Programmierkurs
Vorlesung 5
Templates, Code Reuse, Iteratoren

Andreas Naumann

Institut für Wissenschaftliches Rechnen
Universität Heidelberg

31. März 2023

Projektabgabe

Code Reuse
- Motivation
- Konzepte
- Vererbung (Idee)
- Templates

Iteratoren
- Motivation
- Pointer
- Pointer als Handles
- Iteratoren

Algorithmen

Projektabgabe

- Anmeldung bei `edu.ziti.uni-heidelberg.de` (mit uni-id)
- branch, tag, Mergerequests
- Grundaufgaben: Vektoren, Klasse erstellen, erweitern, Aus- und Eingabe, cmake

Worum geht es?

- Datenstrukturen für mehrere Typen

```
std::array<double, 10> ad;
std::vector<int> vi;
std::vector<Point> vp;
...
```

Worum geht es?

- Datenstrukturen für mehrere Typen

```
std::array<double, 10> ad;
std::vector<int> vi;
std::vector<Point> vp;
...
```

- Algorithmen für mehrere Typen

```
std::array<double, 10> array;
std::vector<int> vi;
std::vector<Point> vp;
...
// reverse order of entries
void reverse_array(std::array<double, 10>&);
void reverse_vector_d(std::vector<int>&);
void reverse_vector_Point (std::vector<Point>&);
```

Worum geht es?

- Datenstrukturen für mehrere Typen

```
std::array<double, 10> ad;
std::vector<int> vi;
std::vector<Point> vp;
...
```

- Algorithmen für mehrere Typen

```
std::array<double, 10> array;
std::vector<int> vi;
std::vector<Point> vp;
...
// reverse order of entries
void reverse_array(std::array<double, 10>&);
void reverse_vector_d(std::vector<int>&);
void reverse_vector_Point (std::vector<Point>&);
```

Implementierung der Klassen / Funktionen jeweils fast identisch

Worum geht es?

- Datenstrukturen für mehrere Typen

```
std::array<double, 10> ad;
std::vector<int> vi;
std::vector<Point> vp;
...
```

- Algorithmen für mehrere Typen

```
...
// reverse order of entries
void reverse_array(std::array<double, 10>& a)
{ std::reverse(a); }

void reverse_vector_d(std::vector<double>& v)
{ std::reverse(v); }

void reverse_vector_Point(std::vector<Point>& v)
{ std::reverse(v); }
```

Worum geht es?

- Datenstrukturen für mehrere Typen

```
std::array<double, 10> ad;
std::vector<int> vi;
std::vector<Point> vp;
...
```

- Algorithmen für mehrere Typen

```
...
// reverse order of entries
void reverse_array(std::array<double, 10>& a)
{ std::reverse(a); }

void reverse_vector_d(std::vector<double>& v)
{ std::reverse(v); }

void reverse_vector_Point(std::vector<Point>& v)
{ std::reverse(v); }
```

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

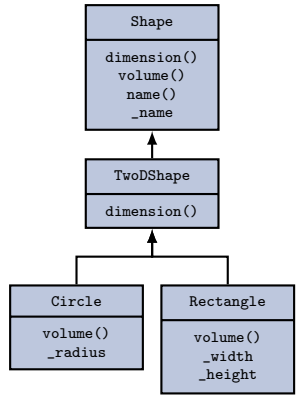
Konzepte

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.



Templates: Motivation

Beobachtung

Oft identischer Code für unterschiedliche Typen: (unter Nutzung des ternären Operators)

```
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: (unter Nutzung des ternären Operators)

```
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(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 T1, typename T2, int size, ...>
class MyTemplate
{
  // Parameters work like normal types and constants
  // within template
  std::array<T1,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

- Funktionen überladen:

```
int max(int, int);
double max(double, double);
```

- Der Compiler wählt die **speziellste** Funktion mit gleichem Namen:
  1. exakt gleiche Typen
  2. integer oder float promotion (`char -> int`, `float -> double`)
  3. implizite Konvertierungen (`char -> short`, Derived → Base)
  4. Nutzerdefinierte Konvertierung (Konstruktur mit einem Argument)

Funktionstemplates

- Syntax:

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

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

```
myFunction(v1,v2);
```

- Wichtig: Compiler darf nie mehr als ein gültiges Template finden!
- `using namespace std;` gefährlich wegen vieler enthaltener Templates.
- Erst beim Aufruf werden die zugehörigen Typen eingesetzt und der zugehörige Code generiert
- Wenn Argumente NICHT erkannt werden können, müssen sie in spitzen Klammern angegeben werden, z.B. `std::get<0>(p);`

Funktionstemplates

- Man kann Klassen- und Funktionstemplates kombinieren

```
template<typename T1, typename T2, int size>
struct MyTemplate
{
  std::array<T1, size> _var1;

  void foo(const T2& );

  template<typename P>
  void foo(const P& w);
};
```

Funktionstemplates

- Man kann Klassen- und Funktionstemplates kombinieren
- Man kann auch mehr Struktur vor schreiben:

```
template<typename T1>
void apply(std::vector<T1>& v)
{
  for(auto& d : v) {
    std::cout << " " d << std::endl;
  }
}
```

Fragen?

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 und Funktionen.
  - 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 und Funktionen.
  - 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
- Instantiierung eines Template:
- 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**
- Empfehlung:
  - Deklaration in .h
  - template-Implementierung in .hh
  - inkludieren der .hh am Ende in der .h

Template-Spezialisierung

- Es ist möglich Templates für bestimmte Template-Parameter Kombinationen zu überschreiben und somit zu spezialisieren.
- Beispiel von https://en.cppreference.com/w/cpp/language/template_specialization:

```
template<typename T>   // primary template
struct is_void : std::false_type {};
template<>  // explicit specialization for T = void
struct is_void<void> : std::true_type {};
int main() {
        // for any type T other than void, the
        // class is derived from false_type
        std::cout << is_void<char>::value << '\n';
        // but when T is void, the class is derived
        // from true_type
        std::cout << is_void<void>::value << '\n';
}
```

- Auch für Funktionen möglich, aber schwieriger nachzuvollziehen

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

```
void scale(double& d, double s)
{ d *= s; }

void scale(Point& p, double s)
{ p.scale(s); }

template<typename T>
void scale(T& v, double s)
{
  for(auto& d : v)
    scale(d, s);
}

int main(int argc, char** argv) {
  std::array<double, 4> a = { 1,4,5,6};
  std::vector<Point> v = {{1.0, 2.0}, {4.0, 6.0}};
  scale(a, 2.0);
  scale(v, 2.0);
}
```

Code Reuse - Zusammenfassung

- Das strenge Typ-System von C++ benötigt in vielen fällen spezialisierte Implementierungen
- C++ Templates ermöglichen das schreiben von Code für mehrere Daten-Typen
- Objektorientiertes Programmieren mit Komposition und Vererbung macht den Code Modular
- Die Menge Code kann durch diese Techniken signifikant verringert werden
  - Der Programmierer muss sich nicht unnötig wiederholen (Don't repeat yourself!)
  - Änderungen im Code betreffen weniger Stellen im Code
- Typedefs, Aliases (`using ...`) und auto-Keyword verringern Schreibarbeit und verbessern Lesbarkeit von Code

Iteratoren — Motivation

Ihr folgenden Code kennt ihr schon:

```
std::vector<int> v(20);
...
for (auto& i : v)
  std::cout << i << std::endl;
```

Doch was passiert eigentlich Hinter den Kulissen?

Iteratoren — Motivation

Der Code kann ungefähr in folgendes übersetzt werden:

```
std::vector<int> v(20);
...
for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
  auto& i = *it;
  std::cout << i << std::endl;
}
```

- Was passiert hier eigentlich?
- Was ist dieser `iterator`?
- Was macht `*it`?
- Was soll das Ganze?

Exkursion: Pointer

- Erinnerung: Jede Variable liegt an einer Adresse im Speicher
- Zugriff auf die Adresse mit Adressoperator &:

```
int i = 0;
std::cout << &i << std::endl;
```

- Variablen, die die Adresse einer anderen Variable speichern, heißen Pointer
- Pointer zeigen immer auf Variablen eines bestimmten Typs. Der Typ einer Pointervariablen ist der Typ der Zielvariablen mit angehängtem *:

```
int i = 0;
int* p = &i; // p is a pointer to int
```

- Pointer, die auf keine gültige Variable verweisen, sollte immer der spezielle Wert `nullptr` zugewiesen werden:

```
int* p = nullptr;
```

Arbeiten mit Pointern

- Pointern können neue Zielvariablen zugewiesen werden:

```
int i = 0, j = 2;
int* p = nullptr; // p does not point anywhere valid
p = &i;           // p now points to i
p = &j;           // p now points to j
```

- Um auf den Wert der Zielvariablen zuzugreifen, dereferenziert man den Pointer, indem man $*$ voranstellt:

```
std::cout << *p << std::endl; // prints 2
```

- Wenn die Zielvariable eine Klasse ist, kann man mit `->` (statt `.`) direkt auf Member der Zielvariablen zugreifen:

```
Point p{1.0,2.0};
Point* pp = &p;
std::cout << pp->x() << std::endl; // prints 2.0
```

Pointer als Handle für Speicher

- `array` und `vector` legen ihre Daten in einen zusammenhängenden Speicherbereich
- Pointer auf Speicherbereich mit Memberfunktion `data()`
- Pointer unterstützen mathematische Operationen:

```
std::vector<int> v(20);
int* data = v.data();
std::cout << *data << std::endl; // prints first entry
++data; // increase pointer by sizeof(int), now points to v[1]
data += 10; // now points to v[11]
std::cout << (data - v.data()) << std::endl; // prints 11
data -= 11; // points to v[0] again
```

- Vektor mit Pointern ausgeben:

```
int* end = v.data() + v.size(); // first invalid address
for(int* p = v.data() ; p != end ; ++p)
  std::cout << *p << std::endl;
```

Iteratoren: Verallgemeinerte Pointer

Warum sind Iteratoren nützlich?

- Nicht alle C++-Container speichern Einträge in zusammenhängendem Speicher (`list`, `map`, `deque`, ...)
- Nicht alle C++-Container erlauben Elementzugriff mit eckigen Klammern
- Wie allgemeine Algorithmen schreiben, die über Container-Elemente iterieren?

Iteratoren: Verallgemeinerte Pointer

Warum sind Iteratoren nützlich?

- Nicht alle C++-Container speichern Einträge in zusammenhängendem Speicher (`list`, `map`, `deque`, ...)
- Nicht alle C++-Container erlauben Elementzugriff mit eckigen Klammern
- Wie allgemeine Algorithmen schreiben, die über Container-Elemente iterieren?

Container stellen Iteratoren zur Verfügungen

- Iteratoren verhalten sich wie Pointer
- Typ des Iterators über geschachtelten Typ `Container::iterator` bzw. `Container::const_iterator` (erlaubt nur lesenden Zugriff auf Elemente)
- Iterator für erstes Element mit `begin()`
- Iterator hinter letztes Element mit `end()`
- Manche Container erlauben Rückwärtsdurchlauf mit `rbegin()`, `rend()`

Iteratoren: Beispiel

Ausgeben von Containern:

```
template<typename T>
void print(const T& t) {
  typename T::const_iterator end = t.end();
  for (auto it = t.begin() ; it != end ; ++it)
    std::cout << *it << std::endl;
}

std::vector<int> v(20);
print(v);

std::list<int> l;
...
print(l);
```

Iteratoren: Kategorien

- Je nach unterliegendem Container unterstützen Iteratoren nicht alle Pointer-Operationen
- Iterator-Kategorien:

`InputIterator`	Lesen von `*it` und `++it`
`OutputIterator`	Schreiben von `*it` und `++it`
`ForwardIterator`	Vollzugriff auf `*it` sowie `++it`
`BidirectionalIterator`	zusätzlich `--it`
`RandomAccessIterator`	zusätzlich `it += n`, `it -= n`

- Jeder Container gibt an, was für eine Iteratorkategorie er hat

Algorithmen

- Alle Algorithmen in der Standardbibliothek arbeiten mit Iteratoren
- Manche Algorithmen haben Anforderungen an die Kategorie (z.B. `std::sort()`)
- Erlauben oft sehr klares Aufschreiben der Intention:

```
std::array<int,20> a;
...
// Replace all occurences of 3 with 7
std::replace(a.begin(),a.end(),3,7);

// Count number of entries with value 7
std::cout << std::count(a.begin(),a.end(),7);

std::vector<int> v;

// Copy array to vector, no need to resize
std::copy(a.begin(),a.end(),std::back_inserter(v));
```

- Erfordern Umgewöhnung und Kenntnis der Möglichkeiten

Zusammenfassung

- Iteratoren ermöglichen es unabhängig vom Containertyp über dessen Inhalt zu iterieren.
- Zugriff auf die Daten des Iterators erfolgt mit `*iterator`.
- Anfangs und (nach-dem-)Ende-Funktion: `begin(cont)` und `end(cont)`
- Ein Iterator braucht in der Regel mindestens einen Operator um das nächste Element zu bekommen und einen Vergleichsoperator (typischerweise `++it` und `!=`).
- Je nach Iteratortyp kann vorwärts und rückwärts iteriert werden oder sogar auf ein beliebiges benachbartes Element zugegriffen werden.
- Viele Algorithmen arbeiten mit Iteratoren, damit diese Unabhängig von einem Containertyp implementiert werden können.