

Programmierkurs

Vorlesung 10

Dr. Ole Klein

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg

29. Januar 2021

Konzepte

Standard-Konzepte für Code Reuse:

- ▶ Polymorphie/Vererbung

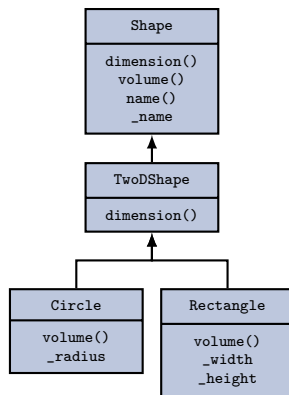
- ▶ Funktionalität wird für Basisklasse geschrieben.
- ▶ Akzeptiert auch Objekte von abgeleitetem Typ.
- ▶ Beschränkt auf objektorientierte Programmierung.
- ▶ Optional: Laufzeit-Polymorphie

- ▶ Templates

- ▶ Der Typ selbst wird ein Parameter.
- ▶ Akzeptiert jeden Typ, für den der geschriebene Code kompiliert werden kann.
- ▶ Compilezeit-Polymorphie

Vererbung

- ▶ Klassen können von anderen Klassen erben.
- ▶ Wichtigste Regel: **is-a**
Is a circle a shape?
- ▶ Abgeleitete Klasse enthält alle Variablen und Methoden der Basisklasse.
- ▶ Methoden können überschrieben werden.
- ▶ Variablen vom Typ der Basisklasse können Objekte von abgeleiteten Klassen zugewiesen werden.
- ▶ Erweitern der Basisklasse um zusätzliche Funktionalität.



Verwendung von Klassenhierarchien

- ▶ Referenzen und Pointer auf Basisklassen funktionieren auch mit abgeleiteten Klassen:

```
Circle c(...);  
Shape& s_ref = c;
```

- ▶ Beim Kopieren von Objekten werden nur die enthaltenen Daten der Basisklasse kopiert, Informationen aus abgeleiteten Klassen gehen verloren:

```
// only copies member variable _name  
Shape s_copy = c;
```

- ▶ Eine Referenz auf die Basisklasse hat nur Zugang zu den Methoden und Variablen der Basis:

```
s_ref._name; // ok  
s_ref._radius // compile error
```

- ▶ Aufgerufene Funktionen sind immer aus der Basisklasse:

```
c.volume() // calls Circle::volume()  
s_ref.volume() // calls Shape::volume()
```

Dynamische Polymorphie

- ▶ Idee: Beim Aufruf einer Methode die Implementierung aus der abgeleiteten Klasse verwenden:

```
Circle c(...);  
Shape& s_ref = c;  
s_ref.volume(); // calls Circle::volume()
```

- ▶ Funktioniert mit **virtual** Funktionen:

```
class Shape {  
    virtual double volume() const;  
    // always make destructor virtual as well!  
    virtual ~Shape();  
};
```

- ▶ Methode ist dadurch auch in allen abgeleiteten Klassen **virtual**.
- ▶ Funktioniert nur mit Pointern / Referenzen:

```
s_ref.volume(); // calls Circle::volume()  
Shape s_copy = c;  
s_copy.volume(); // calls Shape::volume()
```

Dynamische Polymorphie: Pitfalls

- ▶ Keyword `virtual` ist in abgeleiteten Klassen implizit, aber die Redeklaration ist erlaubt.
- ▶ Methoden-Signatur in abgeleiteten Klassen muss `exakt` identisch sein, inklusive `const`-Deklarationen:

```
class Circle {  
    // does NOT override the volume() method in Shape!  
    // (we forgot the const)  
    virtual double volume();  
}
```

- ▶ Besser: `override`, um Tippfehler zu vermeiden:

```
class Circle {  
    double volume() const override; // ok  
    // compile error: no virtual function defined in base class  
    double volume() override;
```

- ▶ Immer auch den Destruktor `virtual` machen, ansonsten oft Speicherlücken und ähnliche Probleme!

Speichern von polymorphen Objekten

- ▶ Bei Verwendung ist der exakte Typ (und Speicherbedarf) nicht bekannt: `sizeof(Circle) != sizeof(Rectangle)`
- ▶ Container brauchen Objekte fixer Grösse
- ▶ Wie legen wir eine Liste mit unterschiedlichen Objekten an?

Speichern von polymorphen Objekten

- ▶ Bei Verwendung ist der exakte Typ (und Speicherbedarf) nicht bekannt: `sizeof(Circle) != sizeof(Rectangle)`
- ▶ Container brauchen Objekte fixer Grösse
- ▶ Wie legen wir eine Liste mit unterschiedlichen Objekten an?

Lösung: Dynamische Speicherverwaltung

- ▶ Wir speichern eine Liste von **Pointern** auf Objekte und lassen uns dynamischen Speicher für das eigentliche Objekt geben
- ▶ Man spricht davon, dass das Objekt auf dem **Heap** angelegt wird, normale Variablen liegen auf dem **Stack**
- ▶ Objekte auf dem Heap werden **NICHT** automatisch aufgeräumt, wenn das aktuelle Scope endet
- ▶ Bei manueller Verwaltung: Speicher geht eventuell verloren
- ▶ Daher: Immer **smart pointer** verwenden

Smart Pointers

- ▶ Ein Smart Pointer reserviert Speicher für ein Objekt auf dem Heap und räumt das Objekt auf, wenn es nicht mehr verwendet wird.
- ▶ `unique_ptr` erzeugt das neue Objekt beim Anlegen und gibt es frei, sobald die Pointer-Variable aufhört zu existieren:

```
#include <memory>  
  
std::unique_ptr<int> foo(int i) {  
    return std::make_unique<int>(i);  
}  
  
int add(int a, int b) {  
    auto p = foo(a);  
    return *p + b;  
} // memory gets freed here
```

- ▶ `unique_ptr` kann nur verschoben werden, nicht kopiert

Smart Pointers für geteilte Objekte

- ▶ Oft ist es nicht möglich, einen eindeutigen Eigentümer für ein Objekt festzulegen.
- ▶ Hierfür gibt es `shared_ptr`.
- ▶ Mehrere `shared_ptr` können auf das gleiche Objekt zeigen.
- ▶ Das Objekt wird genau dann freigegeben, wenn der letzte `shared_ptr` auf das Objekt zerstört wird.
- ▶ **Wichtig:** `shared_ptr` immer nur mit `make_shared` anlegen oder aus anderen `shared_ptr`n kopieren!

```
#include <memory>

std::shared_ptr<int> foo(int i) {
    return std::make_shared<int>(i);
}

int add(int a, int b) {
    auto p = foo(a);
    auto p2 = p;
    return *p + *p2 + b;
} // memory gets freed here
```

Dynamische Polymorphie: Fazit

- ▶ Programm entscheidet zur Laufzeit, welche Methode ausgeführt wird.
 - ▶ Vorteil: Hohe Flexibilität (die gleiche Funktion kann zur Laufzeit für zwei Objekte unterschiedlichen Typs jeweils die richtige Methode aufrufen).
 - ▶ Nachteil: Laufzeit-Overhead (die richtige Methode muß zur Laufzeit identifiziert werden).
- ▶ Erfordert Planung und Disziplin beim Programm-Design:
 - ▶ Gemeinsame Hierarchie für alle Klassen.
 - ▶ Gemeinsame Funktionalität muß in Basisklasse vorgesehen sein (`virtual`-Deklarationen).
 - ▶ Vorhandene Klassen (z.B. aus Standard Library) nicht integrierbar.