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

Übung05: Informationstechnik (IT)

Tobias Schwalb & Michael Tansella

Institut für Technik der Informationsverarbeitung (ITIV)

Teil2: Vererbung & Polymorphie, Strings und verkettete Listen

KIT – Universität des Landes Baden-Württemberg und
nationales Forschungszentrum in der Helmholtz-Gemeinschaft

www.kit.edu

Inhalt: Übung05

Karlsruher Institut für Technologie

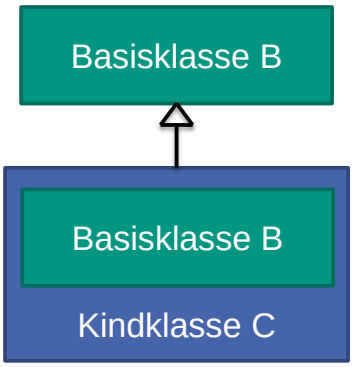
1. Besprechung Übungsaufgaben 4.01-4.04
2. Zulassungsaufgabe
3. Vererbung / Polymorphie
4. Strings (kurze Einführung)
5. Verkettete Listen

Vererbung

- Vererbung ermöglicht eine Reduktion des Quellcodes und die Erstellung einer logischen Hierarchie im Code
- Eine Basisklasse vererbt der abgeleiteten Klasse (Kindklasse) alle ihre **`public`** und **`protected`** Methoden und Attribute
  - Kindklasse hat die ganze Funktionalität der Basisklasse
  - Kann zusätzlich um weitere Attribute und Methoden ergänzt werden

- Beispiel:

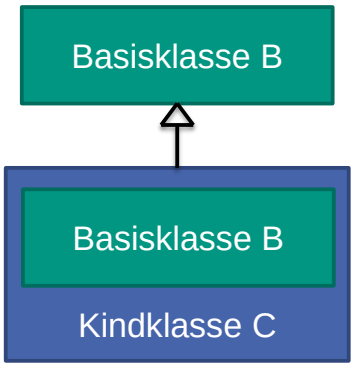
```
class C : public B {
  private:
    /* Deklaration der zusätzlichen privaten
       Datenelemente und Elementfunktionen */
  public:
    /* Deklaration der zusätzlichen öffentlichen
       Datenelemente und Elementfunktionen */
};
```



Vererbung - Zugriffsrechte

- Alle **`public`** Elemente der Basisklasse B sind auch in der abgeleiteten Kindklasse C öffentlich

- Methoden Kindklasse C können allerdings nicht auf die **`private`** Elemente der Basisklasse B zugreifen (nur über dessen öffentliche Methoden)



- Beispiel:

```
class Kfz {
  private:
    long nr;
    string hersteller;

  public:
    long getNr();
    //...
};
```

```
class Pkw : public Kfz {
  private:
    string pkwTyp;
    bool schiebe;

  public:
    void display( void );
    //...
};
```

```
void Pkw::display( void ) {
  cout << "Hersteller: "
       << hersteller << endl;
```
Privates Attribut der Basisklasse
```
  cout << "Kfz-Nummer: "
       << getNr() << endl;
```
Öffentliche Methode der Basisklasse
```
  cout << "Typ: " << pkwTyp;
}
```
Privates Attribut der Kindklasse

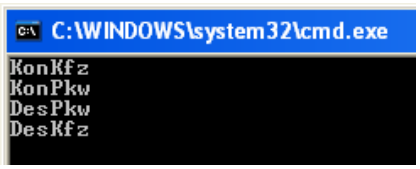
Vererbung - Redefinition

- In der Kindklasse können Methoden und Attribute redefiniert werden
  - Gleicher Namen, wie Methoden bzw. Attribute der Basisklasse
  - Elemente der Basisklasse werden verdeckt, sind aber verfügbar
- Der Zugriff auf Basisklassen-Methoden erfolgt über den Bereichsoperator `::`
- Nützliches Werkzeug in Bezug auf Polymorphie (ab Folie 9)

```
class Kfz {
  public:
    void display( void );
    //...
};

class Pkw : public Kfz {
  public:
    void display( void );
    //...
};

int main() {
  Pkw fahrzeug;

  fahrzeug.display();

  fahrzeug.Kfz::display();
  return 0;
}
```


Vererbung - Konstruktoren & Destruktoren

- Beim Anlegen bzw. Zerstören einer abgeleiteten Klasse wird auch der Konstruktor bzw. Destruktor der Basisklasse aufgerufen
- Dabei gilt die Reihenfolge:
  1. Konstruktor der Basisklasse
  2. Konstruktor der Kindklasse
  3. Verwendung der Klasse
  4. Destruktor der Kindklasse
  5. Destruktor der Basisklasse



- Beispiel:

```
class Kfz {
  public:
    Kfz() { cout << "KonKfz" << endl; }
    ~Kfz() { cout << "DesKfz" << endl; }
};

class Pkw : public Kfz {
  public:
    Pkw() { cout << "KonPkw" << endl; }
    ~Pkw() { cout << "DesPkw" << endl; }
};

int main() {
  Pkw fahrzeug;
  return 0;
}
```

Protected - Bezeichner

- Attribute und Methoden einer Basisklasse sollen einerseits vor dem Zugriff von außen geschützt sein – aber von der Kindklasse aus zugreifbar
- Verwendung der **`protected`**-Deklaration
  - Methoden von abgeleiteten Klassen können auf **`protected`** Elemente der Basisklasse zugreifen
  - **`protected`** Elemente sind allerdings von außerhalb der Klasse nicht zugreifbar

- Beispiel:

```
class Kfz {
  protected:
    bool tunen;
    //...
};

class Pkw : public Kfz {
  public:
    void display( void );
    //...
};

void Pkw::display( void ) {
  cout << tunen << endl;
}

int main() {
  Pkw fahrzeug;
  fahrzeug.display();
  //fahrzeug.tunen = true;
  return 0;
}
```

OK - Protected Attribut aus Methode der Kindklasse zugreifbar

Fehler - Protected Attribut von außen nicht zugreifbar

Typumwandlung / Konvertierung

- Jedem Objekt einer Basisklasse kann ein Objekt einer Kindklasse zugewiesen werden → implizite Typumwandlung
  - Inhalt der entsprechenden Attribute wird kopiert
  - Bsp: **`Pkw fahrzeug;  Kfz auto;  auto = fahrzeug;`**
- Zeiger auf die Basisklasse kann auch eine Kindklasse referenzieren
  - Zeiger kann nur auf öffentliche Elemente der Basisklasse zugreifen
  - Bsp: **`Kfz* kfzPtr = &fahrzeug;`**
- Umkehrung nur durch explizite Typumwandlung möglich
  - Programmierer muss auf den korrekten Objekttyp achten
  - Bsp: **`( (Pkw*) kfzPtr )->display();`**

```
Kfz* zweiterKfzPtr = &auto;
( (Pkw*) zweiterKfzPtr )->display();
```

OK – Nur Klammern beachten

Laufzeitfehler → Programmabsturz

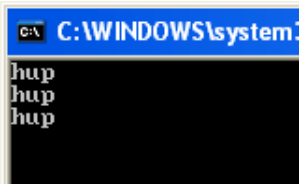
Polymorphie - Problemstellung

- Problemstellung Beispiel:

```
class Kfz {
  public:
    void hupen( void );
    //...
};

class Pkw : public Kfz {
  public:
    void hupen( void );
    //...
};

class Lkw : public Kfz {
  public:
    void hupen( void );
    //...
};

void Kfz::hupen( void ) {
  cout << "hup" << endl;
}
```

```
void Pkw::hupen( void ) {
  cout << "Tuuto" << endl;
}

void Lkw::hupen( void ) {
  cout << "Troeoet" << endl;
}

int main() {
  Kfz* autoPark[3];
  autoPark[0] = new Pkw();
  autoPark[1] = new Lkw();
  autoPark[2] = new Lkw();

  for( int i = 0; i < 3; i++ ) {
    autoPark[i]->hupen();
  }

  return 0;
}
```

Wie können die Fahrzeuge richtig hupen ?



Polymorphie

- Normalerweise werden Elemente während dem Kompilieren entsprechend dem Typ des Elements / Zeigers ausgewählt
  - Im Beispiel war **`autoPark`** vom Typ **`Kfz*`**, daher wurde die Methode **`hupen()`** von der Klasse **`Kfz`** verwendet
  - Andere Methode durch Typcast möglich – ABER: Verschiedene Kindklassen im Array abgelegt → Wie richtige Typ/Methode auswählen?!
- Element anhand des Typen des Objekts zur Laufzeit automatisch auswählen mit dem Schlüsselwort **`virtual`**
  - Element muss in der Basisklasse als **`virtual`** deklariert werden
  - Element kann (muss aber nicht) in den Kindklassen redefiniert werden
  - Entscheidung geschieht zur Laufzeit (Late Binding)
  - Besonders wichtig auch bei Konstruktoren und Destrukturen
  - Virtuelle Methoden kosten etwas mehr Speicherplatz und sind etwas langsamer – Vorteile überwiegen allerdings

Polymorphie - virtual

- Problemstellung Beispiel:

```
class Kfz {
  public:
    virtual void hupen( void );
    //...
};

class Pkw : public Kfz {
  public:
    void hupen( void );
    //...
};

class Lkw : public Kfz {
  public:
    void hupen( void );
    //...
};

void Kfz::hupen( void ) {
  cout << "hup" << endl;
}
```

Einzige Änderung

```
void Pkw::hupen( void ) {
  cout << "Tuuto" << endl;
}

void Lkw::hupen( void ) {
  cout << "Troeoet" << endl;
}

int main() {
  Kfz* autoPark[3];
  autoPark[0] = new Pkw();
  autoPark[1] = new Lkw();
  autoPark[2] = new Lkw();

  for( int i = 0; i < 3; i++ ) {
    autoPark[i]->hupen();
  }

  return 0;
}
```

```
C:\WINDOWS\system32\cm
Tuuto
Troeoet
Troeoet
```


Zwischenübung15: Vererbung & Polymorphie

Bitte zuordnen?

```
//Anweisungsblock A:
irgendeinePflanze = meinBaumBernd;
irgendeinePflanze->vergammel();
```
1

```
//Anweisungsblock A:
irgendeinePflanze = narzisse;
((Blume*)irgendeinePflanze)->vergammel();
```
2

```
//Anweisungsblock A:
meinBaumBernd->bringeSamen();
```
3

```
//Anweisungsblock A:
einfachePflanze->bringeSamen();
```
4

```
C:\WINDOWS\system32\cmd.exe
Ich bin eine grundlegende Pflanze.
```

```
C:\WINDOWS\system32\cmd.exe
oh nein, meine Sch÷nheit ist dahin!
```

```
C:\WINDOWS\system32\cmd.exe
oh ich zerfalle zu Humus
```

```
C:\WINDOWS\system32\cmd.exe
ich habe Samen geworfen.
```

```
class Pflanze {
  public:
    Pflanze(){}
    virtual ~Pflanze(){}
    void vergammel();
    virtual int bringeSamen();
};
class Blume : public Pflanze {
  public:
    void vergammel();
    int bringeSamen();
};
class Baum : public Pflanze {
  public:
    int bringeSamen();
};

void Pflanze::vergammel() {
  cout << "oh ich zerfalle zu Humus" << endl;
}
int Pflanze::bringeSamen() {
  cout << "Ich bin eine grundlegende Pflanze." <<endl;
  return 1;
}
int Blume::bringeSamen() {
  cout << "Ich habe geblüht. Viele Samen." << endl;
  return 400;
}
void Blume::vergammel() {
  cout << "oh nein, meine Schönheit ist dahin!" << endl;
}
int Baum::bringeSamen() {
  cout << "ich habe Samen geworfen." << endl;
  return 5000;
}
```

```
int main() {
  Pflanze* einfachePflanze = new Pflanze();
  Baum* meinBaumBernd = new Baum();
  Blume* narzisse = new Blume();
  Pflanze* irgendeinePflanze;
  //Anweisungsblock A
  return 0;
}
```

Strings - Allgemein

- Allgemein
  - Einbinden der Bibliothek **<string>**
  - Zur Darstellung und Verarbeitung von Zeichenketten in C++
  - Speicherplatz wird automatisch reserviert und angepasst
- Initialisierung / Zuweisung
  - Ohne Initialisierung ist der String leer (kein zufälliger Inhalt)
  - String aus der Konsole einlesen mit **cin**
    - Operator **>>** immer nur ein Wort (ohne führende Leerzeichen)
    - **getline()** einlesen einer kompletten Zeile (mit Leerzeichen)

Beispiel:
```
string meldung = "Guten Morgen";
```

Zeichenkette: **(Länge = 12)**

Position: 0 11

Strings verketten und vergleichen

- Strings verketten mit Operator **+**
  - Mit String-Variablen, String-Konstanten, einzelnen Zeichen
  - Ein Operand muss vom Typ **string** sein
  - Anhängen mit dem Operator **+=**

Beispiel:
```
string erg, s1 = "Kaffee", s2 = "ta";
erg = s1 + s2 + "ss" + 'e';  //erg = "Kaffeetasse"
```

- Strings vergleichen mit Operatoren **< <= == != > >=**
  - Vergleich erfolgt zeichenweise anhand der ASCII-Tabelle
  - Liefert ein Ergebnis von Typ **bool** (**true** oder **false**)

Beispiel:
```
string s3 = "A", s4 = "a", s5 = "Aal";
s3 > s4;    s3 < s5;     s3 + s4 + 'l' == s5;
 False        True                    True
```

Methoden für Strings

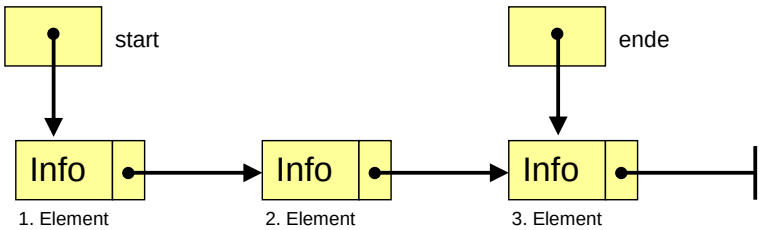
- Benutzen von Methoden mit dem Punkt-Operator
  - **`insert()`:** Zeichenkette an eine Position im String einfügen
  - **`erase()`:** Löschen einer Anzahl an Zeichen in einem String
  - **`find()`:** Suchen einer Zeichenfolge, Rückgabe derPosition
  - **`replace()`:** Ersetzen eines Teilstrings durch einen anderen String
  - **`length()`:** Anzahl der Zeichen als Rückgabewert
  - **`substr()`:** Extraktion eines Teilstrings aus einem String
  - **`at()`:** Zugriff auf ein einzelnes Zeichen in einem String
  - **`c_str()`:** Umwandeln eines Strings in einen C-String

- Nützliche Methoden in Zusammenhang mit Strings
  - **`atoi()`:** Umwandlung eines C-Strings in eine Integer-Zahl
  - **`atof()`:** Umwandlung eines C-Strings in eine Float-Zahl
  - **`strtod()`:** Umwandlung eines C-Strings in eine Double-Zahl

Einfach verkettete Liste

- Eine einfach verkettete Liste ist eine dynamische Datenstruktur

- Definition einer einfach verketten Liste
  - Listenelement (z.B. Objekt einer Klasse)
    - Besteht aus einem Informationsteil (Variablen zum Speichern der Daten)
    - Und einem Zeiger auf das nächste Listenelement
    - Jedes Listenelement hat einen Nachfolger
  - Die Liste hat zusätzlich einen Start- und Ende-Zeiger



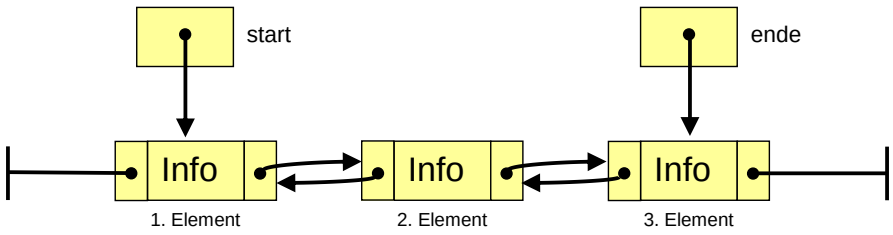
Einfach verkettete Liste: Vor- und Nachteile

- Vorteile
  - Elemente können sehr schnell am Anfang und Ende der Liste eingefügt werden
  - Im Gegensatz zu einem dynamischen Array ist das Einfügen problemlos möglich, ohne dass der komplette Datensatz bei jeder Vergrößerung umkopiert werden muss
- Nachteile
  - Es ist aufwändig nach Daten zu suchen, zu löschen und die Liste zu sortieren, da über jedes einzelne Element gegangen (iteriert) werden und das Verändern an der ersten und letzten Stelle gesondert behandelt werden muss
  - Zusätzlicher Speicherbedarf für die Zeiger

Quelle: http://de.wikipedia.org/wiki/Liste_%28Datenstruktur%29

Doppelt Verkettete Liste

- Jedes Element hat einen Zeiger auf das nachfolgende und auf das vorherige Element



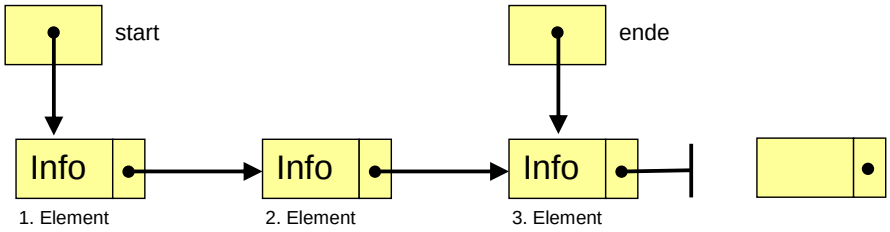
- Vorteile:
  - Die Elemente in der zweiten Hälfte einer sortierten Liste sind schneller aufzufinden
  - Fehlerhafte Verweise können erkannt werden
  - Schnelles Löschen und Einfügen von Elementen
    - Ermöglicht zum Beispiel ein effizientes Sortieren der Liste
  - Über die Liste kann von hinten nach vorne iteriert werden
- Nachteile:
  - Höherer Speicherbedarf für die zusätzlichen Zeiger
  - Das Verwalten der Liste ist komplexer

Quelle: http://de.wikipedia.org/wiki/Liste_%28Datenstruktur%29

Einfügen eines neuen Elements – Schritt 1

Vorgehensweise:

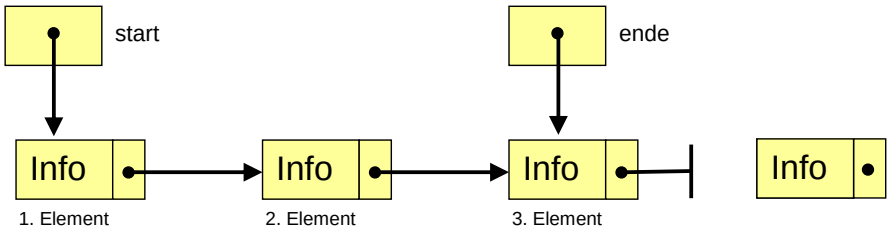
1. Neues Element erstellen – dynamisch erzeugen
2. Daten einlesen und in das neue Element speichern
3. Zeiger (auf das nächste Element) in neuen Element auf NULL setzen
4. Zeiger des letzten Elements der Liste (zu finden über den Zeiger *ende*) auf das neue Element zeigen lassen
5. Zeiger **`ende`** auf das neue Element zeigen lassen



Einfügen eines neuen Elements – Schritt 2

Vorgehensweise:

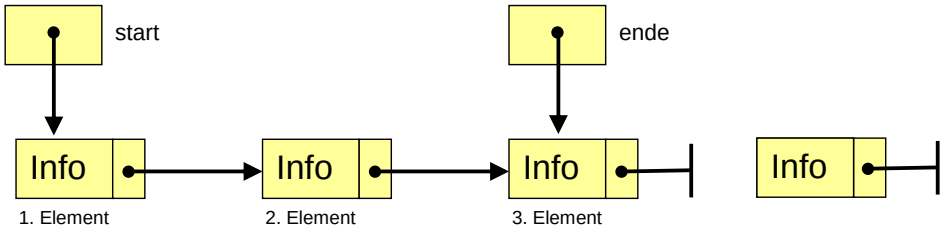
1. Neues Element erstellen – dynamisch erzeugen
2. Daten einlesen und in das neue Element speichern
3. Zeiger (auf das nächste Element) in neuen Element auf NULL setzen
4. Zeiger des letzten Elements der Liste (zu finden über den Zeiger *ende*) auf das neue Element zeigen lassen
5. Zeiger **`ende`** auf das neue Element zeigen lassen



Einfügen eines neuen Elements – Schritt 3

Vorgehensweise:

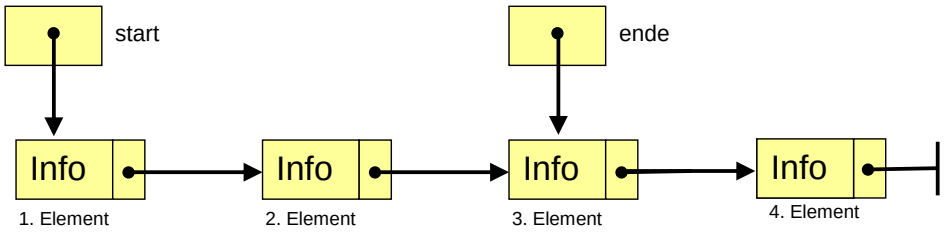
1. Neues Element erstellen – dynamisch erzeugen
2. Daten einlesen und in das neue Element speichern
3. Zeiger (auf das nächste Element) in neuen Element auf NULL setzen
4. Zeiger des letzten Elements der Liste (zu finden über den Zeiger *ende*) auf das neue Element zeigen lassen
5. Zeiger `ende` auf das neue Element zeigen lassen



Einfügen eines neuen Elements – Schritt 4

Vorgehensweise:

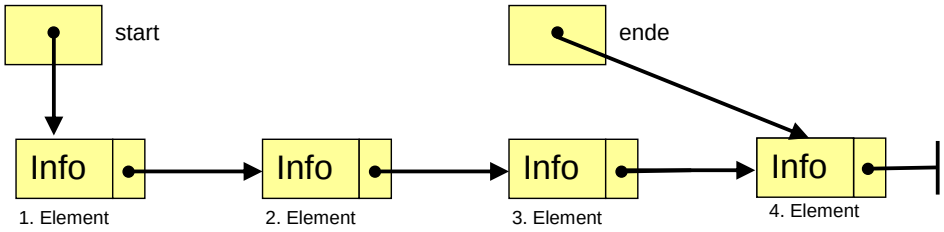
1. Neues Element erstellen – dynamisch erzeugen
2. Daten einlesen und in das neue Element speichern
3. Zeiger (auf das nächste Element) in neuen Element auf NULL setzen
4. Zeiger des letzten Elements der Liste (zu finden über den Zeiger *ende*) auf das neue Element zeigen lassen
5. Zeiger `ende` auf das neue Element zeigen lassen



Einfügen eines neuen Elements – Schritt 5

Vorgehensweise:

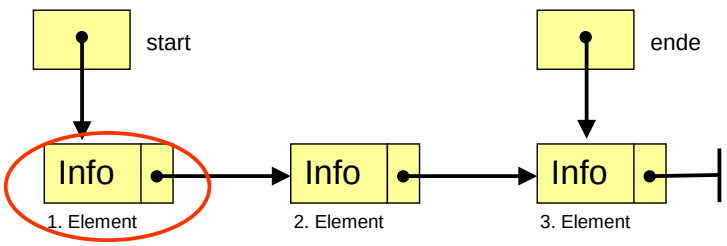
1. Neues Element erstellen – dynamisch erzeugen
2. Daten einlesen und in das neue Element speichern
3. Zeiger (auf das nächste Element) in neuen Element auf NULL setzen
4. Zeiger des letzten Elements der Liste (zu finden über den Zeiger *ende*) auf das neue Element zeigen lassen
5. Zeiger **`ende`** auf das neue Element zeigen lassen



Achtung: Wenn das allererste Element angelegt wird, muss auch der Zeiger **`start`** verändert werden

Suchen eines Elements

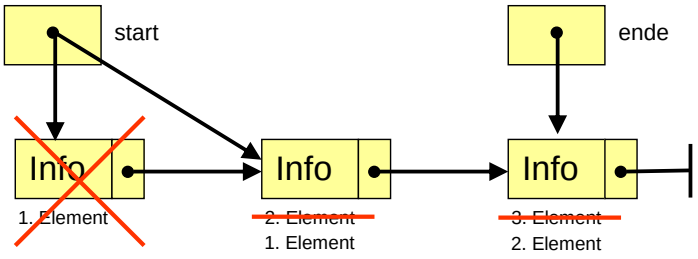
- Vorgehensweise:
  1. Über Zeiger **`start`** das erste Element finden und überprüfen
  2. Wenn nicht gefunden dann nächstes Element über den Zeiger des aktuellen Elements / Wenn gefunden Element Schleife beenden
  3. Schritt 2 solange wiederholen, bis das letzte Element überprüft ist



Löschen eines Elements(1)

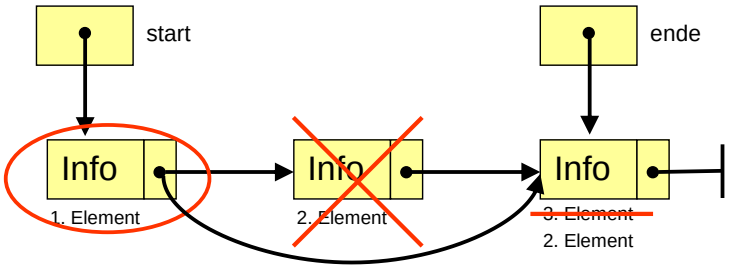
- Achtung: Beim Löschen des ersten und letzten Elements müssen die jeweiligen Zeiger **`start`** und **`ende`** auch verschoben werden
  - Im Speziellen muss beachtet werden, wenn nur noch ein Element in der Liste vorhanden ist

- Löschen am Listenbeginn
  1. Zeiger **`start`** zeigt auf das Element, auf welches das 1. Element gezeigt hat
  2. 1. Element löschen



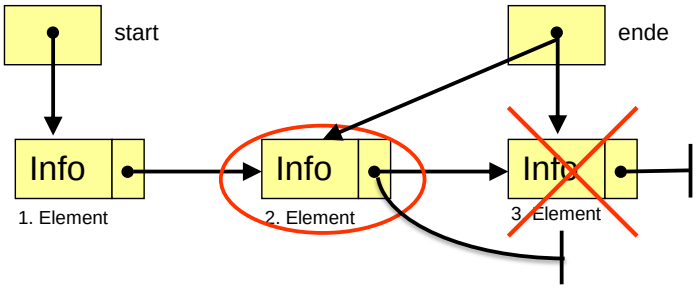
Löschen eines Elements(2)

- Löschen eines Elements mitten in der Liste
  1. Element finden welches auf das zu löschende Element zeigt
  2. Zeiger vom gefundenen Element gleich dem Zeiger des nächsten (zu löschenden) Elements setzen
  3. Zu löschendes Element löschen



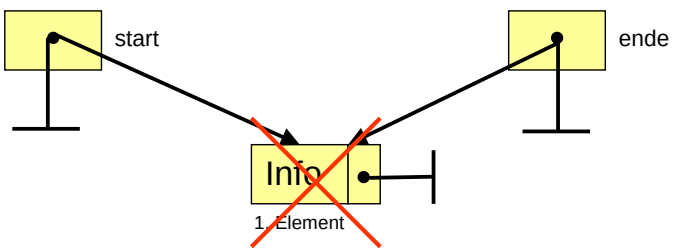
Löschen eines Elements(3)

- Löschen des letzten Listenelements
  1. Element finden welches auf das zu löschende Element zeigt
  2. Zeiger vom gefundenen Element gleich dem Zeiger des nächsten (zu löschenden) Elements setzen
  3. Zu löschendes Element löschen
  4. Zeiger **`ende`** auf das vorherige Element zeigen lassen



Löschen eines Elements(4)

- Löschen des einzigen Listenelements
  1. Setzen der Zeiger start und ende auf NULL
  2. Löschen des Elements



Ausblick

- Kompendium: Kapitel 8 – Objektorientierung
- Tutorium: Aufgabe 17

- Wie kann ich große Dateimenge in mein Programm einlesen?
- Wie kann ich Daten sortieren?
- Wie kann ich einen bestimmten Knoten in einem Graphen finden?
- …

Vielen Dank für Ihre Aufmerksamkeit

Tobias Schwalb & Michael Tansella

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