Objektorientiert Programmieren III
Spezialitäten

Stephan Neuhaus

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken







Letzte Vorlesung

- Beispiel von Datentypen: Verkettete Liste
- Vererbung (Shapes)
- Sichtbarkeit: `public` und `private` inheritance
- Virtuelle (abstrakte) Methoden
- Vererbung in Java







Klassenmethoden, Klassenattribute I

Manche Attribute oder Methoden betreffen keine Instanzen, sondern die Klasse als Ganzes:

- Zähler, wie oft eine Klasse instanziiert wurde
- Klassenglobale Defaulteinstellungen
- Logger

Solche Attribute oder Methoden heißen Klassenattribute oder -methoden und werden (in Java und C++) mit `static` bezeichnet







Klassenmethoden, -attribute II

```
#include <iostream>

class MyClass {
public:
  MyClass() { constructCounter++; }
  ~MyClass() { destructCounter++; }
  static int constructInfo() { return constructCounter; }
  static int destructInfo() { return destructCounter; }
private:
  static int constructCounter;
  static int destructCounter;
};
int MyClass::constructCounter = 0;
int MyClass::destructCounter = 0;
int main(int argc, const char *argv[]) {
  MyClass a, b; { MyClass c[10]; }
  if (MyClass::destructInfo() != MyClass::constructInfo()) {
    std::cerr << "constructed: " << MyClass::constructInfo()
              << ", deleted: " << MyClass::destructInfo() << std::endl;
  }
}
```







Klassenmethoden, -attribute III

```
% g++ -Wall -O -o Static1 Static1.cc
% ./Static1
constructed: 12, deleted: 10
%
```

Warum eigentlich -O im Zusammenhang mit -Wall?

Weil einige Warnungen nur durch Datenflußanalyse erzeugt werden können und die Datenflußanalyse nur bei eingeschalteter Optimierung abläuft







Exceptions I

Maßnahme für die Behandlung von Fällen, die den normalen Programmablauf unterbrechen

Beispiel: `open()` zum Öffnen einer Datei

Fall 1: Alles läuft wie geplant, die Datei kann erfolgreich geöffnet werden. Rückgabewert `ret` ≥ 0

Fall 2: Ein Fehler tritt auf. In dem Fall ist `ret` < 0 und das globale Symbol `errno` enthält einen Fehlercode.

Problem: Rückgabewert wird für zwei verschiedene Sachen genutzt: Ergebnis (file descriptor) und Fehlersignal

Das geht nur, wenn beide Wertebereiche disjunkt sind: Ergebnis von `tan()`, bei Argument, das bis auf Maschinengenauigkeit $\pi/2$ entspricht?







Exceptions II

Fehlerbehandlung umständlich (und wird deshalb selten vollständig gemacht) (Achtung: Bug!)

```
int fd = open("/etc/passwd", O_RDONLY);
if (fd >= 0) {
  char *buf = new char[1024];
  if (buf != 0) {
    ssize_t nbytes = read(fd, buf, 1024);
    if (nbytes > 0) {
      if (close(fd) >= 0) {
        delete[] buf; // <-- Bug!
      } else
        error("can't close /etc/passwd");
    } else
      error("can't read from /etc/passwd");
  } else
    error("can't allocate buffer");
} else
  error("can't open /etc/passwd");
```

Exceptions III

Normaler Programmfluß und Ausnahmebehandlung am Ende

```
char* buf = 0;
try {
  File f = File::open("/etc/passwd", O_RDONLY);
  buf = Buffer::allocBuffer(1024); // throws AllocationError
  ssize_t nbytes = f.read(buf, 1024);
  f.close();
} catch (FileNotFoundError e) {
  error("can't find file %s: %s", e.getFileName(), e.getMessage());
} catch (AllocationError e) {
  error("can't allocate %lu bytes", e.getSize());
} catch (ReadError e) {
  error("can't read %lu bytes: %s", e.getSize(), e.getMessage());
} catch (CloseError e) {
  error("can't close file %s: %s", e.getFileName(), e.getMessage());
}
if (buf != 0) delete[] buf;
```







Exceptions IV: Nesting

```
class XXII { /* ... */ };
void f() {
  try {
  } catch (XXII) {
    try {
      // Something complicated
    } catch (XXII) {
      // Complicated handler failed
    }
  }
}
```







Exceptions V

Ausnahmebehandlung in Java verbindlich: Programm übersetzt nicht, wenn Ausnahmen nicht behandelt werden

Wenn Ausnahmen geworfen werden, muß das in Java deklariert werden

Ausnahmebehandlung in C++ optional (im Sinn von: Kompiliert auch ohne Ausnahmebehandlung)







Templates I

Hier ein Stack für `int`-Werte:

```
class IntStack {
public:
  static const int stackSize = 100;
  IntStack() { t = 0; mem = new int[stackSize]; }
  ~IntStack() { delete[] mem; }
  void push(int item) { if (top < stackSize) mem[t++] = item; }
  int pop() { if (t > 0) return mem[--t]; else return -1; }
  int top() { if (t > 0) return mem[t-1]; else return -1; }
private:
  int t;
  int* mem;
};
```







Templates I

Hier ein Stack für `float`-Werte:

```
class FloatStack {
public:
  static const int stackSize = 100;
  FloatStack() { t = 0; mem = new float[stackSize]; }
  ~FloatStack() { delete[] mem; }
  void push(float item) { if (top < stackSize) mem[t++] = item; }
  float pop() { if (t > 0) return mem[--t]; else return -1; }
  float top() { if (t > 0) return mem[t-1]; else return -1; }
private:
  int t;
  float* mem;
};
```







Generischer Stack

```
class StackEmpty {};
class StackFull {};

template<class C> class Stack {
public:
  static const int stackSize = 100;
  Stack() { t = 0; mem = new C[stackSize]; }
  ~Stack() { delete[] mem; }
  void push(const C& item) {
    if (t < stackSize) mem[t++] = item; else throw StackFull();
  }
  C &pop() {
    if (t > 0) return mem[--t]; else throw StackEmpty();
  }
  const C& top() const {
    if (t > 0) return mem[t-1]; else throw StackEmpty();
  }
private:
  int t;
  C* mem;
};
```







„Generischer“ Stack, Java

```
public class Stack {
  public static final int stackSize = 100;
  public Stack() { t = 0; mem = new Object[stackSize]; }
  public void push(Object item) throws StackFullException {
    if (t < stackSize)
      mem[t++] = item;
    else
      throw new StackFullException();
  }
  public Object pop() throws StackEmptyException {
    if (t > 0) return mem[--t]; else throw new StackEmptyException();
  }
  public Object top() throws StackEmptyException {
    if (t > 0) return mem[t-1]; else throw new StackEmptyException();
  }

  private int t;
  private Object[] mem;
};
```







Templates in C++

- “C++ templates are the first language feature to require more intelligence from the environment than one usually finds on a UNIX system.” (GCC 3.3 Manual)
- Templates haben lokale Sichtbarkeit, d.h., sie müssen in jeder Translation Unit neu definiert werden. Das kann zu aufgeblähtem Code führen, wenn in vielen Dateien gleiche Templates verwendet werden. Abhilfe: Template Repositories (CFront), Common Blocks (Borland)







Template-Instanziierung

- Common Blocks: Templates werden in jedes `.o`-File hineinkompiliert. Der Linker sortiert Duplikate aus. Vorteil: Keine externe Komplexität. Nachteil: Kompilierzeit steigt, jedes Template wird mehrfach (redundant) übersetzt.
- Template Repository: Ein Ort, an dem Templates untergebracht werden. Management automatisch. Templates werden erst im Repository gesucht und erst instanziiert, wenn sie dort nicht gefunden werden. Vorteil: Bessere Übersetzungsgeschwindigkeit. Nachteil: Hoher Aufwand zur Wartung des Repositories.
- GCC unterstützt das Borland-Modell automatisch auf GNU/Linux, ansonsten gibt es Optionen







Vor- und Nachteile Java/C++

- Templates bieten dem Compiler die Möglichkeit zu Optimierungen, die der Java-Compiler nicht hat
- Templates sind typsicher. In Java muß man ein Objekt, das man aus einer generischen Klasse rausholt, erst durch Typumwandlung in den richtigen Typ verwandeln (nicht zur Kompilierzeit prüfbar)
- Problematisches Template-Management (s.o.)
- In Java “generische” Typen nur für Ableger von `java.lang.Object`, in C++ auch für `int`
- Komplizierte Syntax in einer syntaktisch eh schon komplizierten Sprache

Algorithmen und Funktionsobjekte

Die Standard-C++-Bibliothek bietet eine Menge Funktionen an, um mit Collections (also `list`, `vector`, `set`, `map` etc) Sinnvolles anzustellen

Hier nur einfache Beispiele möglich

```
#include <algorithm>
#include <string>
using namespace std;

void f(const list<string>& ls) {
  list<string>::const_iterator p = find(ls.begin(), ls.end(), "Fred");

  if (p == ls.end()) { // "Fred" not found
    // ...
  } else {
    // p points to "Fred"
  }
}
```

Nichtmodifizierende Operationen

`for_each()`	Operation für alle Elemente
`find()`	Element suchen
`find_if()`	Ersten Treffer für Prädikat suchen
`find_first_of()`	Wert aus einer Sequenz in anderer suchen
`adjacent_find()`	Nebeneinanderliegende Werte finden
`count()`	Elemente zählen
`count_if()`	Elemente zählen, die Prädikat erfüllen
`mismatch()`	Erster Unterschied zweier Sequenzen
`equal()`	`true`, wenn zwei Sequenzen gleich sind
`search()`	Erstes Vorkommen als Subsequenz
`find_end()`	Letztes Vorkommen als Subsequenz
`search_n()`	n-tes Vorkommen finden

Funktionsobjekte: Beispiele I

```
void f(vector<int>& vi, list<int>& li) {
  typedef list<int>::iterator LI;
  typedef vector<int>::iterator VI;
  pair<VI,LI> p
    = mismatch(vi.begin(), vi.end(), li.begin(), less<int>());

  // *p.first now points to the first element in li that is not less
  // than its corresponding element in vi
}
```







Funktionsobjekte: Beispiele II

```
class Person { /* ... */ };

struct Club {
  string name;
  list<Person*> members;
  list<Person*> officers;
  // ...
  Club(const string& name);
};
```

Wir wollen eine `list<Club>` nach einem `Club` mit einem bestimmten Namen durchsuchen

Der normale ==-Operator ist da keine große Hilfe, weil wir Clubs nicht als Ganzes vergleichen wollen, sondern nur die Namen

Also schreiben wir uns ein eigenes Prädikat







Funktionsobjekte: Beispiele III

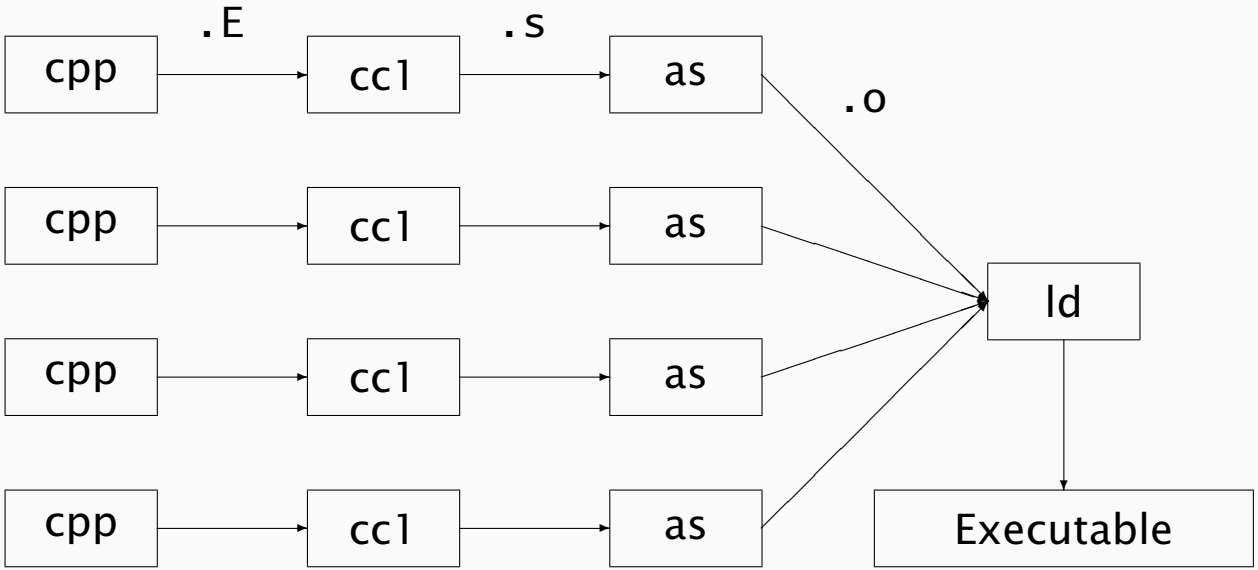
```
class ClubEqual : public unary_function<Club, bool> {
  string clubName;
public:
  explicit ClubEqual(const string& myClubName)
    : clubName(myClubName) {}

  bool operator() (const Club& club) { return club.name == s; }
};


void findClubWithName(list<Club>& lc) {
  list<Club>::iterator p
    = find(lc.begin(), lc.end(), ClubEqual("Dining Philosophers"))
  if (p == lc.end()) { // Club not found
    // ...
  } else {             // Use *p
    // ...
  }
}
```







Der C++-Präprozessor



Deklarationen werden in Header-Dateien gesammelt.
Präprozessor sorgt mit `#include` dafür, daß diese Deklarationen auch zur Verfügung stehen

#include I

Die #include-Anweisung gibt es in zwei Varianten:

- #include <iostream> fügt den Inhalt des Headers <iostream> an der angegebenen Stelle ein.
- Findet sich an einem implementationsabhängigen Ort
- Muß noch nicht einmal eine Datei sein (z.B. als vorkompilierter Header zur schnelleren Übersetzung oder gleich in Compiler eingebaut)







#include //

- `#include "MyClass.h"` fügt den Inhalt der Datei `MyClass.h` ein
- Wird in einer Reihe von Verzeichnissen gesucht
- Wird typischerweise auch im aktuellen Verzeichnis gesucht
- Verzeichnisse und deren Suchreihenfolge können i.d.R. durch Optionen beeinflußt werden (GCC: “`-I/usr/X11/include`”, “`-I../src`”)







Makros I

Makros sind einer der dunklen Kapitel von C++

Kommen aus einer Zeit, als es in C noch keine `const`-Deklarationen oder `inline`-Funktionen gab

Java hat (zum Glück!) keinen Präprozessor!

(Warum, sehen Sie gleich)

Makros geben einem Codestück einen Namen, damit er unter diesem Namen eingesetzt werden kann







Makros II

```
#define MAX 3

void f() {
  int i[MAX] = { 1, 2, 3 }; // Means int i[3] after preprocessor is
                            // done with it
}
```

Besser:

```
static const int max = 3;

void f() {
  int i[max] = { 1, 2, 3 }; // Includes type checking
}
```







Makros: Fehler I

```
% cat /tmp/m1.cc
#define MAX2 6;  // <- oh oh, semicolon probably a mistake

void f() {
  int j[MAX2] = { 1, 2, 3, 4, 5, 6 };
}
% gcc -c /tmp/m1.cc
/tmp/m1.cc: In function ‘void f()’:
/tmp/m1.cc:4: parse error before ‘;’
%
```

Aber der Teil mit dem Semikolon in Zeile 4 ist doch vollkommen OK. . .







Makros: Fehler II

```
#define USES_CACHE false
#define USES_ASM true

#define USES_OPTIMIZATION USES_CACHE || USES_ASM

bool is_optimized(bool condition) {
  // return condition && false  || true  ;
  return condition && USES_OPTIMIZATION;
}
```

Klammern hilft hier:

```
#define USES_CACHE false
#define USES_ASM true

#define USES_OPTIMIZATION (USES_CACHE || USES_ASM)

bool is_optimized(bool condition) {
  // return condition && (false  || true)  ;
  return condition && USES_OPTIMIZATION;
}
```

Makros mit Argumenten I

```
#define MAX1(a,b)  (a > b ? a : b)

void f(int *a, int& i, int& j) {
  // means ( a[i] + 1  > a[j] + 1] ? a[i] + 1 : a[j] + 1] ) ;
  int m = MAX1(a[i] + 1, a[j] + 1]);   // oh, oh...
}
```

Besser:

```
#define MAX2(a,b)  ((a) > (b) ? (a) : (b))

void f(int *a, int& i, int& j) {
  // means ((a[i] + 1) > (a[j] + 1]) ? (a[i] + 1) : (a[j] + 1])) ;
  int m = MAX2(a[i] + 1, a[j] + 1]);   // ok now
}
```

Makros mit Argumenten II

```
#define MAX2(a,b)  ((a) > (b) ? (a) : (b))

void f(int& i, int& j) {
  // means ((i++) > (j++) ? (i++) : (j++)) ;
  int m = MAX2(i++,j++);   // oh, oh...
}
```

Eins der beiden Argumente wird *zweimal* ausgewertet, also auch zweimal inkrementiert. Besser:

```
template<class T> inline const T& max(const T& a, const T& b) {
  return (a < b) ? a : b;
}

void f(int *a, int& i, int& j) {
  int m = max(i++, j++);     // ok now
}
```







Warum das alles?

- Der Präprozessor ist (leider) ein fester Bestandteil von C++, daher sollten Sie ihn kennen
- Sie müssen Code von anderen Leuten lesen, die nicht auf die Benutzung des Präprozessors verzichten
- Manchmal kann der Präprozessor schon nützlich sein







Bedingte Übersetzung I

Manchmal ist es in C++ ein Fehler, wenn man ein Symbol mehrfach deklariert oder definiert

Man bräuchte also eine Möglichkeit, die dafür sorgt, daß ein Header nicht erneut eingefügt wird, wenn er schonmal eingefügt wurde

Dafür gibt es im Präprozessor die *bedingte Übersetzung*







Bedingte Übersetzung II

```
#ifdef MAX
#  define MAX2 (2*MAX)
#else
#  define MAX2 0
#endif
```

Je nachdem, ob das Präprozessor-Symbol MAX definiert war oder nicht, bekommt MAX2 den Wert (2*MAX), bzw. 0

Äquivalent:

```
#ifndef MAX
#  define MAX2 0
#else
#  define MAX2 (2*MAX)
#endif
```







Bedingte Übersetzung III

Der `ifdef`-Wrapper um einen Header schützt vor mehrfachem Einfügen

```
// File MyClass.h
#ifndef _MyClass_h_
#  define _MyClass_h_

class MyClass {
  // ...
};

#endif // _MyClass_h_
```

Bei erneutem Einfügen durch `#include "MyClass.h"` ist das Symbol `_MyClass_h_` bereits definiert und `MyClass` wird nicht erneut deklariert







Benamung I

So nicht:

```
void MrFlo123::grk151a(a_struct *x) {
  s += x->x;
}
```

Eher so:

```
void CheckingAccount::addToBalance(const MonetaryValue& amount) {
  // Assume this->currency == amount.getCurrency();
  balance += amount.getAmount();
}
```







Benamung II

- Klassen sollten Substantive als Namen haben: `CheckingAccount`, `GiroKonto`, `XmlElement`, `AutoKarosse` usw., aber nicht `GelenkDrehen` (besser `GelenkDreher`, noch besser als `drehe()`-Methode von `Gelenk`), `FunktionAbleiten` (besser `Ableitung`)
- Methoden sollten in der Regel Verben in der Befehlsform als Namen haben: `berechneAbleitung()`, `drehe()`, `pruefeZulaessigkeit()` usw., aber nicht `gelenkDrehen()`, `testenObZulaessig()` etc.
- Boolesche Methoden können auch Prädikatsnamen tragen: `zulaessig()`, `kleiner()` (das noch besser als `operator<`), usw.







Benamung III

Je weiter außen (also je globaler) ein Name ist, desto länger sollte er sein

```
class SignalHandler {   // Full-length name: file scope
  int signal;           // Longer name: class scope
public:
  // Long name for method
  int installNewSignal(int& newSignal) {
    int t = signal;     // Short name OK: short scope
    signal = newSignal;
    newSignal = t;

    return t;
  }
};
```







Benamung IV

Bekannte technische Konstanten können natürlich mit ihren bekannten Namen definiert werden. Dann aber bitte in einer Klasse oder in einem geeigneten Namespace.

```
// Characteristic impedance of vacuum, in Ohm
const double PhysicalConstants::Z_0 = 376.730313461;

// Newtonian constant of gravitation, in m^3 kg^-1 s^-2
const double PhysicalConstants::G = 6.673e-11;

// Speed of light in vacuum, in m s^-1
const double PhysicalConstants::c_0 = 299792458.0;

// Mass of electron (at rest), in kg
const double PhysicalConstants::m_e =  9.10938188e-31;

// Number of printer's points in an inch, in in^-1
const double TypographicalConstants::pointsPerInch = 72.27;
```

Magic Numbers I

So nicht:

```
class Board {
  int board[8][8];

  void initialize() {
    for (int i = 0; i <= 7; i++)
      for (int j = 0; j <= 7; j++)
        board[i][j] = 0;
  }
};
```

Bedeutet die zweite 7 dasselbe wie die erste 7? Sind beide vielleicht eigentlich 8 - 1?

Bei der Verwendung von Magic Numbers sind Instanzen „derselben“ Zahl, sowie von verwandten Zahlen (x + 1, x - 1, 2*x usw) nur schwer auszumachen







Magic Numbers II

Besser so:

```
class Board {
  enum Piece {
    free,
    whiteKing, whiteQueen, whiteRook, whiteBishop, whiteKnight, whitePawn,
    blackKing, blackQueen, blackRook, blackBishop, blackKnight, blackPawn,
  };
  static const int boardSize = 8;

  Piece board[boardSize][boardSize];

  static map<Piece,int> value;  // Value of a piece, in pawns

  void initialize() {
    for (int i = 0; i < boardSize; i++)
      for (int j = 0; j < boardSize; j++)
        board[i][j] = free;
  }
  // ...
```







Magic Numbers III

```
  // ...
  static void boardInitialize() {
    value[free] = 0;
    value[whiteKing] = value[blackKing] = limits<int>::max();
    value[whiteQueen] = value[blackQueen] = 9;
    value[whiteRook] = value[blackRook] = 5;
    value[whiteBishop] = value[blackBishop] = 3;
    value[whiteKnight] = value[blackKnight] = 3;
    value[whitePawn] = value[blackPawn] = 1;
  }
};
```







Magic Numbers IV

Oder so:

```
class Board {
  static const int horizontalSize = 8;
  static const int verticalSize = 9;

  int board[horizontalSize][verticalSize];

  void initialize() {
    for (int i = 0; i < horizontalSize; i++)
      for (int j = 0; j < verticalSize; j++)
        board[i][j] = 0;
  }
};
```







Benamung: Abschluß

- Verschmutzen Sie den globalen Namensraum nicht unnötig
- Kapseln sie alle sichtbaren Namen in Namespaces oder in Klassen
- Geben Sie Namen nur die Sichtbarkeit, die sie unbedingt benötigen
- Definieren Sie Namen so lokal wie möglich
- Vermeiden Sie (in der Regel) “magic numbers”