LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Prof. Andreas Butz | Dipl.Inf. Otmar Hilliges

Programmierpraktikum 3D Computer Grafik

Einführung in C++ Teil II:

Klassen, Objekte, Funktionen und Vererbung.

Nachtrag: Virtuelle Funktionen

- Member Funktionen können als virtuell deklariert werden
  - Virtuelle Funktionen können in abgeleiteten Klassen überschrieben werden
  - Keyword *virtual*
  - Normalerweise neue Funktionalität in abgeleiteter Klasse
- Auch nicht virtuelle Funktionen können überschrieben werden
  - Nicht virtuelle Methoden werden zur Compile-Zeit ausgewählt
  - Virtuelle Methoden werden zur Laufzeit ausgewählt

- Häufigster Grund für virtuelle Klassen: Unterschiedliches verhalten zur Laufzeit
  - Eltern-Klasse *Window*
  - Abgeleitete Klasse *Button*
  - Virtuelle Methode *Create*
  - Verwendete Methode wird zur Laufzeit ausgewählt
    - beide Instanzen werden einer Variablen vom Typ *Window* zugewiesen.

Virtuelle Funktionen Beispiel

```
class Window // Base class
{
  public:
     virtual void Create() // virtual function
     {
          cout <<"Base class Window"<<endl;
     }
};
```

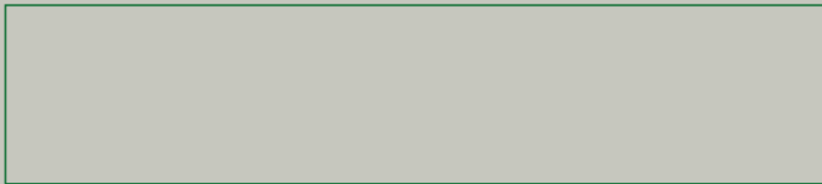
```
class Button : public Window
{
  public:
     void Create()
     {
          cout<<"Derived class Button "<<endl;
     }
};
```

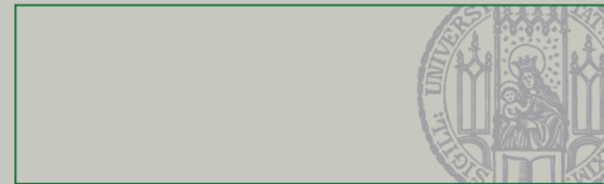
```
void main()
 {
     Window  *x, *y;

     x = new Window();
     x->Create();

     y = new CommandButton();
     y->Create();
 }

 Ausgabe:
      Base class Window
      Derived class Button
```





Präprozessor, Compiler, Linker

Compiler Collection

- Eine Sammlung von Programmen, die man zur Programmerstellung benötigt
- Präprozessor – Textersetzung in Sourcefiles vor dem eigentlichen Kompilieren
- Übersetzer [Compiler] Übersetzt Programm(fragmente) von C++ in Binärcode (maschinenlesbar)
- Verknüpfer [Linker] Setzt Binärcodefragmente zu lauffähigem Programm zusammen

Präprozessor

- Anweisungen für den Präprozessor fangen mit # an
- `#include "file.h"` fügt die Header-Dateifile .h ein
- `#define FOO 1` ersetzt überall im Programmtext `FOO` mit 1

```
#ifdef BAR
        Code 1
    #else
        Code 2
    #endif
```

- Fügt im Programmtext Code 1 ein falls `BAR` definiert wurde, ansonsten Code2

Präprozessor Anwendungen

- Schutz vor mehrfach Einbindung von Headern

```
#ifndef  _MYCLASS_H
#define _MYCLASS_H

#include …

class MyClass{…};

#endif  // _MYCLASS_H
```

- Ersetzungsworte immer in Großbuchstaben
- Präprozessor Einsatz ist unflexibel und sollte minimiert werden!

Bibliotheken (*.lib,*.so,*.dll) sind vorkompilierte Binärdateien ohne Einstiegspunkt

- Zeitvorteil, da nicht immer alles neu kompiliert werden muss
- Wiederverwendbarkeit -> kein mehrfach erfinden des Rades nötig
- Platz- und Geschwindigkeitsvorteil, da häufig benötigter Code nur einmal auf der Platte gespeichert werden muss (und teilweise auch nur einmal im Speicher)

Linker

- Das eigentliche Erstellen von Programmen wird vom Linker erledigt
- Linker ersetzt symbolische Funktionen in Binärcode mit Adressen im Speicher.
- Außerdem werden eigene Programmfragmente und verwendete Bibliotheken zusammen gesetzt.
- Zwei Arten des Verknüpfens möglich:
  - Statisches Verknüpfen
  - Dynamisches Verknüpfen

Statisches Verknüpfen:

- Einfachste Methode
- Erzeugt eine große Datei
- Bei Änderungen an der Bibliothek muss neu kompiliert werden

Dynamisches Verknüpfen:

- Komplizierter Mechanismus der auf unterschiedlichen BS verschieden funktioniert.
- Kleinere, modulare Dateien
- Bibliotheken können von mehreren Programmen gleichzeitig benutzt werden
- Bei Änderungen an der Bibliothek muss nicht neu kompiliert werden

Einbinden von fremden Bibliotheken

- Im eigenen Code den/die Header einbinden (für die Methoden Deklarationen)
- Den Linker-Suchpfad so anpassen, dass Bibliotheken gefunden werden können (lib path)
- Den Compiler-Suchpfad so anpassen, dass Headerdateien gefunden werden können (include path)

Tipps zur Fehlersuche:

- Meist wurde in den angegebenen Objektdateien ein Symbol (Funktion, Variable, Klasse, …) nicht gefunden
  - Schreibfehler
  - Falscher oder fehlender Namensraum
  - Schlüsselwörter `virtual/static` vergessen
- Der Compiler prüft ***nie*** nach, ob eine deklarierte Funktion (im Header) implementiert wurde -> Laufzeitfehler
- Implementierte Methoden ohne Deklaration geben dagegen immer einen Fehler
- SegFaults sind das äquivalent zu Nullpointer Exceptions in Java

Klassen, Objekte, Funktionen und Vererbung

- [Rückgabetyp] [Name] ([Parameter]) {…}
- Wichtig:
  - Funktionen müssen vor dem Aufruf deklariert werden, d.h. sie müssen in den Zeilen über dem Aufruf stehen
- Funktionsprototypen umgehen diese Bedingung und stellen eine Signatur dar
- Beispiel für eine Signatur:
  - `int factorial(int);`

- Enthalten Attribute und Funktionen
- Werden typischerweise in Header-Dateien deklariert
- Beispiel einer Deklaration:

```
class Person {
private:
    char* name;
    int age;
public:
    void setName(char*);
    void setAge(int);
};
```

Deklaration von Funktionen mit `inline`:

- Implementierung direkt in der Deklaration
- Zwei verschiedene Arten:
  - Direkt in der Deklaration (ohne `inline`)
  - Im Header-File nach der Deklaration (mit `inline`)
- Beispiel (direkt in der Deklaration):

```
… // statt void setAge(int);
void setAge(int age) {
        this->age = age;
}
…
```

Deklaration von Funktionen mit inline:

- Beispiel (mit `inline`):

```
class Person {
        …
        void setAge(int);}
        …
};
inline void Person::setAge(int a) {
        age = a;
}
```

Konstruktoren:

- Definiert als Methode einer Klasse
- Name ist identisch zum Namen der Klasse
- Kein Rückgabewert (auch nicht `void`)
- Mehrere Konstruktoren durch Überladen
- Deklaration im `public` Abschnitt
- Minimaler Default-Konstruktor wird erzeugt, falls kein Konstruktor vorhanden ist

Konstruktoren:

- Aufruf bei der Erzeugung durch `new` oder bei der Erzeugung eines statischen Objekts
- Minimaler Konstruktor:

```
Person::Person() { }
```

- Standard-Konstruktor:

```
Person::Person() {
        name = NULL;
        age = 0;
}
```

Konstruktoren:

```
Person::Person(char* n, int i = 0) {
        name = n;
        age = i;
}
Person::Person(char* n, int i = 0) :
        name(n),
        age(i)
        {…}
```

Beispiele für statisches Erzeugen (auf dem Stack):

```
Person::Person; // Default
Person::Person("Peter"); // age = 0
Person::Person("Peter", 45);
```

Beispiele für dynamisches Erzeugen (auf dem Heap):

```
peter* = new Person::Person;
peter* = new Person::Person("Peter");
```

Löschen von Objekten:

- Mittels Destruktor (beginnt mit ~)
- Wird aufgerufen, sobald der Gültigkeitsbereich eines Objekts verlassen wird, oder `delete` aufgerufen wird
- Minimaler Destruktor:

```
Person::~Person() {}
```

- Besitzt keinen Rückgabetyp (wie Konstruktor)
- Nur ein Destruktor möglich
- Sinn: Löschen von Unterobjekten

Zugriffsoperator:

- Mit dem Punkt-Operator .
- Bei Zeigern auf ein Objekt mit Pfeil ->

Beispiel:

```
Person peter;

Person* john = new Person;

peter.setName("Peter");

john->setAge(35);

delete john;
```

Vererbung

- Syntax:

```
class subClass :
        [modifier] superClass1,
        [modifier] superClass2,
        …
        [modifier] superClassN
{
        …
};
```

- Keine Vererbung für Konstruktoren, Destruktor und Zuweisungs-Operator

Generell gilt:

- Alles was nicht überschrieben wird, wird geerbt
- Zugriff auf eine überschriebene Methode der Superklasse (**`super`**`.[method]` in Java):
  - Scope-Operator `::`
  - Beispiel: `Employee` erbt von `Person`

```
Employee a;
a.print();
a.Person::print();
```

Beispiel:

```
class Employee : public Person {…
        void print() {
                // print(); // endlos
                Person::print();
                cout << “blah” << endl;
        }
};
Employee a;
a.print();
a.Person::print();
```

Konstruktoren:

- Elemente der Basisklasse werden durch Konstruktor der Basisklasse initialisiert
- Initialisierungsliste:

```
class_name::class_name(parameter_list)
        : superClass1(parameters),
          superClass2(parameters), …
```

- Konstruktoren der Basisklasse werden **vor** dem Konstruktor der abgeleiteten Klasse ausgeführt!

Destruktoren:

- Löschen von Elementen der Basisklasse **muss** in der Basisklasse erfolgen
- Destruktor der Basisklasse wird **automatisch** aufgerufen **nach** dem Destruktor der abgeleiteten Klasse

- Zeiger auf Objekte:
  - Zeiger auf ein Objekt einer abgeleiteten Klasse kann einem Zeiger auf eine Basisklasse zugewiesen werden:
    - Unterklasse ist eine Erweiterung (*is-a* Beziehung)
  - Umkehrung funktioniert das nicht:
    - Typecasting notwendig
  - Allgemeine Regel:
    - Speziellere Typen (abgel. Klasse) können einem allgemeinerem Typ (Basisklasse) zugewiesen werden
    - Dabei wird das Objekt automatisch auf die Komponenten der Basisklasse reduziert.

- Typecasting:
  - **reinterpret_cast**<*type*>(*parameter*);
    - Keine Überprüfungen, keine Wertänderung
    - Zeigertyp in beliebigen anderen Zeigertyp ändern
  - **static_cast**<*type*>(*parameter*);
    - Implizit mögliche Casts und Umkehrung möglich
    - Basisklasse <-> abgel. Klasse, `int` <-> `char`
    - Keine echte Überprüfung auf Typkompatibilität
  - **dynamic_cast**<*type*>(*parameter*);
    - Laufzeittyp wird bei der Überprüfung verwendet
    - Wie das Casting in Java

- Potenzielle Probleme:
  - Oft werden Methoden der Basisklasse überschrieben (um Konsitenz sicher zu stellen)
  - Wird ein Objekt der abgeleiteten Klasse als Objekt der Basisklasse verwendet kann das zu Fehlern füren
    - Beim Aufruf wird die Methode der Basisklasse aufgerufen nicht die der erweiterten.
- Gründe:
  - Statisches vs. dynamisches Binden.
    - Der C++ Compiler bindet per default statisch (Typ des Objekts wird zur Compilezeit festgelegt)
    - Das gilt auch für Zeiger und Referenzen weisen wir ein abgeleitetes Objekt einem Pointer vom Typ der Basisklasse zu wird dieses vom Compiler auf ein Basisobjekt reduziert.

- Lösung:
  - Ausführung einer geeigneten Methode der Unterklasse (ohne diese explizit zu kennen (bei Java immer so)
  - Typ des Objekts wird erst zur Laufzeit bestimmt -> dadurch wird immer die richtige Methode aufgerufen.
  - Realisierung mit dem Schlüsselwort `virtual` in der Basisklasse
  - Speicherverbrauch steigt (Objekt enthält Zeiger auf die *vtable* der Klasse)
- Faustregeln:
  - Methoden der Basisklasse die überschrieben werden können/sollen sollten immer virtuell sein
  - Klassen die nicht virtuelle Methoden haben - und wenn diese überschrieben werden müssen, sollten *niemals* als Basisklasse dienen.

- Beispiel in Java:

```
class Person {…
        public void print();
}
class Employee extends Person {…
        public void print();
}
Person p = new Person();
Person pe = new Employee();
p.print();      // Person.print();
pe.print();     // Employee.print();
```

- Beispiel in C++ (ohne virtuelle Methoden):

```
class Person {…
        public: void print();
};
class Employee : public Person {…
        public: void print();
};
Person* p = new Person;
Person* pe = new Employee;
p.print();      // Person::print();
pe.print();     // Person::print();
```

- Beispiel in C++ (mit virtuellen Methoden):

```
class Person {…
        public: virtual void print();
};
class Employee : public Person {…
        public: void print();
};
Person* p = new Person;
Person* pe = new Employee;
p->print();    // Person::print();
pe->print();   // Employee::print();
```

Virtuelle Destruktoren:

- Dynamisch erzeugte Objekte können einem Zeiger der Oberklasse zugewiesen werden
- Wird das Objekt gelöscht, wird nur der Destruktor der Oberklasse aufgerufen
- **Lösung:** virtueller Destruktor
- Das Schlüsselwort `virtual` wird in der Basisklasse vor dem Destruktor angegeben

```
class class_name {
        virtual ~class_name() {…}
};
```

Zusammenfassung

- Eine abgeleitete Klasse besitzt alle Eigenschaften der Basisklasse (plus Ergänzungen)
- *is-a* Beziehung: Ein Objekt der abgeleiteten Klasse ist immer auch ein Objekt der Basisklasse
- Ein Objekt einer abgeleiteten Klasse darf immer als Objekt der Basisklasse verwendet werden (es reduziert sich dann auf die Basiseigenschaften)
- Konstruktoren und Destruktoren werden nicht vererbt – aber implizit aufgerufen.
- Konstruktoren werden *top-down* aufgerufen.
- Destruktoren werden *bottom-up* aufgerufen.
- Initialisierungslisten dienen zur Weitergabe von Parametern

Mehrfachvererbung

- Unterklasse hat mehrfache Basisklassen
- Unterklasse enthält jede Basisklasse
- Konstruktor der abgel. Klasse kann Konstruktoren von allen Basisklassen aufrufen
- Wird ein Objekt einer abgel. Klasse vernichtet, werden die Destruktoren aller Basisklassen aufgerufen

Beispiel:

```
class Base1 {…
  public: Base1(int, char*);
};
class Base2 {…
  public: Base2(int, float);
};
class Derived : public Base1,
                         public Base2 {…
  public: Derived(char* s, int i) :
        Base1(i, s), Base2(i, 4.2f) {…}
};
```

Problem: Mehrdeutigkeiten bei Namenskollisionen

zwei oder mehr Basisklassen haben gleiches Element:

- Member-Variablen
- Methoden (gleicher Name und Parameter)
- Nutzlos: `private`

Scope-Operator `::` als Lösung:

- Methode überschreiben und gewünschte Methode der Basisklasse mit `::` auswählen

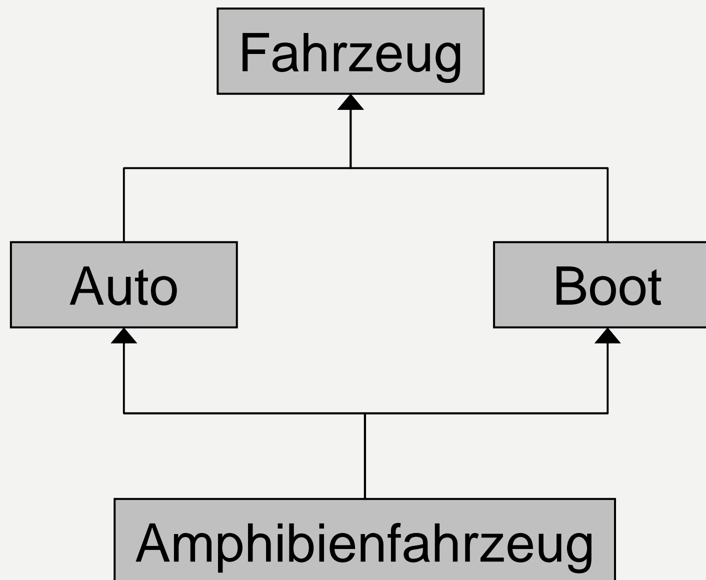
Beispiel:

- *Amphibienfahrzeug* ist sowohl *Auto* als auch *Boot*
- Deklarieren beide Basisklassen eine Methode `move(int d)` gibt folgender Aufruf einen Fehler:

```
Amphibienfahrzeug a;
a.move(10);
```

- Es ist ein mehrdeutiger Aufruf, da nicht klar ist, ob `move` der Klasse `Boot` oder der Klasse `Auto` aufgerufen werden soll

Mehrfachvererbung



- Klasse `Fahrzeug` wird mehrfach eingebunden und damit auch ihre Attribute und Methoden

- **Lösung:** Erben mit dem Schlüsselwort `virtual` führt dazu, dass die Komponenten nur einmal eingebunden werden
- Zugriff ohne Bereichsoperator eindeutig und deshalb möglich

Fehlerbehandlung durch Ausnahmen

Ausnahmen

- Drei Schlüsselwörter:
  - Versuchsblock: `try`
  - Ausnahme erzeugen: `throw`
  - Abfangen: `catch`
- Es existiert kein `finally`
- Exceptions sind nicht Teil der Methodensignatur, d.h. sie können überall geworfen werden
- Fangen aller Exceptions: `catch(...)`

Ausnahmen

- Funktionen können eine Exception-Liste angeben
- Das Schlüsselwort `throw` im Funktionsprototyp:

```
return_type method_name(parameters)
        throw (exception_list) {…}
```

- Funktionen können dennoch weitere Exceptions erzeugen (auch wenn diese nicht in der Liste enthalten sind)

Abfangen von Ausnahmen:

```
try
{
    …
    if(error) throw exception_class(…);
}
catch(exception_class variable)
{
    // exception handling
}
```

Nächste Woche:

- GLUT
- Erzeugen eines Fensters
- Erkennen und Verarbeiten von Tastatur- und Mauseingaben
- Start mit einfachen Objekten in OpenGL
- Grundlagen der 3D-Computergrafik