Programmierkurs

Ole Klein Steffen Müthing

Interdisciplinary Center for Scientific Computing, Heidelberg University

January 12, 2018

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: Don't repeat yourself

Code Reuse

Funktionalität nach Möglichkeit nur einmal schreiben

- Zeitersparnis
- Wartungsaufwand
- Abstraktion vom konkreten Fall führt oft zu klarerem Code

Code Reuse

Funktionalität nach Möglichkeit nur einmal schreiben

- Zeitersparnis
- Wartungsaufwand
- Abstraktion vom konkreten Fall führt oft zu klarerem Code

ABER

Funktionalität spezialisieren falls nötig

- eingeschränkte Funktionalität eines Typs
- Performance

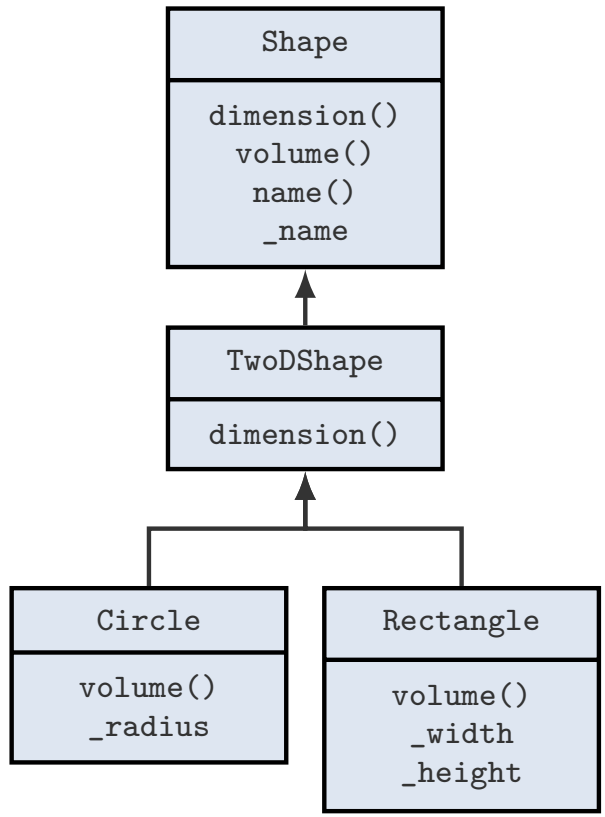
Konzepte

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 werden immer aus der Basisklasse genommen:

```
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 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!

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.

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 in 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**,...).
  - Werte beim Verwenden der Template müssen zur Compile-Zeit bekannt sein:

```
MyTemplate<int,double,3> mt1; // ok
int size = 3; //
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 eine gültige Template finden!
- `using namespace std` gefährlich wegen vieler enthaltener Templates.

Instantiierung

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

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

- Instantiierung einer 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!
  $\Rightarrow$ Aller 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
  - Default-Constructible
  - Sequence Container
  - ...
- Wird nicht explizit geprüft, sondern führt zu schwer lesbaren Instantiierungsfehlern.
  $\Rightarrow$ 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;
```

- Wenn man Aliases von Template-Parametern verwendet, muss man **typename** davor schreiben:

```
template<typename V>
typename V::value_type add(const V& vec) {
  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.

- Beispiel:

```
template<typename V>
auto sum(const V& v)
{
  auto sum = v[0];
  sum = 0;
  for (auto e : v)
    sum += e;
  return sum;
}
```

Vergleich

- Dynamische Polymorphie
  - entscheidet zur Laufzeit über ausgeführten Code.
  - erlaubt z.B. einen Container mit Pointern auf verschiedene Klassen.
  - funktioniert nur mit Pointern und Referenzen.
  - benötigt eine Klassenhierarchie.
  - einfacher zu schreiben und guter Tool-Support (IDEs).
- Templates (statische Polymorphie)
  - entscheiden zur Compilezeit über ausgeführten Code (schnell!).
  - eine Instantiierung pro Set von Template-Argumenten.
  - funktionieren mit beliebigen Typen.
  - schwieriger zu schreiben (Fehlermeldungen) und kein Tool-Support.

Vergleich

- Dynamische Polymorphie
  - entscheidet zur Laufzeit über ausgeführten Code.
  - erlaubt z.B. einen Container mit Pointern auf verschiedene Klassen.
  - funktioniert nur mit Pointern und Referenzen.
  - benötigt eine Klassenhierarchie.
  - einfacher zu schreiben und guter Tool-Support (IDEs).
- Templates (statische Polymorphie)
  - entscheiden zur Compilezeit über ausgeführten Code (schnell!).
  - eine Instantiierung pro Set von Template-Argumenten.
  - funktionieren mit beliebigen Typen.
  - schwieriger zu schreiben (Fehlermeldungen) und kein Tool-Support.

Konzepte ergänzen sich, je nach Anforderungen wählen.

Polymorphie: Beispiel

```
class Shape {
  virtual double volume() { return 0; }
  // immer virtuellen Destruktor definieren
  virtual ~Shape() = default;
};

class Circle : public Shape {
  ...
  double volume() override { return pi*_r*_r; }
};

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
}
```

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
}
```