Programmierkurs
Vorlesung 7

Dr. Ole Klein

Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Universität Heidelberg

18. Dezember 2020

Code Reuse

- Motivation
- Konzepte
- Vererbung (Idee)
- Templates

Worum geht es?

- Datenstrukturen für mehrere Typen

```
int_vector v1;
Point_vector v2;
...
```

Worum geht es?

- Datenstrukturen für mehrere Typen

```
int_vector v1;
Point_vector v2;
...
```

- Algorithmen für mehrere Typen

```
double array[10];
int_vector vec;
Point_list list;
...
// reverse order of entries
reverse_double_array(array);
reverse_int_vector (vec);
reverse_Point_list(list);
```

Worum geht es?

- Datenstrukturen für mehrere Typen

```
int_vector v1;
Point_vector v2;
...
```

- Algorithmen für mehrere Typen

```
double array[10];
int_vector vec;
Point_list list;
...
// reverse order of entries
reverse_double_array(array);
reverse_int_vector (vec);
reverse_Point_list(list);
```

Implementierung der Klassen / Funktionen jeweils fast identisch

Worum geht es?

- Datenstrukturen für mehrere Typen

```
std::vector<int> v1;
std::vector<Point> v2;
...
```

- Algorithmen für mehrere Typen

```
double array[10];
std::vector<int> vec;
std::list<Point> list;
...
// reverse order of entries
std::reverse(begin(array),end(array));
std::reverse(begin(vec),end(vec));
std::reverse(begin(list),end(list));
```

Worum geht es?

- Datenstrukturen für mehrere Typen

```
std::vector<int> v1;
std::vector<Point> v2;
...
```

- Algorithmen für mehrere Typen

```
double array[10];
std::vector<int> vec;
std::list<Point> list;
...
// reverse order of entries
std::reverse(begin(array),end(array));
std::reverse(begin(vec),end(vec));
std::reverse(begin(list),end(list));
```

DRY-Prinzip: Don't repeat yourself

Code Reuse

Funktionalität nach Möglichkeit nur einmal schreiben

- Zeitersparnis
- geringerer Wartungsaufwand
- Abstraktion vom konkreten Fall führt oft zu lesbarerem Code

Code Reuse

Funktionalität nach Möglichkeit nur einmal schreiben

- Zeitersparnis
- geringerer Wartungsaufwand
- Abstraktion vom konkreten Fall führt oft zu lesbarerem Code

ABER

Funktionalität spezialisieren falls nötig

- Zusatzfunktionen
- Workarounds manchmal nötig
- Performance beachten

Konzepte

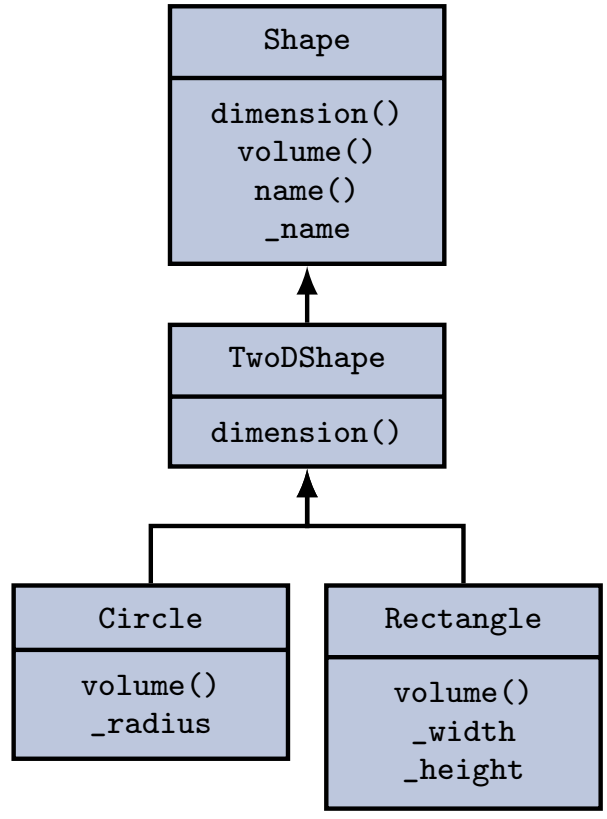
Standard-Konzepte für Code Reuse:

- Objektorientierte Programmierung
  - Komposition
    - Komplexe Objekte aus einfachen zusammensetzen
    - Konstruktives Prinzip
    - Bausteine als unveränderliche *black boxes*
  - Vererbung
    - Gemeinsame Funktionalität in Basisklasse
    - Abgeleitete Klassen können Funktionalität überschreiben
    - Optional: Laufzeit-Polymorphie

- Templates
  - Klassen und Funktionen, bei denen man zur Compilezeit *Typen* als Parameter angeben kann.
  - Template und Parameter-Typen nur schwach gekoppelt
  - Compilezeit-Polymorphie
  - Alle Informationen zur Compilezeit bekannt $\Rightarrow$ optimaler Code

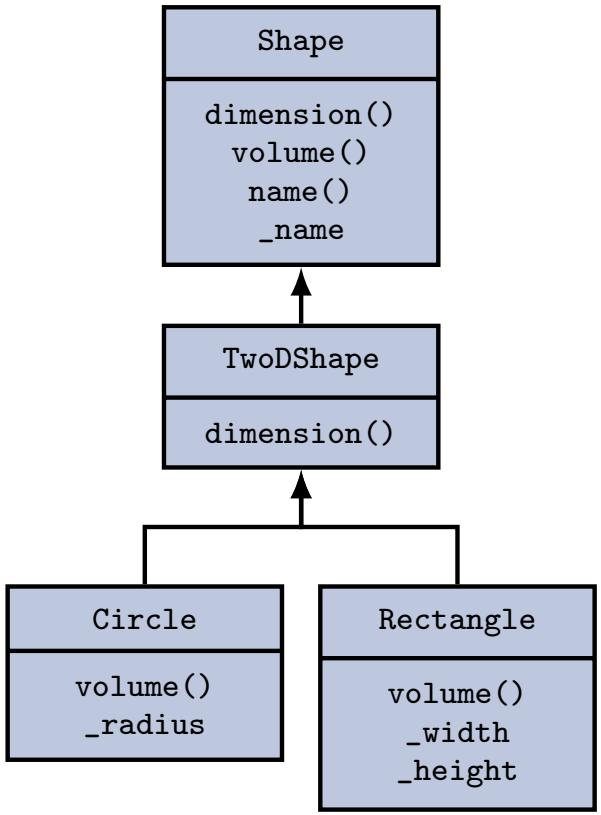
Vererbung

- 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.



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.



dazu später mehr

Templates: Motivation

Beobachtung

Oft identischer Code für unterschiedliche Typen:

```
int max(int a, int b) {
  return a > b ? a : b;
}

double max(double a, double b) {
  return a > b ? a : b;
}
```

Templates: Motivation

Beobachtung

Oft identischer Code für unterschiedliche Typen:

```
int max(int a, int b) {
  return a > b ? a : b;
}

double max(double a, double b) {
  return a > b ? a : b;
}
```

Idee

Vorlage mit Typ als Parameter:

```
SOMETYPE max(SOMETYPE a, SOMETYPE b) {
  return a > b ? a : b;
}
```

Templates: Umsetzung

Frage

Wie Version für `int`, `double`, ... erzeugen?

- Externes Programm / Präprozessor
  - (Keine Sprachunterstützung nötig)
  - Namensgebung der Varianten?
  - Welche Varianten werden benötigt?
- Compiler (Templates)
  - Automatische Generierung aller benötigten Varianten
  - Keine unterschiedlichen Namen nötig
  - Neue Syntax erforderlich

Klassentemplates

- Syntax:

```
template<typename OneType, typename T2, int size, ...>
class MyTemplate
{
  // Parameters work like normal types and constants
  // within template
  std::array<OneType,size> _var1;
  void foo(const T2& t2);
};
```

- Typ-Parameter: Statt `typename` auch `class` erlaubt:

```
template<class T> class MyTemplate;
```

- Wert-Parameter
  - Erlaubte Typen: Eingebaute Integer (`int`, `long`, `bool`, ...).
  - Beim Verwenden des Templates müssen Werte zur Compile-Zeit bekannt sein:

```
MyTemplate<int,double,3> mt1; // ok
int size = 3; // value only known at runtime
MyTemplate<int,double,size> mt1; // compile error
```

Funktionstemplates

- Syntax:

```
template<typename T1, typename T2, ...>
T2 myFunction(const T1& t1, const T2& t2) {
  return t1.size() + t2.size();
}
```

- Aufruf:

```
std::vector<int> v1; std::vector<double> v2;
myFunction<std::vector<int>,std::vector<double>>(v1,v2);
```

- Compiler kann Template-Argumente von Laufzeit-Argumenten ableiten:

```
myFunction(v1,v2); // identical to above
```

- Wichtig: Compiler darf nie mehr als ein gültiges Template finden!
- `using namespace std;` gefährlich wegen vieler enthaltener Templates.

Template-Instantiierung

- Templates werden nur auf grundlegende Syntaxfehler geprüft, nicht kompiliert (erscheinen nicht in .o-Datei).
- Compiler erzeugt Template-Instanz bei Benutzung:

```
std::vector<int> vi; // Erzeugt Code für std::vector<int>
std::vector<double> vi; // Erzeugt Code für std::vector<double>
```

- Instantiierung eines Template:
  - Für einen kompletten Satz von Template-Argumenten.
  - Benötigt Zugriff auf Template-Definition.
  - Verschiedene Instantiierungen sind für C++ verschiedene Typen.
  - Kann zu Compile-Fehlern führen.
- Templates werden in jeder Translation Unit (.cc-Datei) einzeln instantiiert
  $\Rightarrow$ Erhöhter Compile-Aufwand
- Implementierung muß beim Instantiieren sichtbar sein!
  $\Rightarrow$ Gesamter Code in Header, keine .cc-Datei

Template-Instantiierung

- Templates werden nur auf grundlegende Syntaxfehler geprüft, nicht kompiliert (erscheinen nicht in .o-Datei).
- Compiler erzeugt Template-Instanz bei Benutzung:

```
std::vector<int> vi; // Erzeugt Code für std::vector<int>
std::vector<double> vi; // Erzeugt Code für std::vector<double>
```

- Instantiierung eines Template:
  - Für einen kompletten Satz von Template-Argumenten.
  - Benötigt Zugriff auf Template-Definition.
  - Verschiedene Instantiierungen sind für C++ verschiedene Typen.
  - Kann zu Compile-Fehlern führen.
- Templates werden in jeder Translation Unit (.cc-Datei) einzeln instantiiert
  $\Rightarrow$ Erhöhter Compile-Aufwand
- Implementierung muß beim Instantiieren sichtbar sein!
  $\Rightarrow$ **Gesamter Code in Header, keine .cc-Datei**

Concepts

- Templates akzeptieren prinzipiell jeden Typ (*duck typing*)
- Impliziter Vertrag zwischen Template und Argumenten:

```
template<typename T>
int size(const T& t) {
  return t.size();
}
```

- Argument muß Methode `int size() const` besitzen.
- In der Standard-Library Anforderungen oft in Concepts zusammengefasst:

  Copy-Constructible hat Copy-Konstruktor
  Default-Constructible hat Default-Konstruktor
  Sequence Container verhält sich wie `vector` etc.

  ...

- Wird nicht explizit geprüft, sondern führt bei Verwendung fehlender Funktionen zu schwer lesbaren Compilefehlern.
  $\Rightarrow$ Concepts als Sprachfeature in C++20

Typedefs / Aliases

- Neuen Namen für existierenden Typ vergeben:

```
typedef oldtype newtype; // C-compatible syntax
using newtype = oldtype; // new syntax (more readable)
```

- Oft in Template-Kontext verwendet:

```
template<typename T>
struct Vector {
  using Element = T;
};
...
using IntVector = Vector<int>;
IntVector::Element e = 2; // same as int e = 2;
```

Type Aliases in Templates

- Beim Parsen von geschachtelten Namen in Templates weiß der Compiler nicht automatisch, ob es sich um einen Typ oder eine Member-Variable handelt.
- Standardmässig nimmt der Compiler an, dass eine Member-Variable vorliegt.
- Wenn man einen geschachtelten Typ in einer Template verwenden will, muss man `typename` davor schreiben:

```
template<typename V>
typename V::value_type add(const V& vec) {
  // compile error in next line without "typename"
  typename V::value_type sum = 0;
  for (int i = 0 ; i < v.size() ; ++i)
    sum += v[i];
  return sum;
}
```

Keyword `auto`

- Zugriff auf type aliases in Template-Parametern oft umständlich:

```
typename T1::ScalarProduct::NormType s;
s = t1.scalarProduct().norm();
```

- `auto` *rät* Variablentypen nach gleichen Regeln wie Template-Instantiierung:

```
auto S = t1.scalarProduct().norm();
```

  - Typ deduziert aus Rückgabewert.
  - Standardmäßig immer value type (kopiert Rückgabewert).
  - Nach Bedarf mit `&` und `const` qualifizieren.
  - Kann auch für Rückgabewert von Funktionen verwendet werden.

Keyword auto: Beispiel

```
template<typename V>
// let the compiler deduce the return type
auto sum(const V& v)
{
  // create a variable with the type of the elements
  // in the container
  auto sum = v[0];
  // set it to zero
  sum = 0;
  // sum over all entries
  for (auto e : v)
    sum += e;
  return sum;
}
```

Templates: Beispiel

```
class Rectangle {
  ...
  double volume() { return _width * _height; }
};

class Circle {
  ...
  double volume() { return pi*_r*_r; }
};

template<typename Shape>
bool checkEmpty(const Shape& shape) {
  return shape.volume() == 0;
}

int main(int argc, char** argv) {
  Shape s;
  Circle c(radius);
  checkEmpty(s); // ruft Shape::volume() auf
  checkEmpty(c); // ruft Circle::volume() auf
}
```