Praxis der Programmierung

Objektorientierte Programmierung mit C++

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Das Paradigma der objektorientierten Programmierung

Grundidee des Paradigmas

- *bisher:* Wie soll ein Ziel erreicht werden?
  - Algorithmen/Prozeduren im Mittelpunkt
  - Daten werden durch Prozeduren manipuliert

  $\rightsquigarrow$ prozedurale Programmierung / imperative Programmierung

- *jetzt:* Was soll erreicht werden?
  - Daten werden zu Typen zusammengefasst (siehe `struct`-Typen)
  - Durch Zuweisung von Werten (konkreten Daten) entstehen Datenobjekte.
  - Algorithmen durch Aktivitäten / Interaktionen von Datenobjekten

Objekte

- repräsentieren Objektes der realen Welt in der Terminologie objektorientierter Programmiersprachen
- besitzen einen **Namen**, mit dem sie angesprochen werden können
- besitzen **Attribute** (Eigenschaften), deren Werte im Allgemeinen veränderlich sind und den **Zustand** der Objekte bestimmen
- reagieren auf an sie gesendete **Botschaften** durch gewisse Aktionen

Klassen

- Gesamtheiten (Mengen) von Objekten
  - mit denselben Attributen,
  - die dieselben Botschaften verstehen und auf dieselbe Weise darauf reagieren,
  - unterscheiden sich in den Werten ihrer Attribute
  - Verhalten eines Objekts kann von seinem Zustand abhängen

- Objekte sind *Exemplare* (*Instanzen*) einer Klasse

  - alle Attribute besitzen einen Wert (Objektzustand)
  - repräsentiert durch eine Variable vom Typ der Klasse,
    z.B. für Klasse `Cls` mit Exemplar `obj`:

```
Cls obj;
```

Klassen *versus* Stukturen

```
typedef struct datum {
   int day;
   int month;
   int year;
} Date;
```

```
class Date {
public:
   int day;
   int month;
   int year;
};
```

- definieren den gleichen Datentyp
- Membervariablen in Strukturen sind Attribute in Klassen (Datenelemente)
- Schlüsselwort `public` erlaubt Zugriffe auf Attribute wie auf Membervariablen bei Strukturen
- direkte Initialisierung erst ab **C**++11-Standard möglich

Klassen und Zugriffe auf Datenelemente

```
class Date {
public:
   int day;
   int month;
   int year;
};
```

```
int main() {
   Date heute;
   heute.day = 4;
   heute.month = 6;
   heute.year = 2014;
}
```

Verhalten von Objekten: Methoden

- Definition von Funktionen in der Klassendef. (innerhalb von `class {...}`)
- Signaturen der Methoden definieren, welche "Botschaften" von Objekten dieser Klasse verstanden werden
- Implementierungen definieren Verhalten/Reaktion der Objekte

```
class Date {
public:
   int day, month, year;

   void setNewYear(int y) {
      day = 1;
      month = 1;
      year = y;
   }
};
```

```
int main() {
   Date d;
   d.setNewYear(2014);
   Date * dptr = &d;
   dptr->setNewYear(2014);

   // beides liefert 1.1.2014;
}
```

Auflösung von Verdeckungen

- lokale Variblen (auch Parameter) verdecken gleichnamige Datenelemente
- Nutzen der vordefinierten Selbstreferenz `this` (ist Pointer!)
  $\rightsquigarrow$ es soll explizit ein Element des Objekts angesprochen werden

```
class Date {
public:
   int day, month, year;

   void setNewYear(int year) {
      day = 1;
      month = 1;
      this->year = year;
   }
};
```

```
int main() {
   Date d;
   d.setNewYear(2014);
   Date * dptr = &d;
   dptr->setNewYear(2014);

   // beides liefert 1.1.2014;
}
```

Konzepte des objektorientierten Paradigmas

- Klasse und Objekt
- Datenelemente/Instanzvariablen (Attribute) und Methoden
- Kapselung
  Interna von Objekten sind nach außen unsichtbar und können von außen nicht manipuliert werden
- Vererbung
  Weitergabe von Merkmalen und Fähigkeiten (Datenelementen und Methoden)
  $\longrightarrow$ hierarchisches Klassensystem ($Ober$- und $Unterklassen$)
- Polymorphismus
  verschiedene Reaktionen von Instanzen verschiedener Unterklassen auf eine gemeinsam verstandene Botschaft
  $\longrightarrow$ Überschreiben von Methoden

Erzeugen, Verwenden und Zerstören von Objekten

Initialisieren von Objekten

```
class Date {
public:
   int day;
   int month;
   int year;
};
```

```
int main() {
   Date heute;
   heute.day = 4;
   heute.month = 6;
   heute.year = 2014;
}
```

unsauber, da Initialisieren aller Datenelemente nicht erzwungen ist:

```
// weiter in main:
                                   Date morgen;
                                   morgen.day = heute.day + 1;
                                   morgen.month = heute.month;

// Ausgabe aller Attributwerte von morgen ergibt etwas wie  5.6.4197168
```

Konstruktoren

- Definition einer speziellen Methode zum Erzeugen eines Objekts (Konstruktor)
- Name des Konstruktors ist immer Name der Klasse
- kein Rückgabetyp (*auch nicht* `void`)

```
class Date {
public:
   int day, month, year;

   Date() {   // hier parameterlos
      day =1; month = 1; year = 1;
   }
};
```

- Programmierer muss für Initialisierung aller Datenelemente sorgen!!!

Überladen von Konstruktoren

- Vereinbarung mehrerer Konstruktoren mit unterschiedlichen Parameterlisten
- Standardkonstruktor
  - parameterlos $\rightsquigarrow$ genau einer pro Klasse (`Date()`)
  - Initialisierung aller Datenelemente mit festen Standardwerten
- Initialisierungskonstruktor
  - Initialisierung (einiger) Datenelemente mit Parameterwerten (`Date(int year)`)
  - verschiedene Parameterlisten für verschiedene Initialisierungen
- Compiler erkennt aufgerufene Funktion am Namen und den aktuellen Parametern

Verwenden von Konstruktoren

- Definition einer Objektvariablen ruft Konstruktor auf:

```
Date d;         // Standardkonstruktor Date()
Date d(2014);   // Initialisierungskonstruktor Date(int year)
```

- ist *gar kein* Konstruktor definiert, so wird der *implizite* Konstruktor aufgerufen
- Der implizite Konstruktor ist parameterlos und erzeugt eine Instanz, ohne die Datenelemente explizit zu initialisieren; (z.B., wenn Initialisierung durch Methoden erfolgen soll)
- impliziter Konstruktor kann als Standardkonstruktor angefordert werden:

```
Date() = default;
```

Objekte ohne impliziten Konstruktor anlegen

`Date d;` nur möglich, wenn ein parameterloser Konstruktor zur Verfügung steht

```
class Date {
public:
   int day, month, year;
   Date(int year) {
      day = 1;
      month = 1;
      this->year = year;
   }
};

int main() {
   Date gehtNicht;        // Compilerfehler
   Date gehtDoch(2014);   // Aufruf des definierten Konstruktors
```

Überladen von Konstruktoren bei `Date`

```
class Date {
public:
   int day, month, year;
   Date() {
      day = 1; month = 1; year = 2000;
   } // Standardkonstruktor

   Date(int year) {
      day = 1;
      month = 1;
      this->year = year;
   } // ein Initialisierungskonstruktor

   Date(int day, int month, int year) {
     ... } // noch ein Initialisierungskonstruktor
};
```

Erzeugen von Objekten auf dem Heap

- Anlegen eines Objekts im Speicher mit `new`
- liefert Referenz auf ein vorinitialisiertes Objekt auf dem Heap

```
Date * irgendwann;
irgendwann = new Date;

cout << (*irgendwann).day << "." (*irgendwann).month << ".";
cout << irgendwann->year << endl;
```

- `new` "ersetzt" `malloc()`

Zerstören von Objekten

- bei "manueller" Definition eines Objekts (Variablendefinition):
  Benutzen des Stacks
  $\rightsquigarrow$ Speicherfreigabe bei Verlassen des Blocks mit der Variablendefinition

- bei Anlegen des Objekts mit `new`:
  Benutzen des Heaps
  $\rightsquigarrow$ Speicherfreigabe durch Programmierer mit `delete` erforderlich:

```
delete irgendwann;
irgendwann = 0;          // irgendwann = nullptr;
                         // im C++11-Standard
```

Destruktoren

- werden automatisch bei Zerstören von Objekten aufgerufen
  (bei `delete` bzw. Erreichen des Blockendes, in dem das Objekt definiert wurde)
- impliziter Standarddestruktor, falls kein Destruktor defniert ist
- Definition: `~`*Klassenname*`()`, z.B. `~Date()`
- wichtig zum Freigeben von Ressourcen, die das Objekt angefordert hat
  (z.B. durch Erzeugen von Exemplaren anderer Klassen, die nur von diesem Objekt genutzt wurden)

Destruktoren Beispiel

```
class HighScore {
public:
   int score;
   Date * d;
   HighScore();
   ~HighScore();
};

HighScore::HighScore() {
   score = 0;
   d = new Date;
}
```

```
HighScore::~HighScore() {
   delete d;
   d = 0;
}
```

Übersichtliche Klassendefinitionen

... durch Vorwärtsdeklaration der Konstruktoren und Methoden:

```
class Date {
public:
   int day, month, year;
   Date(int year);
   void setNewYear(int year);
};

Date::Date(int year) {
   this->year = year;
}

void Date::setNewYear(int year) {
   day = 1; month = 1;
   this->year = year;
}
```

```
/* Die Namen muessen
 * qualifiziert werden:
 * Date::name
 *
 * Dadurch werden sie
 * von global definierten
 * Namen (wie Date) unterschieden.
 */
```

Kapselung

Kapselung

- Klassenelemente können vor dem Zugriff von außen geschützt werden
- Schlüsselwort `private` (ist default-Einstellung)
  $\rightsquigarrow$ nur Methoden der Klasse selbst können zugreifen/aufrufen
- Schlüsselwort `public`
  $\rightsquigarrow$ alle Funktionen können zugreifen
- Schlüsselwort `friend` vor fremden Klassen oder Methoden
  $\rightsquigarrow$ diese dürfen auf private Elemente zugreifen

Kapselung bei `Date`

```
class Date {
   int day, month, yaer;   // private als default-Modifikator
public:
   Date();
   ~Date();
   int getDay();                // Getter-Methoden,
   int getMonth();              // damit die Datenelemente
   int getYear();               // gelesen werden koennen

   void setDay(int day);        // Setter-Methoden,
   void setMonth(int month);    // damit die Datenelemente
   void setYear(int year);      // veraendert werden koennen
};
```

Ausnutzung der Kapselung

- Unterscheiden von Lese- und Schreibzugriffen durch gezielte Definition von Gettern und Settern

- Kontrolle über die Art der Zugriffe:

```
int getDay() {
   return day;
}

void Date::setDay(int day) {
   if (0 < day && day < 32)
      this->day = day;
}
```

Gezieltes Öffnen der Kapselung

Erlauben von Zugriffen auf private Datenelemente für eine globale Funktion:

```
class Date {
public: Date();
private: int day, month, year;
friend void setNewYear(Date * d, int year);
};

Date::Date() { day = 1; month = 1; year = 2000; }

void setNewYear(Date * d, int year) { d->year = year; }

int main() {
   Date nextNewYear;  // Aufruf des Standardkonstruktors
   setNewYear(&nextNewYear, 2015);
}
```

Namenskonventionen

- Syntax: Bezeichner wie in **C**
- sprechende Bezeichner (Ausnahme: Schleifenzähler u.ä.)
- Grundsätze für Bezeichner:

–	einfache Datentypen	vollständig klein	`int`
–	Klassen	jedes Teilwortes groß	`Date`
–	Variablen und Methoden	große Anfangsbuchstaben außer beim ersten Teilwort	`ptr` `setNewYear`
–	Konstanten	vollständig groß	`PI`