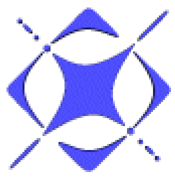
Überladene Methoden



- Die Methoden werden mit dem static Attribute auf private gesetzt und sind von außen nicht aufrufbar:

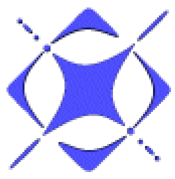
```
static double getEuroSaldo(Konto konto) {
    double saldo = konto->saldo;
    return saldo;
}

static double getForeignSaldo(Konto konto) {
    double saldo = konto->saldo;
    saldo *= ((FremdKonto)konto)->konvertierung;
    return saldo;
}
```

Ein C Konstruktor



```
Konto newKonto(char* waehrung, char* kontoNr, double saldo)
    Konto konto;
    if ( !strcmp(EURO, waehrung) ) {
       konto = (Konto) malloc(sizeof(struct konto_struct));
       konto->getSaldo = getEuroSaldo;
    } else {
       konto = (Konto) malloc(sizeof(struct
                         fremdWaehrungs_konto_struct));
       konto->getSaldo = getForeignSaldo;
    }
    konto->waehrung = waehrung;
    konto->kontoNr = kontoNr;
    konto->saldo = saldo;
    return konto;
}
```

Transparenter Methodenaufruf

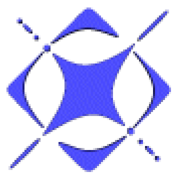


- Der Aufruf der unterschiedlichen Konten ist jetzt bis auf den Konstruktor identisch:

```
Konto konto = newKonto(EURO, "0815-567",10000);
Konto fremd = newKonto(USD,  "0845-56", 500);

printf(" EuroKonto %f \n", konto->getSaldo(konto) );
printf(" FremdKonto %f \n",fremd->getSaldo(fremd) );
```

- Beim Fremdkonto wird jedoch automatisch eine andere Implementierung der getSaldo Methode aufgerufen, ohne das der Aufrufer diese bemerkt!

Doppelte Verkettung



- Was stört, ist die doppelte Verwendung der Konto Struktur:

```
konto->getSaldo(konto);
```

- Zum einen enthält die Kontostruktur den Pointer auf die zu verwendende Methode, zum anderen benötigt diese Methode genau die Attribute der Struktur zum Berechnen des Saldos ...
- Hier kann der Präprozessor kosmetisch helfen:

```
#define actualSaldo(k) (k)->getSaldo((k))

printf(" Saldo von %f \n", actualSaldo(konto) );
```