Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. K. D. Müller-Glaser
Prof. Dr.-Ing. J. Becker
Prof. Dr. rer. nat. W. Stork

Übung04: Informationstechnik (IT)

Tobias Schwalb & Michael Tansella

Institut für Technik der Informationsverarbeitung (ITIV)

Teil2: Dynamische Speicherverwaltung und Objektorientierung

Inhalt: Übung04

Karlsruher Institut für Technologie

1. Besprechung Übungsaufgaben 3.01-3.06
2. Dynamische Speicherverwaltung
3. Objektorientierung 1
4. Objektorientierung 2

Speicherverwaltung für Variablen

- Speicherbereich für statische Variablen ist der Stack (= Stapel)
  - Neue Variablen werden auf den Stack gelegt, überdecken evtl. unter ihnen liegende Variablen und wieder von Stack genommen
- Größe des Stacks konstant ← wird vom Compiler errechnet
- Vorteile:
  - Variablen werden komplett automatisch verwaltet
- Nachteile:
  - Gültigkeitsbereich kann nur eingeschränkt selbst bestimmt werden
  - Große Datenmengen die nur punktuell benötigt werden sorgen für einen überdimensionierten Stack
  - Platz auf dem Stack für große Datenmengen nicht ausreichend

Dynamische Speicherverwaltung

- Problemstellung
  - Wie kann ich die Nachteile von statischen Variablen umgehen?
  - Wie groß soll ich mein Array machen? Ich weiß ja nicht wie viele Daten der Benutzer eingibt.
- Lösung: Dynamische Speicherverwaltung
  - Speicher für Variablen kann dynamisch auf dem Heap reserviert und wieder freigegeben werden
- Vorteile:
  - Keine Verschwendung von Speicherplatz
  - Heap bietet mehr Speicherplatz als der Stack
- Nachteile:
  - Entwickler muss sich um Anlegen / Löschen des Speicherplatzes kümmern

Speicher auf dem Heap reservieren

- Neues Objekt anlegen mit den Befehl **new**
  - Benötigt den **Typ** des neuen Objekts und gibt einen **Zeiger** zurück
  - Zeiger passender Adressvariablen zuweisen

- Syntax:
```
Typ* ZeigerAufTyp = new Typ;
```

- Beispiel:
```
double* pld = new double;
double* psd = new double( 100.99 );
```

- Zugriff auf die neu angelegte Variable über Dereferenzierung
- Beispiel:
```
*pld = 100.0;
*psd = *psd * 100.0;
```


Speicher auf dem Heap freigeben

- Speicherplatz freizugeben mit dem Befehl **delete**
  - Nur mit **new** reservierter Speicher kann freigegeben werden
  - Speicher wird am Programmende automatisch freigegeben
  - Sollte immer explizit durch **delete** freigegeben werden

- Syntax:
```
delete ZeigerAufTyp;
```

- Beispiel:
```
delete pld;
```

- Die Adressvariable **pld** existiert weiterhin nur der reservierte Speicher, worauf der Zeiger zeigte, ist wieder freigegeben
  - Daraus folgt: Zeiger zeigt nun auf undefinierten Speicher
  - Tipp: Zeiger NULL setzen → `pld = NULL;`

Arrays auf dem Heap

- Arrays können dynamisch reserviert und freigegeben werden
- Syntax: `Typ* ZeigerAufTyp = new Typ[anzahl];`
  - Man erhält einen Zeiger auf das erste Element des Arrays
- Beispiel:
```
short elemente = 100;
int* intvekptr = new int[elemente];
```
- Speicherplatz von Arrays freigeben mit `delete[]` (ohne Anzahl)
- Syntax: `delete[] ZeigerAufTyp;`
- Beispiel: `delete[] intvekptr;`
  - Adressvariable `intvekptr` existiert weiterhin

Gültigkeitsbereich dyn. Variablen

- Eine dynamisch erzeugte Variable lebt von ihrer Erzeugung mit `new`, bis diese mit `delete` gelöscht wird

```
int* gibPointerAufSumme( int zahl1, int zahl2 ) {
  int* summe = new int;
  *summe = zahl1 + zahl2;
  return summe;
}

int main() {
  int* meinPointer = gibPointerAufSumme( 10, 4 );
  int meineSumme = *meinPointer;
  cout << meineSumme;
  delete meinPointer;
  return 0;
}
```

- Integer-Variable wird dynamisch erzeugt
- Addieren und zuweisen
- `summe` (Adresse) zurückgeben
- Hier wird die Adressvariable `summe` zerstört, nicht jedoch der Wert auf den `summe` zeigt
- `meinPointer` zeigt jetzt auf ein existierendes Objekt – wurde in der Funktion angelegt
- `meinPointer` wird dereferenziert
- Speicher freigeben

Zwischenübung12: Speicherreservierung

Finden Sie den Fehler in den folgenden Programmabschnitten

a)
```
long* p = new long;
long* q = new long( 1000L );
p = q;
```

b)
```
double* p;
double* q = new double( 9.5 );
*p = *q;
```

c)
```
float* p = new float( 10.0 );
cout << *p << endl;
delete *p;
```

d)
```
double* p1;
double* p2 = new double( 32.1 );
p1 = p2;
delete p1;
delete p2;
```


Objekt orientierte Programmierung: OOP

- Problemstellung:
  - Wie kann ich reelle Objekte oder dem Menschen naheliegende Gedankenkonstrukte in Programmstrukturen möglichst sinngemäß abbilden?
- Lösung: Programmierparadigma **Objektorientierung**
  - Reelle Objekte werden in Software nachgebildet: Klassen
  - Klasse hat Attribute (Eigenschaften) und Methoden (Fähigkeiten)
  - Klasse kann Beziehungen zu anderen Klassen haben: Assoziation, Aggregation, Komposition, Vererbung, Polymorphie, …
- Klassen sind Baupläne für Objekte
  - Auch mehrere Objekte vom selben Bauplan möglich
- Kapselung von Daten und Methoden
  - Private und öffentliche Daten / Methoden bzw. Schnittstellen

Definition von Klassen

Definition von Klassen normalerweise in der Header-Datei: *Klassenname.h*

```
class Konto {
  private:
    string inhaber;
    unsigned long nr;
    double stand;
    void inhaberAendern( const string& );

  public:
    bool init( const string&, unsigned long, double);
    void display();
};
```

Klassenname (Beginn mit Großbuchstaben)

Abschnitt der privaten Attribute und Methoden
- Nur innerhalb der Klasse zugreifbar

z.B. Anlegen von Variablen

z.B. Deklaration von Methoden

Abschnitt der öffentlichen Attribute und Methoden
- Von innerhalb und außerhalb der Klasse zugreifbar

Semikolon am Ende der Definition der Klasse!

Definition von Methoden

- Definition von Methoden, wie bei Funktionen plus Angabe des Klassennamens und Verwendung des Bereichsoperators `::`
- Alle Elemente (Variablen / Methoden) einer Klasse können in allen Methoden der Klasse direkt mit Ihrem Namen verwendet werden
- Syntax:

```
typ klassenname::methodenname( parameterliste ) {
  //Anweisungen
}
```

Definition von Methoden normalerweise in der CPP-Datei: *Klassenname.cpp*

- Beispiel:

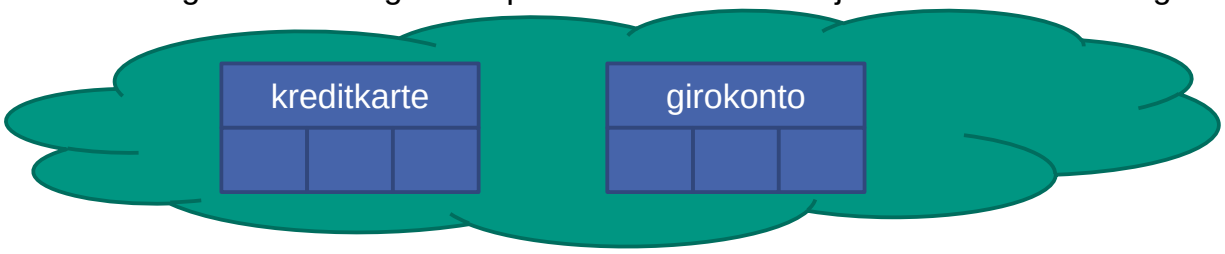
```
void Konto::display() {
  cout << fixed << setprecision(2)
       << "-------------------------------------\n"
       << "Kontoinhaber:   " << name  << '\n'
       << "Kontonummer:    " << nr    << '\n'
       << "Kontostand:     " << stand << '\n';
}
```

Definition von Objekten

- Klassen bilden den Bauplan für Objekte – wie bei Variablentypen
  - Objekte sind *Instanzen* von Klassen

- Syntax: `klassenname objektname1 [, objektname2, ...];`
- Beispiel: `Konto kreditkarte, girokonto;`

- Für jedes Objekt wird dabei Speicherplatz reserviert – nur Attribute
- Methoden sind nur einmal vorhanden, greifen allerdings jeweils auf die entsprechenden Attribute des Objektes zu
  - Belegen auch Programmspeicher wenn kein Objekt der Klasse erzeugt wird



Verwendung von Objekten

- Im Anwendungsprogramm Zugriff auf Attribute & Methoden eines Objektes über den Punktoperator `.`
  - NUR auf Public-Elemente (lesend und schreibend)
  - Wird von Compiler überprüft → Kompilierungsfehler

- Syntax: `objekt.element; objekt.methode();`
- Beispiel:

```
kreditkarte.init( "Paul", 5789, 111.12 );
kreditkarte.nr = 5789;     //privates Element!
kreditkarte.display();     //öffentliche Methode
```

- Direktes Kopieren von Objekten (nur selbe Klasse) möglich
  - Inhalt aller Attribute wird kopiert

- Beispiel: `girokonto = kreditkarte;`

Zeiger auf Objekte

- Objekt (Instanzen von Klassen) haben auch eine Adresse im Speicher
  - Kann einem entsprechendem Zeiger zugewiesen werden
- Beispiel: `Konto* ptrKonto = &kreditkarte;`
- Zugriff auf Methoden und Attribute auch über Zeiger möglich
  - Verwendung des Pfeiloperators `->`
- Beispiel:
```
(*ptrKonto).display(); //Klammern beachten
ptrKonto->display(); //einfacher
```
- Merke: Der linke Operand des Punktoperators ist ein Objekt
  Der linke Operand des Pfeiloperators ist ein Zeiger

Zwischenübung13: Definition von Methoden

Wie kann man die Methoden getPsZahl() & setMarke()
der Klasse CAuto in CAuto.h definieren?

```cpp
// CAuto.cpp
// Enthält die Definition der Methoden
// der Klasse CAuto.
//--------------------------------------

// Definition der Klasse
#include "CAuto.h"

int CAuto::getPsZahl() {
  return psZahl;
}

void CAuto::setMarke( Marke m ) {
  marke = m;
}
//...
```

```cpp
// CAuto.h
// Enthält die Definition der Klasse CAuto
//--------------------------------------

#include "myTypes.h"

class CAuto {
private:
  int psZahl;
  short baujahr;

protected:
  Marke marke;

public:
  int geschwindigkeit;
  CAuto();
  ~CAuto();
  void init( int, Marke, short, int );
  int getPsZahl();
  short getBaujahr();
  void setMarke( Marke m );
  void setPsZahl( int ps );
  void setBaujahr( short jahr );
  void beschleunigen( int betrag );
  int bremsen();
};
```

Konstruktoren

- Methode, welche beim Erstellen eines Objekts aufgerufen wird
  - Nützlich zur Initialisierung von Attributen
  - Name entspricht dem Klassennamen, hat keinen Rückgabewert
  - Konstruktoren können nicht für existierende Objekte aufgerufen werden

- Deklaration Syntax: `Klassenname( parameterliste );`
- Definition Syntax: `Klassenname::Klassenname ( parameterliste )`
- Aufruf Syntax: `Klassenname Objekt ( Übergabeparameter );`

- Bsp:
```
Konto::Konto( int ktrNr; double stand ) { … }
Konto::Konto() { } //Methoden überladen
Konto sparbuch( 12345, 112.23 );
```
Parameteranzahl und jeweiliger Typ muss zu einem Konstruktor passen

- Jede Klasse besitzt einen leeren Default- Konstruktor, wenn kein anderer Konstruktor definiert wurde

Destruktoren

- Werden bei der Zerstörung eines Objektes aufgerufen
  - Nützlich zum Freigeben von Speicherplatz, Schließen von Dateien, …
  - Name entspricht dem Klassennamen, mit vorangestellter Tilde ~
  - Hat keinen Rückgabewert und keine Parameterliste

- Deklaration Syntax: `~Klassenname();`
- Definition Syntax: `Klassenname::~Klassenname () { … }`

- Aufruf bei Programmende oder bei dynamischer Speicherallokation bei Zerstörung des Objektes durch den Befehl **`delete`**

- Jede Klasse besitzt einen leeren Default-Destruktor, wenn kein anderer Destruktor definiert wurde
  - Eine Klasse kann nur einen Destruktor haben

Inline-Methoden

- Häufiger Aufruf von kurzen Methoden verlängert die Programmlaufzeit
  - Sicherung der Rücksprungadresse + Hin- und Rücksprung
- Vermeidung des Overhead durch Deklaration der Methode als inline
  - Methode wird jeweils an die entsprechende Stelle im Programm kopiert
  - Programmspeicher vs. Rechenzeit
- Explizite inline Definition durch das Schlüsselwort: **`inline`**
  - Beispiel: `inline void Konto::display() { … }`
- Implizite inline Definition durch Deklaration innerhalb der Klasse
  - Beispiel:

```
class Konto {
    //...
    bool isNull() { return (stand == 0); }
};
```


Zugriffmethoden & Standardmethoden

- Zugriff auf private Attribute über **`get`**- und **`set`**-Methoden
  - Datenkapselung, Zugriff auf Daten über feste Schnittstellen
  - Einfache Verwendung, Strukturen innerhalb der Klasse unwichtig
  - Überprüfung auf gültige Werte (z.B. nur positive Zahlen erlaubt)
- Beispiel: `getName(); getNr(); setName("Paul");`
- Jede Klasse besitzt 4 Standardmethoden
  - Default-Konstruktor (wenn kein anderer Konstruktor definiert)
  - Destruktor (wenn kein anderer Destruktor definiert)
  - Kopier-Konstruktor (Initialisierung eines Objekts mit einem anderen)
    - Beispiel: `Konto sparcardNeu(sparcard);`
  - Zuweisung
    - Beispiel: `sparcardNeu = sparcard;`

Der Inhalt aller Attribute wird kopiert

Beide Objekt müssen existieren

Konstante Objekte und Methoden

- Objekt können – wie Variablen – als **`const`** deklariert werden
  - Objekt muss initialisiert werden, Programm kann nur lesend zugreifen
  - Es können nur **`const`**-Methoden des Objekts aufgerufen werden
- Beispiel: `const Konto sparbrief( "WTS_3J", 5879, 5000.00 );`

- Auch Methoden können als nur lesend mit **`const`** deklariert werden
  - Methode nur lesend auf Attribute zugreifen und **`const`**-Methoden aufrufen
  - Methoden können auch von nicht konstanten Objekten verwendet werden
- Beispiel Deklaration: `double getStand() const;`

  Aufruf: `sparbrief.getStand(); sparbuch.getStand();`

- Compiler überprüft Einhaltung beim Kompilieren → Fehlermeldung
- Eine Methode gilt nicht automatisch als **`const`**, wenn diese nicht schreibend auf Daten zugreift – die Methode muss explizit als **`const`** deklariert werden

this-Zeiger

- Eine Methode kann auf jedes Element eines Objektes zugreifen
  - Objekt wird allerdings innerhalb der Methode nicht angegeben
  - Arbeitet immer mit dem Objekt auf welchem Sie aufgerufen wurde
- Beispiel: `girokonto.display();` `kreditkarte.display();`

- Beim Aufruf der Methode wird ihr die Adresse des Objekts als versteckter konstanter Zeiger übergeben

- Zugriff auf den Zeiger mit dem Schlüsselwort: **`this`**
  - this-Zeiger ist ein Zeiger auf das aktuelle Objekt

- Nützlich um zum Beispiel lokale Variablen einer Methode von Klassenelementen zu unterscheiden

Übergabe von Objekten

- Objekte können - wie Variablen - auch an Methoden übergeben werden
- Beispiel: `bool Konto::vergleich( Konto vergleichsKonto );`

- Call by Value
  - Es wird eine Kopie erzeugt & zerstört (Kopier-Konstruktor & Destruktor)
  - Bei großen Objekten kostet dies viel Speicherplatz und Rechenzeit

- Call by Reference
  - Es wird eine Referenz an die Methode übergeben – ACHTUNG: Methode kann verändernd auf das Original-Objekt zugreifen
  - Auch Übergabe einer konstanten Referenz möglich – Methode kann das Objekt nicht verändern
  - Vermeidung des Overhead durch anlegen und zerstören von Objekten
  - Beispiel: `bool Konto::vergleich( const Konto& vergleichsKonto );`

Objekte als Rückgabewert

- Eine Methode kann auch ein Objekt als Return-Wert liefern
  - Rückgabe als Kopie, Referenz oder als Zeiger
- Beispiel: `Konto Konto::wandeln();`

- Rückgabe einer Kopie
  - Objekt wird auch kopiert (Speicherplatz und Rechenzeit)

- Rückgabe als Referenz oder Zeiger

  - Die Lebensdauer des Objektes, welches zurückgegeben wird, darf nicht lokal sein → keine Referenz, Zeiger auf ein (lokales) Objektes, welches am Ende der Methode zerstört wird
  - Rückgabe auf ein `static`-Objekt oder Rückgabe eines dynamisch erzeugten Objektes

Referenz & Ausblick

- Kompendium: Kapitel 7 - Dynamische Speicherverwaltung
  Kapitel 8 – Objektorientierung (ohne Vererbung & Polymorphie)
- Tutorium: Aufgabe 15 & 17
- Wie kann ich die Eigenschaften einer Klasse an eine andere Klasse weitergeben und diese zusätzlich erweitern?
- OOP: Vererbung und Polymorphie?
- Wie kann ich Zeichenketten in C++ anlegen und diese verwenden?
- …

Vielen Dank für Ihre Aufmerksamkeit

Tobias Schwalb & Michael Tansella

Karlsruher Institut für Technology (KIT) – ITIV
tobias.schwalb@kit.edu
michael.tansella@kit.edu