Karlsruher Institut für Technologie

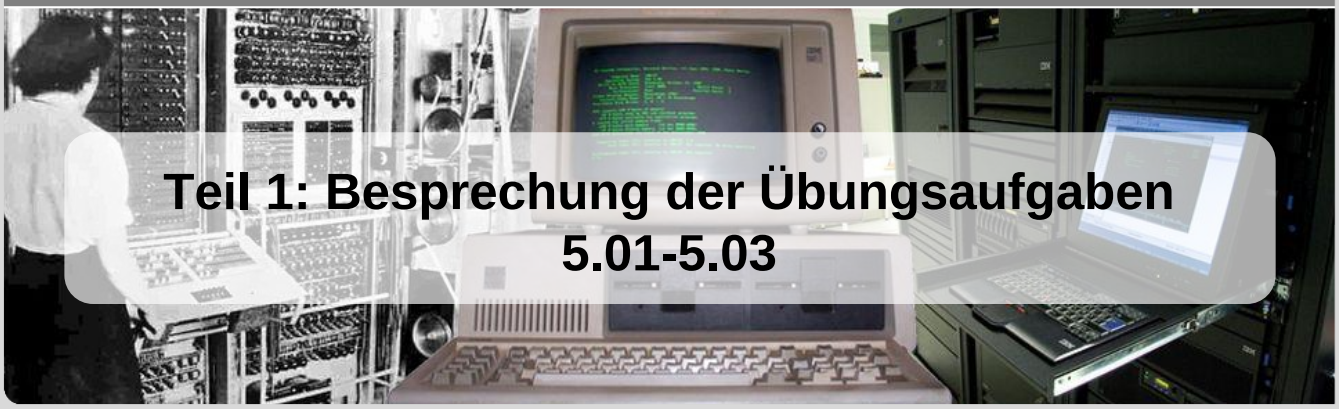
Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. Dr. h.c. J. Becker
Prof. Dr.-Ing. E. Sax
Prof. Dr. rer. nat. W. Stork

Übung05: Informationstechnik (IT)

Harald Bucher

Institut für Technik der Informationsverarbeitung (ITIV)



Teil 1: Besprechung der Übungsaufgaben 5.01-5.03

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

www.kit.edu

Inhalt: Übung05 – Teil 1

Karlsruher Institut für Technologie

1. Aufg. 5.01: Verständnisfragen
2. Aufg. 5.02: OOP, Klassendefinition und Vererbung
3. Aufg. 5.03: Stringmanipulation

Polymorphie - Problemstellung

- Problemstellung Beispiel:

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

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();
  }
  for( int i = 0; i < 3; i++ ) {
    delete autoPark[i];
  }
  return 0;
```

Wie können die Fahrzeuge richtig hupen ?

Polymorphie

- Normalerweise werden Methoden während des Kompilierens entsprechend dem Typ des Zeigers ausgewählt (*Early Binding*)
  - 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?!
- Methoden anhand des Typs des Objekts zur Laufzeit automatisch auswählen mit dem Schlüsselwort **`virtual`**
  - Methode muss in der Basisklasse als **`virtual`** deklariert werden
  - Methode kann (muss aber nicht) in den Kindklassen redefiniert/überladen werden
  - Entscheidung geschieht zur Laufzeit (*Late Binding*)
  - Besonders wichtig auch bei Destruktoren
  - Virtuelle Methoden kosten etwas mehr Speicherplatz und sind etwas langsamer – Vorteile überwiegen allerdings!

Polymorphie - virtual

- Problemstellung Beispiel:

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

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

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

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

Einzige Änderungen

```
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();
  }

  for( int i = 0; i < 3; i++ ) {
    delete autoPark[i];
  }
  return 0;
}
```

```
C:\WINDOWS\
Tuuto
Troeoet
Troeoet
```


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)

Strings und C-Strings

- C-Strings:
  - Char-Arrays zur Speicherung von Zeichenfolgen
  - Gespeichert wird: Zeichenfolge + String-Endezeichen `'\0'`
  - Spezielle Funktionen **`strlen(), strcpy(), strcmp()`**

Beispiel:
```
char name[] = "Evi"; //oder
char name[] = { 'E', 'v', 'i', '\0' };
```

- Umwandlung von Strings in C-Strings (für spezielle Funktionen):
  - Anhand der speziellen Methode **`c_str()`** der Klasse **`string`**
  - Rückgabewert dieser Methode ist vom Typ **`const char*`**

Beispiel:
```
string name = "ADAM";
double x = atof( name.c_str() );
```


Methoden für Strings

- Suchen mit Methode **`find()`**
  - Suchen einer Zeichenfolge in einem String, Rückgabe der Position
  - Gibt **`-1`** bzw. **`string::npos`** zurück, falls die Zeichenfolge nicht gefunden wurde

Zu suchender String / Startposition (optional)

Beispiel:
```
int pos = erg.find( 'a', 3 );
//erg == "Kaffeemasse"                    //pos == 7
```

- Ersetzen mit Methode **`replace()`**
  - Ersetzen eines Teilstrings durch einen anderen String

Startposition / Startposition des einzufügenden Strings (optional) / Länge zum Ersetzen / Anzahl der einzufügenden Zeichen (optional)

Beispiel:
```
erg.replace( 6, 5, "maschinencode", 0, 8 );
//erg == "Kaffeemasse"                    //erg == "Kaffeemaschine"
```

Zugriff auf Zeichen in Strings

- Zugriff mit dem Index-Operator `[]` oder Methode **`at()`**
- Zeichen wird anhand seines Index / Position identifiziert
  - Erstes Zeichen: `0` Letztes Zeichen: **`s.length()-1`**

Beispiel:

```
string s = "Tee";
char c = s[0];     //c == 'T'
//s[0] == 'T'  s[1] == 'e'  s[2] == 'e'
```

- Index muss ein Integer-Ausdruck sein
  - Beim Index-Operator erfolgt keine Fehlermeldung beim Kompilieren beim Überschreiten des Bereichs → Absturz / Fehlverhalten zur Laufzeit
  - Methode **`at()`** führt eine Bereichsprüfung durch → Auslösung einer Exception → kann abgefangen werden

Beispiel:

```
s.at( 3 ) = 'x'; //Auslösung einer Exception
```


Aufg. 5.01: Verständnisfragen Lsg. (1)

a) Die Methoden einer abgeleiteten Klasse können auf die private-Elemente der Basisklasse zugreifen. Falsch

b) Eine abgeleitete Klasse erbt eine public-Methode der Basisklasse nicht, wenn sie selbst eine public-Methode mit gleichem Namen besitzt. Falsch
(bei unterschiedlichem Prototyp oder bei gleichem Prototyp über Basisklassenoperation)

c) Methoden einer abgeleiteten Klasse können auf **`protected`** Elemente der Basisklasse zugreifen. Die **`protected`** Elemente sind aber von außerhalb der Basisklasse nicht zugreifbar. Richtig

d) Jedem Objekt einer Basisklasse kann ein Objekt einer Kindklasse zugewiesen werden und umgekehrt. Falsch

Aufg. 5.01: Verständnisfragen Lsg. (2)

e) Polymorphe Klassen werden in C++ mit Hilfe von virtual-Methoden implementiert.

f) Bei der Ausführung der Anweisungen: **`string name; cin >> name;`** werden Zeichen von der Standardeingabe eingelesen und zwar
   i. alle Zeichen einer Zeile ohne führende Zwischenraumzeichen Falsch
   ii. genau ein Wort ohne führende Zwischenraumzeichen Richtig
   iii. eine ganz Textzeile Falsch

g) Zum verketten zweier Objekte vom Typ **`string`** kann man den Operator **`+ oder +=`** verwenden.

h) Das erste Zeichen in einem String hat die Position 0.

i) Für den Zugriff auf die einzelnen Zeichen in einem **`string`**-Objekt kann der Operator **`[]`** verwendet werden.

Aufg. 5.02: OOP, Klassendefinition und Vererbung Lsg. (1)

a) Angenommen die folgenden Klassen sind in der Header-Datei **`"myClasses.h"`** definiert:

```
class Polynom {
  int x;
public:
  Polynom();
  virtual int calc( int a );
};
```

```
class Add : public Polynom {
  int y;
public:
  Add();
  int calc();
};
```

Für ein Objekt **`obj`** der Klasse **`Add`** ist dann folgende Anweisung zulässig:

```
int res = obj.calc( 2 );
```

Richtig / Falsch

Falsch:
1. Polymorphie funktioniert nur für Methoden, die die selbe Parameterliste und den selben Rückgabewert haben.
2. In der Klasse Add wird die geerbte Funktion **`calc(int)`** von der eigenen Funktion **`calc()`** überdeckt. Deshalb ist der Aufruf mit Parameter ungültig.

Info: Möglich wäre hingegen **`int res = obj.Polynom::calc( 2 );`**

Aufg. 5.02: OOP, Klassendefinition und Vererbung Lsg. (2)

b) Was gibt das folgende Programm auf dem Bildschirm aus?

```
#include <iostream>

using namespace std;

class B {
public:
  B() { cout << "Konstruktor der Klasse B \n"; }
  ~B() { cout << "Destruktor der Klasse B \n"; }
};

class D : public B {
public:
  D() { cout << "Konstruktor der Klasse D \n"; }
  ~D() { cout << "Destruktor der Klasse D \n"; }
};

class X {
private:
  D d;
public:
  X() { cout << "Konstruktor der Klasse X \n"; }
  ~X() { cout << "Destruktor der Klasse X \n"; }
};

int main() {
  X xObj;
  cout << "Bye, bye!" << endl;
  return 0;
}
```

Beim Anlegen vom Objekt **xObj** der Klasse **X** gilt die folgende Reihenfolge:

1) Anlegen des Objekts d

1.1) Aufruf von **B()** der Basisklasse

1.2) Aufruf von **D()** der Klasse **D**

2) Aufruf von **X()** der Klasse **X**

Beim Beenden des Programms wird der Destruktor aufgerufen und dann der Speicher für die angelegten Objekte freigegeben

Es gilt die umgekehrte Reihenfolge, wie beim Anlegen!

```
C:\WINDOWS\system32\cmd.exe
Konstruktor der Klasse B
Konstruktor der Klasse D
Konstruktor der Klasse X
Bye, bye!
Destruktor der Klasse X
Destruktor der Klasse D
Destruktor der Klasse B
Drücken Sie eine beliebige Taste . . .
```


Aufg. 5.03: Stringmanipulation

- Schreiben Sie ein Programm, welches in einen String alle Vorkommen eines (anderen) Strings **findstring** sucht und die jeweiligen Positionen auf dem Bildschirm ausgibt. In nächsten Schritt soll dieser String durch einen anderen **replaceString** ersetzt werden. Die Länge der Strings können dabei beliebig (auch unterschiedlich) sein. Wenn **findstring** nicht im String gefunden wird soll eine Fehlermeldung ausgegeben werden.

- Beispielausgabe

```
C:\WINDOWS\system32\cmd.exe
Text eingeben : Mein roter Koffer und meine rote Tasche
findstring    : rot
replacestring : blau

String rot gefunden an Pos: 5
String rot gefunden an Pos: 28
Ergebnis: Mein blauer Koffer und meine blaue Tasche
Drücken Sie eine beliebige Taste . . .
```

Aufg. 5.03: Stringmanipulation Lsg. (1)

```
#include <iostream>
#include <string>
using namespace std;

int main() {
  string text, findstring, replacestring;
  int i = -1;

  cout << "Text eingeben : ";
  getline( cin, text );
  cout << "findstring    : ";
  cin >> findstring;
  cout << "replacestring : ";
  cin >> replacestring;

  if( text.find( findstring, 0 ) == -1 ) {
    cout << "Fehler: findString ist nicht vorhanden\n";
    return 1;
  }
```

Einbinden der Bibliothek für String

Stringvariablen zum Speichern der Eingaben

Einlesen einer Zeile mit Leerzeichen

Einlesen eines einzelnen Worts

Überprüfen, ob die Zeichenkette vorhanden ist

Programm mit Fehlermeldung beenden

Aufg. 5.03: Stringmanipulation Lsg. (2)

```
cout << endl;

  do {
    i = text.find( findstring, i + 1 );
    if( i != -1 ) {
      cout << "String " << findstring
           << " gefunden an Pos: " << i << endl;
    }
  } while( i >= 0 );

  i = -1;
  do {
    i = text.find( findstring, i + 1 );
    if( i != -1 ) {
      text.replace( i, findstring.length(), replacestring );
    }
  } while( i >= 0 );

  cout << "Ergebnis: " << text << endl;

  return 0;
}
```

Schleife wird so oft durchlaufen, wie `i >= 0`

`findstring` suchen und Position in **`i`**
→ nicht gefunden: Rückgabe **`-1`**

Position ausgeben

Zwei getrennte Schleifen, da **`text`** sonst zu früh verändert wird und sich dadurch Positionen evtl. verschieben!

`findstring` durch **`replacestring`** ersetzen

Ergebnistext ausgeben

Referenz

- Kompendium: Kapitel 5 - Erweiterte Datentypen
  Kapitel 8 - Objektorientierung (ab 8.3)
- Tutorium: Aufgabe 12, 13, 16, 17