Softwareentwicklung 4

4. Einheit

Dominik Dolezal,
Michael Borko,
Erhard List,
Hans Brabenetz

Höhere Lehranstalt für Informationstechnologie

20. Oktober 2016

Inhalt

Wiederholung

Wir hatten:

- Dynamische Speicherverwaltung für Arrays

```
int* dyn_array = new int[dyn_size];
for (int i = 0; i < dyn_size ; i++)
  dyn_array[i] = 2 * i;
delete[] dyn_array ;
```

- Dynamische Speicherverwaltung für Klassen in Klassenhierarchien

```
Figur * f1 = new Quadrat (10 , 3, 3);
f1->draw();
delete f1;
```

Wiederholung

Wir hatten:

- Strings

```
cout << "Was lernst du?" << endl;
string antwort;
cin >> antwort;
if (antwort == "C++")
  cout << "Ich auch!";

string fach = "Softwareentwicklung";
string ware = fach.substr(4, 4);
fach.replace(0, 8, "hardware");
fach[0] = toupper(fach[0]);
```

Wiederholung

Wir hatten: Dateien

- Wir kennen schon zwei verschiedene Streams: `cout` als Ausgabe und `cin` als Eingabe
- Streams schreiben oder lesen Sequenzen von Werten (Zeichen)
- Ein solcher Stream muss nicht unbedingt an der Konsole „angeschlossen“ sein – dasselbe Konzept kann auch für Dateien verwendet werden

```
#include <fstream>
// ...
ofstream writefile{ "hallowelt.txt" };
if (writefile) {
  writefile << "Hallo Welt!" << endl;
}
writefile.close();
```

Wiederholung

- Der Status der letzten Operation eines Streams (Öffnen, Lesen) kann praktischerweise als `bool` abgefragt werden

```
ofstream writefile{ "hallowelt.txt" };
int zahl;
while(cin >> zahl)
  writefile << zahl << endl;
writefile.close();
```

Wiederholung

- Das Einlesen einer Datei funktioniert analog

```
ifstream readfile{ "hallowelt.txt" };
if (readfile) {
   string line;
   while (getline(readfile, line))
   {
      cout << line << '\n';
   }
}
readfile.close();
```

- `getline()` wird verwendet, da der >>-Operator Whitespaces ignoriert

Vererbung

- Wie in Java wird Vererbung verwendet, um bestehende Klassen zu erweitern, ohne sie ändern zu müssen
  - Wenn wir beispielsweise unserer Vektor-Klasse ein Attribut für die dritte Dimension hinzufügen wollen, erben wir davon:

```
class Vektor3D : public Vektor
{
public:
  Vektor3D(double, double, double);
  virtual ~Vektor3D();
  double getZ() const { return z; }
protected:
  double z;
};
```

Vererbung

- Vererbung ermöglicht außerdem *Polymorphie*, d.h. unterschiedliches Verhalten bei gleichem Interface
  - Beispielsweise erben `cin` und `ifstream` beide von der Klasse `basic_ifstream`
  - `getline()` akzeptiert beliebige Referenzen vom Typen `basic_ifstream` und man kann somit mit derselben Funktion sowohl Files als auch Benutzereingaben lesen
  - Für neue Eingabemöglichkeit (z.B. Netzwerkeingabe) muss man nur von `basic_ifstream` erben und die jeweiligen Funktionen implementieren – `getline()` wird nicht geändert

```
string line;
if(getline(cin, line))
  //...
ifstream readfile{ "hallowelt.txt" };
if(getline(readfile, line))
  //...
```

Vererbung

Achtung:

- Polymorphes Verhalten benötigt Pointer oder Referenzen

```
Figur* f1 = new Quadrat (10 , 3, 3);
f1->draw();
delete f1;

Figur& f = Quadrat(10, 3, 3);
f.draw();
```

Pointer vs. Referenzen

Pointer	**Referenzen**

`Figur* f1=new Quadrat(1,3,3);`
`f1->draw();`
`delete f1;`
	`Figur& f=Quadrat(1,3,3);`
	`f.draw();`

-> Operator	. Operator
Oft in Verbindung mit `new`, erfordert manchmal ein `delete`	`delete` sehr unüblich
Kann `nullptr` darstellen	Ist immer gültig
Für Pointer-Arithmetik sinnvollere Wahl (z.B. Arrays)	Für Output-Parameter sinnvollere Wahl
Pointervariable änderbar	Konstant

Pointer vs. Referenzen

- Objektvariablen in Java sind Referenzen in C++ sehr ähnlich (aber nicht ident)
- Referenzen wurden in C++ als „sicherere“ Alternative zu Pointern aus C eingeführt
- Funktionsparameter können per Referenz übergeben werden

```
void inc(int& x) { x++; }

int main() {
  int zahl = 0;
  inc(zahl);
  cout << zahl; // "1"
  return EXIT_SUCCESS; }
```

- Ausführliches Tutorial hier abrufbar

Virtuelle Methoden

Achtung:

- Methoden müssen in der Superklasse mit `virtual` gekennzeichnet werden, damit sie in der Subklasse überschrieben werden können
- Sonst wird die Funktion lediglich „versteckt“ (hide)

```
class Figur
{
public:
  virtual void
  printpolytype() const
    { cout << "Figur";};
  void printtype() const
    { cout << "Figur";};
};
```

```
class Quadrat : public Figur
{
public:
  virtual void
  printpolytype() const
    { cout << "Quadrat"; };
  void printtype() const
    { cout << "Quadrat"; };
};
```

Virtuelle Methoden

```
Figur f{};
Quadrat q{};
f.printtype(); // "Figur"
f.printpolytype(); // "Figur"
q.printtype(); // "Quadrat"
q.printpolytype(); // "Quadrat"
Figur& qf = q;
qf.printtype(); // <- "Figur" (!!!)
qf.printpolytype(); // "Quadrat"
```

Virtuelle Methoden

- Durch das Wort `virtual` weiß der Compiler, dass er in Subklassen nach einer überschriebenen Methode suchen muss
- Intern werden sog. *Virtual Function Tables* verwendet, um Methodenaufrufe von virtuellen Funktionen zur Laufzeit aufzulösen
- Dieser Mechanismus ist performant implementiert und benötigt weniger als 25% Overhead
- Auch Java verwendet solche Virtual Function Tables!

Virtuelle Methoden

- Konstruktoren und Destruktoren haben eine spezielle Rolle
- Konstruktoren können nicht virtuell sein, da beim Instanziieren ja immer der konkrete Subtyp verwendet wird
- Destruktoren sollten immer virtuell sein, da bei einem `delete`-Aufruf auf eine Variable vom Typen der Basisklasse ggf. auch in den Subklassen aufgeräumt werden muss
- Objekte werden bottom-up erstellt (Basisklasse $\rightarrow$ Member-Attribute $\rightarrow$ Subklasse)
- Objekte werden top-down zerstört (Subklasse $\rightarrow$ Member-Attribute $\rightarrow$ Basisklasse)

Virtuelle Methoden

Was gibt das Programm aus?

```
#include <iostream>
using namespace std;

class Person {
public:
  Person() { cout << "Konstruktor Person" << endl; }
  ~Person() { cout << "Destruktor Person" << endl; }
};
class Mitarbeiter : public Person {
public:
  Mitarbeiter() { cout << "Konstruktor Mitarbeiter"
      << endl; }
  ~Mitarbeiter() { cout << "Destruktor Mitarbeiter"
      << endl; }
};
```

Virtuelle Methoden

Was gibt das Programm aus?

```
class Firma {
private:
  Mitarbeiter ceo;
public:
  Firma() { cout << "Konstruktor Firma" << endl; }
  ~Firma() { cout << "Destruktor Firma" << endl; }
};

int main() {
  Firma f;
  cout << "Tschuess!" << endl;
  return EXIT_SUCCESS;
}
```

Zugriffsmodifikatoren

- Eine Klasse besteht aus Members – das sind Daten (Attibute) und Funktionen (Methoden)
- Member können `private`, `protected` und `public` sein
- `private` (default): Der Member kann nur innerhalb von Funktionen derselben Klasse verwendet werden (und friends)
- `protected`: Der Member ist zusätzlich in Funktionen von Subklassen verwendbar (und dessen friends)
- `public`: Alle Funktionen haben Zugriff auf den Member
- Subklassen (abgeleitete Klassen) erben nun alle Member der Superklasse (Basisklasse) und können – sofern sie Zugriff haben – darauf zugreifen, als wären sie in der Subklasse selbst definiert

Zugriffsmodifikatoren

- Der Zugriff auf Superklassen wird ebenfalls gesteuert
  `class B : public A`
- `private` (default): Alle `public` und `protected` Member von A können nur innerhalb von Funktionen von B verwendet werden (und friends) und auch nur dort kann ein `B*` zu ein `A*` gecastet werden (und in friends).
- `protected`: Zusätzlich sind die `public` und `protected` Member von A auch in Subklassen von B verfügbar (und friends) und auch dort kann ein `B*` zu ein `A*` gecastet werden (und in friends).
- `public`: Alle Funktionen haben Zugriff auf `public` Member von B und alle Funktionen können `B*` auf `A*` casten.

Zugriffsmodifikatoren

Gegeben sind folgende Klassen:

```
class Produkt {
public:
  Produkt(double preis) :preis{ preis } {}
  double getPreis() { return preis; }
private:
  double preis;
};
class Kleidungsstueck : public Produkt {
public:
  Kleidungsstueck(double preis, double groesse) : Produkt{
      preis }, groesse{ groesse } {}
  double getGroesse() { return groesse; }
private:
  double groesse;
};
```

Ich brauche eine/n Freiwillige/n!

Zugriffsmodifikatoren

Welche Fehler findet ihr?

```
int main() {
  // 1)
  Produkt p{ 1 };
  cout << p.getGroesse() << endl;

  // 2)
  Kleidungsstueck k1{ 1,2 };
  cout << k1.getPreis() << endl;

  // 3)
  Kleidungsstueck k2{ 1 };
  cout << k2.getGroesse() << endl;

  return EXIT_SUCCESS;
}
```

Übung

Eure Projektmanagerin braucht ein Programm, welches Kundendaten zeilenweise aus einer Textdatei einliest. Die eingelesenen Daten sollen über einen Logger geloggt werden. Es gibt zwei Arten von Loggern: Einen Konsolenlogger, welcher strings in der Konsole ausgibt, und einen Filelogger, welcher strings in Textdateien schreiben kann.
Beim Programmstart kann über eine Eingabe festgelegt werden, welche Art von Logger verwendet werden soll. Handelt es sich um einen Filelogger, ist auch noch zusätzlich der Name der zu schreibenden Logdatei einzugeben.
Schreibe ein Programm, welches diese Anforderungen mithilfe von Polymorphie löst! Gib reservierten dynamischen Speicher frei und schließe geöffnete Files – berücksichtige auch Fehlerfälle!

Erweiterungen

Folgendes ist weiters zu recherchieren und hinzuzufügen:

- Verwende einen geeigneten „Smart Pointer“, um dynamischen Speicher automatisch wieder freizugeben
- Füge dem Programm eine eigene Klasse für Kunden hinzu
  - Ein Kunde besitzt eine Kundennummer und einen Namen
  - Ein Kundenobjekt kann über den <<-Operator in ein strukturiertes Format umgewandelt werden
  - Ein Kundenobjekt kann über den >>-Operator eingelesen werden – denke an Fehlerfälle!
- Finde geeignete Konstruktoren, Destruktoren und (falls nötig) weitere notwendige Methoden!
- Verwende die implementierte Kundenklasse für das Einlesen der Kundendaten und die Ausgabe im Logger!