2 Objektorientierte Programmierung in C

2.1 Beispiel: Bankkonten

2.1.1 Aufgabenstellung

- Definieren Sie eine Datenstruktur `Account` zur Repräsentation von Bankkonten mit Kontonummer, Kontoinhaber und aktuellem Kontostand!
- Implementieren Sie eine Funktion `newAccount`, die ein dynamisches Objekt des Typs `Account` erzeugt, es geeignet mit den als Parameter übergebenen Werten initialisiert und einen Zeiger auf das Objekt zurückliefert!
- Implementieren Sie Funktionen `number`, `holder` und `balance`, um die Kontonummer, den Kontoinhaber und den aktuellen Kontostand eines bestimmten Kontos abzufragen!
  (Benutzer des Typs `Account` sollen auf die entsprechenden Datenfelder nicht direkt zugreifen, weil sie z. B. nicht beliebig geändert werden dürfen; Geheimnisprinzip!)
- Implementieren Sie Funktionen `deposit` und `withdraw`, um den Kontostand eines bestimmten Kontos um einen bestimmten Betrag zu erhöhen bzw. zu erniedrigen, sowie eine Funktion `transfer`, um Geld von einem Konto auf ein anderes zu überweisen!

2.1.2 Datenstruktur

```
/* Zeichenkette variabler Länge. */
/* (»typedef« definiert den Namen »String« */
/* als Abkürzung für den Typ »const char*«.) */
typedef const char* String;

/* Konto. */
struct Account {
  long number;          /* Kontonummer. */
  String holder;        /* Kontoinhaber. */
  long balance;         /* Kontostand in Cent. */
};

/* Verwendung von long, weil int eventuell nur 16 Bit groß ist. */
```

2.1.3 Konten erzeugen und initialisieren

```
#include <stdlib.h>      /* malloc, exit, NULL */
#include <stdio.h>       /* printf */

/* Nächste zu vergebende Kontonummer. */
long nextNumber = 1;

/* Konto mit Inhaber h, eindeutiger Nummer */
/* und Anfangsbetrag 0 erzeugen. */
struct Account* newAccount (String h) {
  struct Account* this = malloc(sizeof(struct Account));
  if (this == NULL) {
    printf("newAccount: out of memory\n");
    exit(1);
  }
  this->number = nextNumber++;
  this->holder = h;
  this->balance = 0;
  return this;
}
```

2.1.4 Kontodaten abfragen

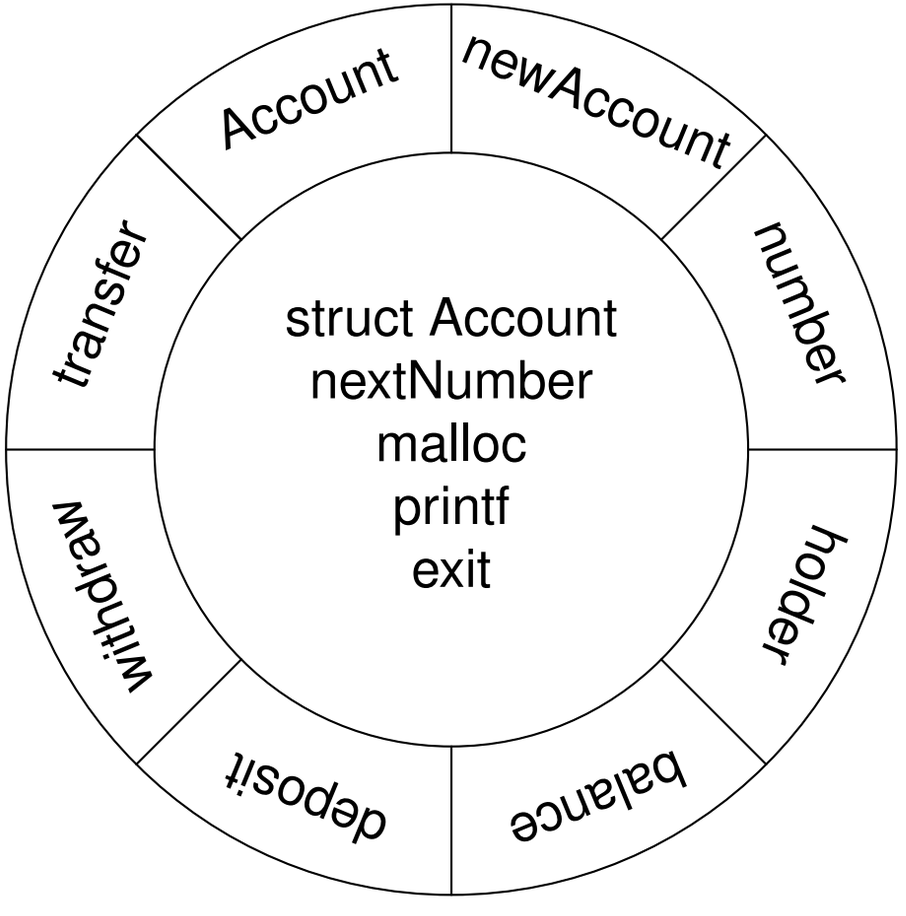
```
/* Nummer von Konto this liefern. */
long number (struct Account* this) {
  return this->number;
}

/* Inhaber von Konto this liefern. */
String holder (struct Account* this) {
  return this->holder;
}

/* Kontostand von Konto this liefern. */
long balance (struct Account* this) {
  return this->balance;
}
```

2.1.5 Geld einzahlen, abheben und überweisen

```
/* Betrag amount auf Konto this einzahlen. */
void deposit (struct Account* this, long amount) {
  this->balance += amount;
}

/* Betrag amount von Konto this abheben. */
void withdraw (struct Account* this, long amount) {
  this->balance -= amount;
}

/* Betrag amount von Konto this auf Konto that überweisen. */
void transfer (struct Account* this, long amount,
                                     struct Account* that) {
  withdraw(this, amount);
  deposit(that, amount);
}
```

2.1.6 Datenkapselung/Geheimnisprinzip

```
// Account als Abkürzung und Abstraktion von struct Account*,
// sodass Benutzer den Typ Account ohne Kenntnis seiner genauen
// Bedeutung verwenden können.
typedef struct Account* Account;
```



Benutzer „sehen“ und verwenden nur die Dinge am Rand der Kapsel, d. h. an ihrer Schnittstelle, aber nicht direkt die Dinge in ihrem Inneren.

2.1.7 Anwendungsbeispiel

```
int main () {
  /* Zwei Konten erzeugen. */
  Account a = newAccount("Max Mustermann");
  Account b = newAccount("Erika Mustermann");

  /* 1000 Cent auf Konto a einzahlen, */
  /* dann 300 Cent auf Konto b überweisen. */
  deposit(a, 1000);
  transfer(a, 300, b);

  /* Kontodaten ausgeben. */
  printf("Konto a: %ld, %s, %ld\n",
              number(a), holder(a), balance(a));
  printf("Konto b: %ld, %s, %ld\n",
              number(b), holder(b), balance(b));

  return 0;
}
```

2.2 Limitierte Konten

2.2.1 Aufgabenstellung

- Definieren Sie eine weitere Datenstruktur `LimitedAccount` zur Repräsentation von limitierten Konten, d. h. Konten, die nur bis zu einer bestimmten Kreditlinie überzogen werden dürfen!
- Implementieren Sie analog zu `newAccount` eine Funktion `newLimitedAccount`, die ein dynamisches Objekt des Typs `LimitedAccount` erzeugt und geeignet initialisiert!
- Implementieren Sie analog zu `number` etc. eine Funktion `limit`, um die Kreditlinie eines limitierten Kontos abzufragen!
- Sorgen Sie dafür, dass ein limitiertes Konto überall verwendet werden kann, wo ein allgemeines Konto erwartet wird! Insbesondere sollen alle für `Account` definierten Funktionen (wie z. B. `number` oder `transfer`) auch für `LimitedAccount`-Objekte aufrufbar sein!
- Redefinieren Sie die Funktionen `withdraw` und `transfer` für limitierte Konten dahingehend, dass die Operation nur dann ausgeführt wird, wenn dadurch die Kreditlinie nicht überschritten wird! (Andernfalls soll eine Fehlermeldung ausgegeben werden.)

2.2.2 Datenstruktur

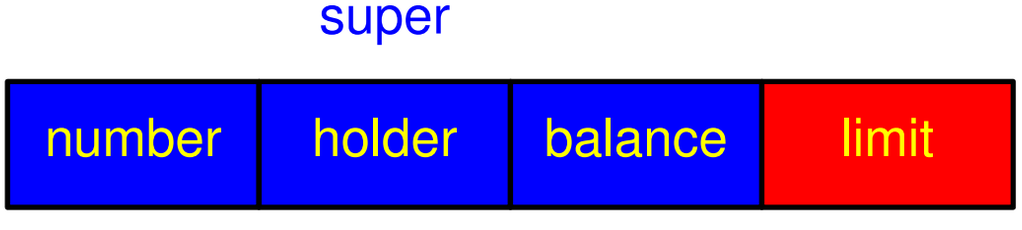
Problem

- `struct LimitedAccount` sollte von `struct Account` „erben“, d. h. alle seine Datenfelder besitzen, ohne dass deren Definition wiederholt werden muss.

Lösung

- `struct LimitedAccount` besitzt als erste Komponente ein „Teilobjekt“ des Typs `struct Account`:

```
/* Limitiertes Konto. */
struct LimitedAccount {
  struct Account super; /* Account-Daten. */
  long limit;           /* Kreditlinie in Cent (positiv). */
};
```



2.2.3 Limitierte Konten erzeugen und initialisieren

Problem

- `newLimitedAccount` muss, ebenso wie `newAccount`:
  - mittels `malloc` ein dynamisches Objekt erzeugen,
  - den Resultatwert von `malloc` überprüfen und ggf. eine Fehlermeldung ausgeben,
  - `number`, `holder` und `balance` initialisieren.
- Andererseits soll das „oberste Gebot der Programmierung“ beachtet werden:
  „Du sollst nicht Code verdoppeln!“

Lösung

- Die wiederverwendbaren Teile von `newAccount` werden in Hilfsfunktionen `newObject` und `initAccount` ausgelagert:

```
/* Dynamisches Objekt der Größe n Byte erzeugen. */
void* newObject (int n) {
  void* this = malloc(n);
  if (this == NULL) {
    printf("newObject: out of memory\n");
    exit(1);
  }
  return this;
}
```

```
/* Nächste zu vergebende Kontonummer. */
long nextNumber = 1;

/* Konto this mit Inhaber h, eindeutiger Nummer */
/* und Anfangsbetrag 0 initialisieren. */
void initAccount (struct Account* this, String h) {
  this->number = nextNumber++;
  this->holder = h;
  this->balance = 0;
}

/* Konto mit Inhaber h, eindeutiger Nummer */
/* und Anfangsbetrag 0 erzeugen. */
struct Account* newAccount (String h) {
  struct Account* this = newObject(sizeof(struct Account));
  initAccount(this, h);
  return this;
}
```

- Die Initialisierungen von `newLimitedAccount` werden ebenfalls in eine Hilfsfunktion `initLimitedAccount` ausgelagert, die ihrerseits `initAccount` (mit Typumwandlung von `struct LimitedAccount*` nach `struct Account*`) aufruft:

```
/* Limitiertes Konto this mit Inhaber h, Kreditlinie l, */
/* eindeutiger Nummer und Anfangsbetrag 0 initialisieren. */
void initLimitedAccount (struct LimitedAccount* this,
                                              String h, long l) {
  initAccount((struct Account*)this, h);
  this->limit = l;
}

/* Limitiertes Konto mit Inhaber h, Kreditlinie l, */
/* eindeutiger Nummer und Anfangsbetrag 0 erzeugen. */
struct LimitedAccount* newLimitedAccount (String h, long l) {
  struct LimitedAccount* this =
                        newObject(sizeof(struct LimitedAccount));
  initLimitedAccount(this, h, l);
  return this;
}
```

2.2.4 Kreditlinie abfragen

```
/* Kreditlinie des limitierten Kontos this liefern. */
long limit (struct LimitedAccount* this) {
  return this->limit;
}
```

2.2.5 Ersetzbarkeit von Konten durch limitierte Konten

Problem

- Obwohl ein Zeiger auf ein `LimitedAccount`-Objekt technisch gleichzeitig ein Zeiger auf das `Account`-Teilobjekt `super` dieses Objekts ist, sind die entsprechenden Zeigertypen logisch verschieden, sodass ein Zeigerwert des Typs `struct LimitedAccount*` nicht an eine Variable oder einen Parameter des Typs `struct Account*` zugewiesen werden kann.

- Daher können die für `Account` definierten Funktionen (wie z. B. `number` oder `transfer`) nicht für `LimitedAccount`-Objekte aufgerufen werden.

Lösung

- Die Zeigertypen `Account` und `LimitedAccount` werden beide als Synonyme des generischen Zeigertyps `void*` definiert:

```
typedef void* Account;
typedef void* LimitedAccount;
```

- Da `void*` mit jedem anderen Zeigertyp kompatibel ist, können nun sowohl `Account`- als auch `LimitedAccount`-Objekte an Parameter des Typs `struct Account*` übergeben werden.

- Daher können die für `Account` definierten Funktionen – die jeweils Parameter des Typ `struct Account*` besitzen – sowohl für `Account`- als auch für `LimitedAccount`-Objekte aufgerufen werden.

2.2.6 Exkurs zu Funktionszeigern

Prinzip

- So wie gewöhnliche Zeiger die Adresse eines Datenobjekts im Speicher enthalten, enthalten Funktionszeiger die Adresse (des Codes) einer Funktion im Speicher.
- Ein solcher Funktionszeiger kann wie ein Funktionsname verwendet werden, um die Funktion aufzurufen, auf die er momentan zeigt.
- Umgekehrt kann ein Funktionsname als Funktionszeiger verwendet werden, um einer Variablen oder einem Parameter mit Funktionszeigertyp die entsprechende Funktion zuzuweisen.
- Damit kann eine solche Variable oder ein solcher Parameter während der Laufzeit eines Programms bei Bedarf auf unterschiedliche Funktionen zeigen.

Beispiel

- Ausgabe einer Wertetabelle einer beliebigen Funktion,
  die als Parameter übergeben wird

```
#include <stdio.h>

/* Wertetabelle der Funktion f mit Parameter- und Resultattyp */
/* double für x von x1 bis x2 mit Schrittweite dx ausgeben. */
void print (double (*f) (double),
                              double x1, double x2, double dx) {
  double x;
  for (x = x1; x <= x2; x += dx) printf("%lg\t%lg\n", x, f(x));
}

/* Zwei exemplarische Funktionen: x*x und 1/x. */
double f1 (double x) { return x*x; }
double f2 (double x) { return 1/x; }

/* Testprogramm. */
int main () {
  /* Wertetabellen von x*x und 1/x ausgeben. */
  print(f1, 1, 10, 1);
  print(f2, 1, 10, 1);
  return 0;
}
```

Funktionszeiger in Strukturen

- Funktionszeiger können auch Teil von Strukturen sein.
- Damit können die einzelnen Objekte eines Strukturtyps nicht nur unterschiedliche Daten enthalten, sondern auch unterschiedliches Verhalten besitzen.
- Zum Beispiel:

```
/* Geometrisches Objekt. */
typedef struct Figure* Figure;
struct Figure {
  double width, height;   /* Breite und Höhe des Objekts. */
  void (*print) (Figure); /* Funktion zur Ausgabe des Objekts. */
};

/* Rechteck f ausgeben. */
void printRectangle (Figure f) {
  printf("Rechteck mit Seiten %lf und %lf\n",
                                f->width, f->height);
}
```

```
/* Ellipse f ausgeben. */
void printEllipse (Figure f) {
  printf("Ellipse mit Halbachsen %lf und %lf\n",
                                  f->width/2, f->height/2);
}

/* Geometrisches Objekt mit Breite w, Höhe h */
/* und Ausgabefunktion p erzeugen. */
Figure newFigure (double w, double h, void (*p) (Figure)) {
  Figure f = newObject(sizeof(struct Figure));
  f->width = w;
  f->height = h;
  f->print = p;
  return f;
}

/* Rechteck mit Breite w und Höhe h erzeugen. */
Figure newRectangle (double w, double h) {
  return newFigure(w, h, printRectangle);
}
```

```
/* Ellipse mit Halbachsen a und b erzeugen. */
Figure newEllipse (double a, double b) {
  return newFigure(2*a, 2*b, printEllipse);
}

/* Testprogramm. */
int main () {
  /* Reihe mit unterschiedlichen geometrischen Objekten. */
  Figure fs [] = {
    newRectangle(3, 4),
    newEllipse(2, 5),
    newRectangle(4, 3)
  };

  /* Jedes Objekt mit seiner Ausgabefunktion ausgeben. */
  int i, n = sizeof(fs) / sizeof(fs[0]);
  for (i = 0; i < n; i++) {
    Figure f = fs[i];
    f->print(f);
  }

  return 0;
}
```

Funktionszeigertypen mit beliebigen Parametern

- Wenn die Parameterliste einer Funktion leer ist, kann die Funktion prinzipiell mit beliebigen Parametern aufgerufen werden. (Um wirklich eine parameterlose Funktion zu erhalten, muss die Parameterliste `void` sein.)

- Ebenso gilt: Wenn die Parameterliste eines Funktionszeigertyps leer ist, kann an eine Variable oder einen Parameter dieses Typs eine Funktion mit beliebigen Parametern zugewiesen werden, sofern sie den richtigen Resultattyp besitzt.

2.2.7 Überschreiben und dynamisches Binden von Funktionen

Problem

- Obwohl die Funktionen `withdraw` und `transfer` auch für `LimitedAccount`-Objekte aufrufbar sein sollen – damit ein `LimitedAccount`-Objekt überall verwendet werden kann, wo ein `Account`-Objekt erwartet wird –, soll ihr Verhalten für `LimitedAccount`-Objekte anders sein als für `Account`-Objekte.

- Allgemein gesprochen, soll das Verhalten bestimmter Funktionen davon abhängen, für welche Art von Objekten sie aufgerufen werden.

Lösung

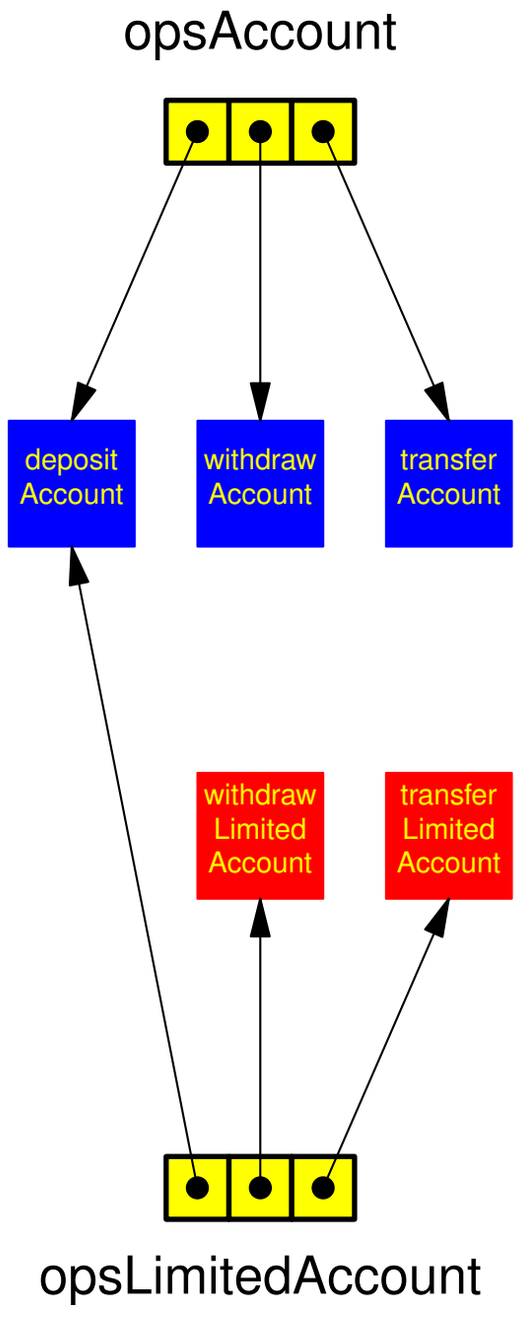
- Es gibt ggf. unterschiedliche Implementierungen der Funktionen `deposit`, `withdraw` und `transfer` für `Account`- und `LimitedAccount`-Objekte:

```
/* Betrag amount auf gewöhnliches Konto this einzahlen. */
void depositAccount (struct Account* this, long amount) {
  this->balance += amount;
}

/* Betrag amount von gewöhnlichem Konto this abheben. */
void withdrawAccount (struct Account* this, long amount) {
  this->balance -= amount;
}

/* Betrag amount von gewöhnlichem Konto this */
/* auf Konto that überweisen. */
void transferAccount (struct Account* this, long amount,
                                            struct Account* that) {
  withdraw(this, amount);
  deposit(that, amount);
}
```

```
/* Überprüfen, ob Betrag amount von limitiertem Konto this */
/* abgezogen werden kann, ohne die Kreditlinie zu überschreiten. */
int check (struct LimitedAccount* this, long amount) {
  if (this->super.balance - amount >= -this->limit) return 1;
  printf("Unzulässige Kontoüberziehung!\n");
  return 0;
}

/* Betrag amount von limitiertem Konto this abheben, falls mgl. */
void withdrawLimitedAccount (struct LimitedAccount* this,
                                                  long amount) {
  if (check(this, amount)) {
    withdrawAccount((struct Account*)this, amount);
  }
}
/* Betrag amount von limitiertem Konto this */
/* auf Konto that überweisen, falls möglich. */
void transferLimitedAccount (struct LimitedAccount* this,
                                long amount, struct Account* that) {
  if (check(this, amount)) {
    transferAccount((struct Account*)this, amount, that);
  }
}
```

- Die für `Account`- bzw. `LimitedAccount`-Objekte passenden Implementierungen werden jeweils in einem *Funktionszeigersatz* zusammengefasst:

```
/* Funktionszeigersatz für Konten. */
struct AccountOps {
  void (*deposit) ();   /* Funktion für Einzahlung. */
  void (*withdraw) ();  /* Funktion für Abhebung. */
  void (*transfer) ();  /* Funktion für Überweisung. */
};

/* Funktionszeigersatz für gewöhnliche Konten. */
struct AccountOps opsAccount = {
  depositAccount,
  withdrawAccount,
  transferAccount
};

/* Funktionszeigersatz für limitierte Konten. */
struct AccountOps opsLimitedAccount = {
  depositAccount,
  withdrawLimitedAccount,
  transferLimitedAccount
};
```

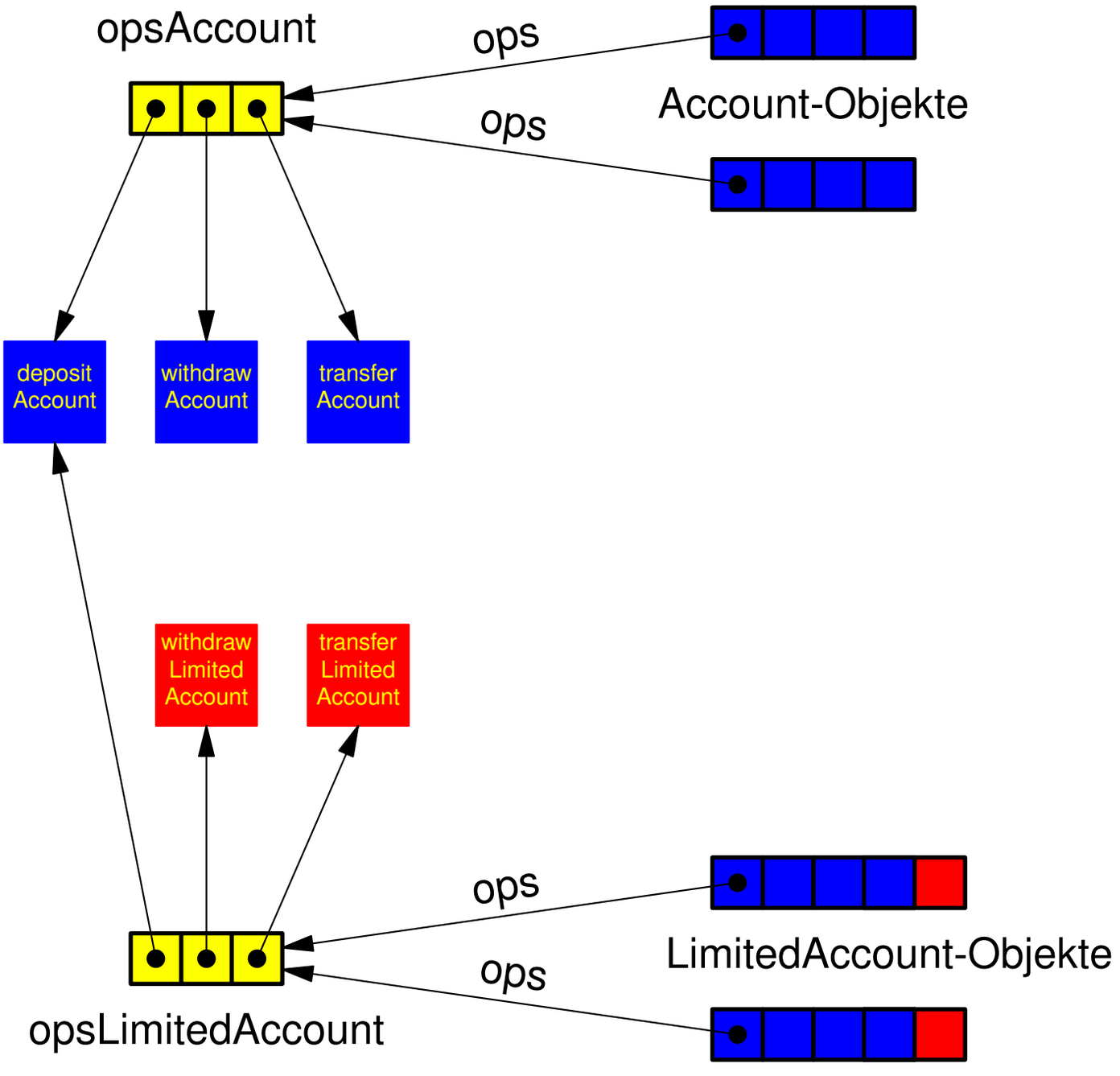

opsAccount
deposit Account
withdraw Account
transfer Account
withdraw Limited Account
transfer Limited Account
opsLimitedAccount

- `Account`-Objekte – und damit auch `LimitedAccount`-Objekte – besitzen als erste Komponente einen Zeiger auf einen Funktionszeigersatz:

```
/* Konto. */
struct Account {
  struct AccountOps* ops; /* Zeiger auf Funktionszeigersatz. */
  long number;            /* Kontonummer. */
  String holder;          /* Kontoinhaber. */
  long balance;           /* Kontostand in Cent. */
};

/* Limitiertes Konto. */
struct LimitedAccount {
  struct Account super;   /* Account-Daten (inkl. ops-Zeiger). */
  long limit;             /* Kreditlinie in Cent. */
};
```

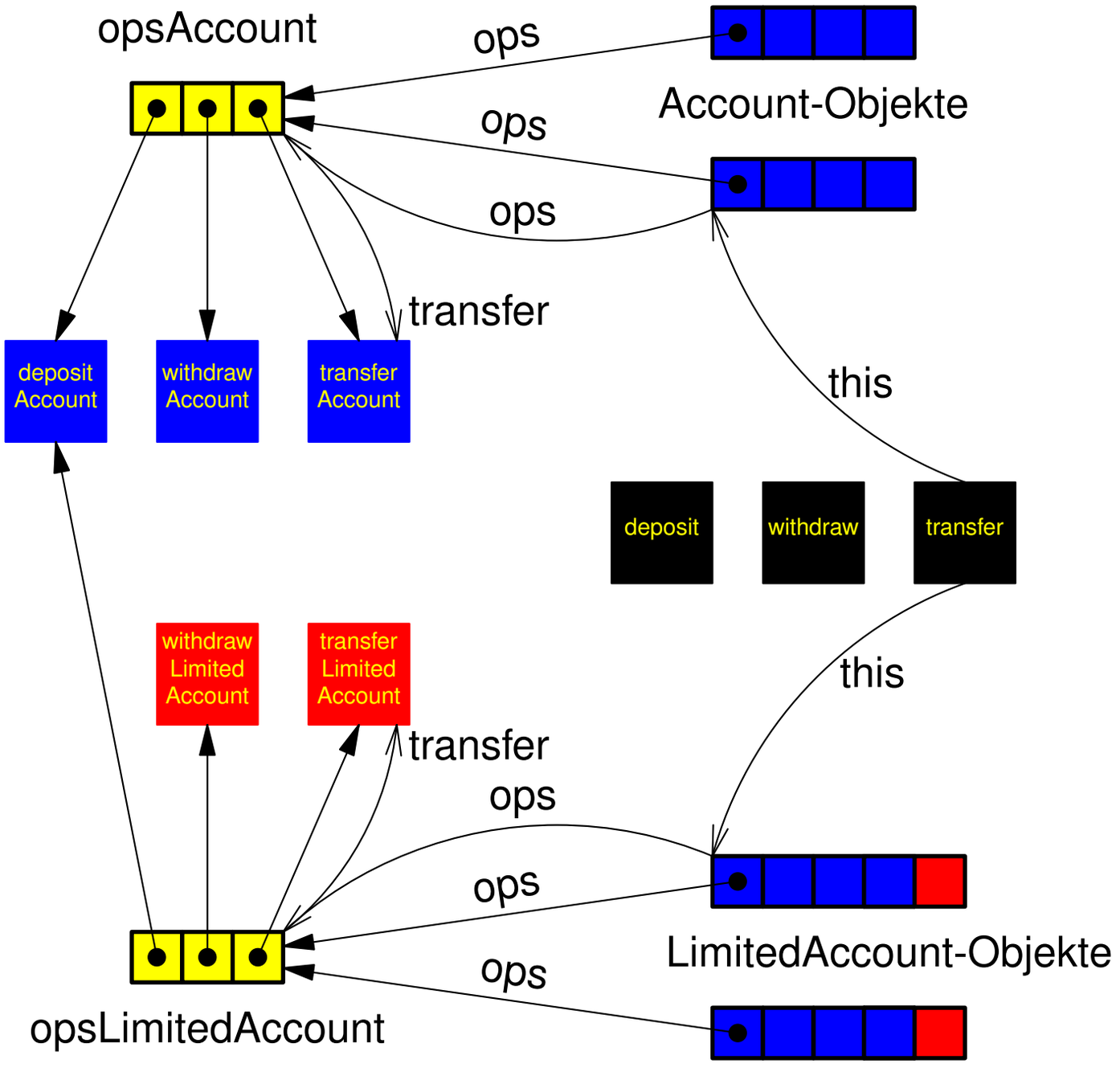
- Die „Konstruktoren“ `newAccount` und `newLimitedAccount` initialisieren den Zeiger auf den Funktionszeigersatz mit der passenden Adresse:

```
/* Konto mit Inhaber h, eindeutiger Nummer */
/* und Anfangsbetrag 0 erzeugen. */
Account newAccount (String h) {
  struct Account* this = newObject(sizeof(struct Account));
  this->ops = &opsAccount;
  initAccount(this, h);
  return this;
}

/* Limitiertes Konto mit Inhaber h, Kreditlinie l, */
/* eindeutiger Nummer und Anfangsbetrag 0 erzeugen. */
LimitedAccount newLimitedAccount (String h, long l) {
  struct LimitedAccount* this =
                        newObject(sizeof(struct LimitedAccount));
  this->super.ops = &opsLimitedAccount;
  initLimitedAccount(this, h, l);
  return this;
}
```


opsAccount
ops
ops
Account-Objekte
deposit Account
withdraw Account
transfer Account
withdraw Limited Account
transfer Limited Account
ops
ops
LimitedAccount-Objekte
opsLimitedAccount

- Die „generischen“ Funktionen `deposit`, `withdraw` und `transfer` rufen über den Funktionszeigersatz ihres Zielobjekts `this` automatisch die jeweils passende Implementierung auf:

```
/* Betrag amount auf irgendein Konto this einzahlen. */
void deposit (struct Account* this, long amount) {
  this->ops->deposit(this, amount);
}

/* Betrag amount von irgendeinem Konto this abheben. */
void withdraw (struct Account* this, long amount) {
  this->ops->withdraw(this, amount);
}

/* Betrag amount von irgendeinem Konto this */
/* auf Konto that überweisen. */
void transfer (struct Account* this, long amount, Account that) {
  this->ops->transfer(this, amount, that);
}
```


opsAccount
ops
Account-Objekte
ops
ops
transfer
deposit Account
withdraw Account
transfer Account
this
deposit
withdraw
transfer
withdraw Limited Account
transfer Limited Account
this
transfer
ops
ops
ops
LimitedAccount-Objekte
opsLimitedAccount

2.2.8 Anwendungsbeispiel

```
int main () {
  /* Ein limitiertes und ein normales Konto erzeugen. */
  LimitedAccount a = newLimitedAccount("Max Mustermann", 500);
  Account b = newAccount("Erika Mustermann");

  /* 1000 Cent auf Konto a einzahlen, */
  /* dann 300 Cent auf Konto b überweisen. */
  /* (a kann wie ein normales Konto verwendet werden.) */
  deposit(a, 1000);
  transfer(a, 300, b);

  /* Von jedem Konto 2000 Cent abheben. */
  /* Bei a würde das die Kreditlinie überschreiten. */
  withdraw(a, 2000);
  withdraw(b, 2000);
```

```
  /* Kontodaten ausgeben. */
  printf("Konto a: %ld, %s, %ld\n",
                 number(a), holder(a), balance(a));
  printf("Konto b: %ld, %s, %ld\n",
                 number(b), holder(b), balance(b));

  return 0;
}
```

2.3 Rückblick

- Objektorientierte Programmierung beinhaltet u. a.:
  - Datenkapselung/Geheimnisprinzip
  - Vererbung/Wiederverwendung von Daten und Operationen
  - Überschreiben und dynamisches Binden von Operationen
- Objektorientierte Programmierung mit C ist prinzipiell möglich, aber mühsam:
  - Explizite Verwaltung von Funktionszeigersätzen
  - Umständlicher Zugriff auf „geerbte“ Datenfelder
  - Mechanisch zu generierender Code
- Außerdem sind ein paar Tricks erforderlich, um zu strenge Typprüfungen des Übersetzers zu umgehen:
  - `void*` als Zeiger auf `Account`- und `LimitedAccount`-Objekte, damit `LimitedAccount`-Objekte als `Account`-Objekte verwendbar sind. Kehrseite: `Account`-Objekte können auch als `LimitedAccount`-Objekte verwendet werden, was jedoch zu undefiniertem Verhalten führt.
  - Bewusst ungenaue Typen in Funktionszeigersätzen, z. B. `void (*withdraw) ()` mit beliebiger Parameterliste statt `void (*withdraw) (struct Account*, long)`.

2.4 Implementierung in Java

Allgemeine Konten

```
// Klasse: Konto.
class Account {
  // Objektvariablen:
  private int number;           // Kontonummer.
  private String holder;        // Kontoinhaber.
  private int balance;          // Kontostand in Cent.

  // Klassenvariable: Nächste zu vergebende Kontonummer.
  private static int nextNumber = 1;

  // Konstruktor:
  // Konto mit Inhaber h, eindeutiger Nummer
  // und Anfangsbetrag 0 initialisieren.
  public Account (String h) {
    this.number = nextNumber++;
    this.holder = h;
    this.balance = 0;
  }
```

```
  // Objektmethoden: Kontonummer/-inhaber/-stand abfragen.
  public int number () {
    return this.number;
  }
  public String holder () {
    return this.holder;
  }
  public int balance () {
    return this.balance;
  }

  // Objektmethoden: Betrag amount einzahlen/abheben/überweisen.
  public void deposit (int amount) {
    this.balance += amount;
  }
  public void withdraw (int amount) {
    this.balance -= amount;
  }
  public void transfer (int amount, Account that) {
    this.withdraw(amount);
    that.deposit(amount);
  }
}
```

Limitierte Konten

```
// Unterklasse von Account: Limitiertes Konto.
class LimitedAccount extends Account {
  // Zusätzliche Objektvariable:
  private int limit;              // Kreditlinie in Cent.

  // Konstruktor:
  // Limitiertes Konto mit Inhaber h, Kreditlinie l,
  // eindeutiger Nummer und Anfangsbetrag 0 initialisieren.
  public LimitedAccount (String h, int l) {
    // Konstruktor der Oberklasse Account aufrufen,
    // um deren Objektvariablen zu initialisieren.
    super(h);

    limit = l;
  }

  // Zusätzliche Objektmethode: Kreditlinie abfragen.
  public int limit () { return limit; }
```

```
  // Hilfsmethode: Kann Betrag amount abgezogen werden,
  // ohne die Kreditlinie zu überschreiten?
  private boolean check (int amount) {
    if (balance() - amount >= -limit) return true;
    System.out.println("Unzulässige Kontoüberziehung!");
    return false;
  }

  // Überschreiben geerbter Objektmethoden:
  // Betrag amount abheben/überweisen.
  public void withdraw (int amount) {
    if (check(amount)) {
      // Überschriebene Methode aufrufen.
      super.withdraw(amount);
    }
  }
  public void transfer (int amount, Account that) {
    if (check(amount)) {
      // Überschriebene Methode aufrufen.
      super.transfer(amount, that);
    }
  }
}
```

Anwendungsbeispiel

```
class Test {
  // Hauptprogramm.
  public static void main (String [] args) {
    // Objekte erzeugen und durch Konstruktoraufrufe initialisieren.
    LimitedAccount a = new LimitedAccount("Max Mustermann", 500);
    Account b = new Account("Erika Mustermann");

    // Methoden auf Objekten aufrufen.
    a.deposit(1000);
    a.transfer(300, b);

    a.withdraw(2000);
    b.withdraw(2000);

    // Ausgabe.
    System.out.println("Konto a: " +
            a.number() + " " + a.holder() + " " + a.balance());
    System.out.println("Konto b: " +
            b.number() + " " + b.holder() + " " + b.balance());
  }
}
```

2.5 Darstellung als UML-Klassendiagramm