Materialpaket 06_POLY – Polymorphism

C/C++, Autor: Prof. Dr.-Ing. Carsten Link

Version 1.3.1 March 3, 2019

Contents

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten, Fertigkeiten zur Problemlösung):

Sie können Mechanismen bei der Implementierung einsetzen, welche den Programmablauf dynamisch gestalten.

Die oben genannten Kompetenzen erwerben Sie, indem Sie Lernziele erreichen, welche sich prüfen lassen. Lernegebnisse: Sie können nachweislich[1]:

- Funktionszeiger einsetzen, um das Verhalten eines Programmes dynamischer zu gestalten

[1] Sie können das Erzielen der einzelnen Lernergebnisse beispielsweise bei einem Testat im Praktikum oder einer Aufgabe in der Modulprüfung nachweisen

- virtuelle Methoden einsetzen, um 1) das Verhalten von Objekten von abgeleiteten Klassen zu modifizieren und 2) abstrakteren Code auf der Nutzerseite zu bekommen
- Funktionen und Methoden überladen
- den statischen und die möglichen dynamischen Typen von Objekten von Klassen aus einer Vererbungshierarchie benennen
- mit Hilfe von Vererbung, Überschreiben, Überladen und Typumwandlungen gezielt Methoden von Klassen innerhalb einer Vererbungshierarchie aufrufen lassen

2 Konzepte

Im Folgenden wird das Konzept des *Subtyping Polymorphism* vorgestellt, welches es ermöglicht, Programme zu schreiben, welche erst zur Laufzeit bestimmen, welche Methodenimplementierungen ausgeführt werden sollen.

2.1 Funktionszeiger

Im folgenden Quelltext deklariert `typedef void (*funcPointer_void_void)(void);` einen Funktionszeiger. Der Typ `funcPointer_void_void` steht nach dem `typedef` für Zeiger auf Funktionen, welche `void` als Argumentenliste haben und `void` als Rückgabewert. Die vier Funktionen `sing_gingle()` bis `sing_safari()` haben eine solche Signatur, so dass ihre Adresse in einer Variablen vom Typ `funcPointer_void_void` gespeichert werden kann.

```
1  typedef void (*funcPointer_void_void)(void);
2
3  void sing_gingle(void){
4    print("gingle ");
5  }
6
7  void sing_bells(void){
8    print("bells ";
9  }
10
11 void sing_surf(void){
12   print("surf ");
13 }
14
15 void sing_safari(void){
16   print("surfin safari ");
17 }
```

Die Funktion `songOfSeason()` befüllt einen Vektor von Funktionszeigern (`std::vector<funcPointer_void_void>`) mit Aufrufen von Funktionen (in Abhängigkeit von der Temperatur wird ein angemessenes Lied zusammengestellt).

```
std::vector<funcPointer_void_void> songOfSeason(int currentTemperatureCelsius) {
  std::vector<funcPointer_void_void> result;

  if(currentTemperatureCelsius <= 0){
    result.push_back(sing_gingle);
    result.push_back(sing_bells);
    result.push_back(sing_gingle);
    result.push_back(sing_bells);
  }else{
    result.push_back(sing_surf);
    result.push_back(sing_surf);
    result.push_back(sing_safari);
    result.push_back(sing_surf);
    result.push_back(sing_surf);
    result.push_back(sing_safari);
  }

  return result;
}
```

Die Funktion `sing()` lässt nun von `songOfSeason()` einen Vektor befüllen und ruft im Anschluss daran für jedes Element im Vektor die Funktion auf, auf die das Element (also der Funktionszeiger) zeigt.

```
void sing(int currentTemperatureCelsius){
  std::vector<funcPointer_void_void> song = songOfSeason(currentTemperatureCelsius);

  // traditional C-style for-loop
  for(int i=0; i<song.size(); ++i){
    song[i](); // invoke funtion (pointed to by function pointer at song[i])
  }
  print("");

  // C++11 range-based for-loop looks better:
  for(auto func: song){
    func();
  }
  println("");
}
```

Der Aufruf von `sing(35)` sorgt für folgende Ausgabe:

```
surf surf surfin safari surf surf surfin safari
surf surf surfin safari surf surf surfin safari
```

Wohingegen der Aufruf von `sing(-17)` für diese Ausgabe sorgt:

```
1 gingle bells gingle bells
2 gingle bells gingle bells
```

Das obige Beispiel verdeutlich, dass es mit Funktionszeigern möglich ist, erst zur Laufzeit zu bestimmen, welche Funktionen aufgerufen werden sollen (späte/dynamische Bindung).

Bei gewöhnlichen (unmittelbaren) Aufrufen steht bereits zur Übersetzungszeit fest, welche Funktion später zur Laufzeit einmal aufgerufen werden wird – und zwar diejenige, deren Signatur auf die zur Übersetzungszeit ersichtlichen Typen der Parameter passt (frühe/statische Bindung des Funktionsaufrufs).

Unumgänglich ist die Verwendung von Funktionszeigern, wenn der Aufrufende Code nicht entscheiden kann, welche Funktion zur Laufzeit konkret aufgerufen werden soll – eine Selektion per `if` ist dann nicht möglich. Dies kann daran liegen, dass der aufrufende Code die Bedingungen nicht kennt oder die Funktion. Beispiel: die Funktion `qsort()` aus der C-Standardbibliothek `stdlib.h` kann Arrays gleichartiger Elemente sortieren. Dazu wird eine Vergleichsfunktion (d.h. `a < b`) benötigt, welche `qsort` nicht kennen kann, da `qsort` ja bereits die Typen der Elemente nicht kennt (es wird mit `void *` gearbeitet).

2.2 Umsetzung virtueller Methoden

Beim Aufruf von virtuellen Methoden wird erst zur Laufzeit bestimmt, welche Methode tatsächlich aufgerufen wird (*late binding*, späte/dynamische Bindung). Die konkrete Umsetzung dieses Mechanismus' bleibt dem Compiler (-hersteller) überlassen. Es hat sich jedoch als nützlich herausgestellt, sich die Implementierung durch den Compiler wie folgt vorzustellen:

Zusätzlich zu den Daten eines Objektes wird eine Tabelle gespeichert (*vtable* genannt). Diese Tabelle enthält Funktionszeiger, welche auf die virtuellen Methoden zeigen. Für den Aufruf einer virtuellen Methode erzeugt der Compiler Code, welcher aus der vtable den Zeiger auf die Methode holt und diese dann mit den gegebenen Parametern aufruft. Der Compiler erzeugt für jede Klasse eine eigene vtable (sofern virtuelle Methoden vorhanden sind).

Unter bestimmten Umständen kann auch bei virtuellen Methoden eine frühe bzw. statische Bindung des Methodenaufrufs stattfinden: 1) durch Vollqualifizierung des Methodennamens und 2) Optimierung des Compilers (z. B. wenn der Compiler den konkreten Laufzeittyp vorhersagen kann). Oft wird das statische Binden eines Aufrufs einer virtuellen Methode benötigt, um innerhalb einer überschriebenen Methode (in einer abgeleiteten Klasse) die Basisklassenimplemtierung dieser Methode aufzurufen.

2.3 Überladen von Funktionen und Methoden

In Kapitel 04_UDEFT wurde Operatorenüberladung vorgestellt. Hierbei ist es möglich, einem Operator mehrere verschiedene Implementierungen zu geben, wobei die zu verwendende Implementierung anhand der Typen der Operanden herausgesucht wird.

C++ ermöglicht es darüber hinaus, einer Funktion oder Methode mehrere verschieden Implementierungen zu geben, welche sich in den Typen der Parameter unterscheiden. Dies wird überladen genannt.

Hier ist die Funktion `foo` überladen; es gibt eine Implementierung, die einen `int`-Parameter erwartet und eine, die einen `std::string`-Parameter erwartet:

```
1  void foo(int i){
2    // ...
3  }
4
5  void foo(std::string s){
6    // ...
7  }
8
9  void bar(){
10   foo(1);
11   foo("hello")
12 }
```

Überladen wird vom Compiler während des Übersetzungsvorgangs aufgelöst (Gegensatz: überschriebene `virtual`-Methoden zur Laufzeit).

Beim Überladen von Funktionen und Methoden können Mehrdeutigkeiten entstehen, wenn implizite Typumwandlungen möglich sind (z.B. über Konstruktoren mit einem Parameter).

2.4 Bindung von Methodenaufrufen

Gegeben sind die folgenden Klassen:

```
1  #include <iostream>
2  #include <vector>
3
4  class Base {
5  public:
6    void           overloadedMethod();
7    void           overloadedMethod(int);
8    void           overloadedMethod(std::string);
9    void           nonVirtualMethod(void);
10   virtual void      virtualMethod(void);
```

```
};

class Derived_1 : public Base {
public:
  Derived_1();
  ~Derived_1();
  void           nonVirtualMethod(void);
  virtual void      virtualMethod(void);
};

class Derived_2 : public Derived_1 {
public:
  void           nonVirtualMethod(void);
  virtual void      virtualMethod(void);
};
```

Die Methoden sind so definiert:

```
// see https://isocpp.org/wiki/faq/strange-inheritance#calling-virtuals-from-ctors
Derived_1::Derived_1(){
  virtualMethod();                    // 6a
}

Derived_1::~Derived_1(){
  virtualMethod();                    // 6b
}

// --------------------
void    Base::overloadedMethod(void){
  println("Base::overloadedMethod(void)");
}

void    Base::overloadedMethod(int){
  println("Base::overloadedMethod(int)");
}

void    Base::overloadedMethod(std::string){
  println("Base::overloadedMethod(std::string)";
}

void    Base::nonVirtualMethod(void){
  println("Base::nonVirtualMethod(void)");
}

void    Base::virtualMethod(void){
  println("Base::virtualMethod");
}
```

```
30
31  // --------------------
32  void    Derived_1::nonVirtualMethod(void){
33    println("Derived_1::nonVirtualMethod");
34  }
35  void    Derived_1::virtualMethod(void){
36    println("Derived_1::virtualMethod");
37  }
38
39  // --------------------
40  void    Derived_2::nonVirtualMethod(void){
41    println("Derived_2::nonVirtualMethod");
42  }
43  void    Derived_2::virtualMethod(void){
44    println("Derived_2::virtualMethod");
45  }
```

In der Funktion `foobar()` werden nun verschiedene Arten von Methodenaufrufen getätigt:

```
1   /*
2    illustrates binding of method invocations
3    compile time vs. run time binding
4    */
5
6   void foobar(void){
7
8     std::cout << "6a) ";
9     Derived_2* pR = new Derived_2();
10
11    Base*      pBase      = pR;
12    Derived_1* pDerived_1 = pR;
13    Derived_2* pDerived_2 = pR;
14
15    println("1) ", pBase->virtualMethod();
16
17    println("2) ", pBase->nonVirtualMethod());
18
19    println("3) ", pDerived_2->virtualMethod());
20
21    println("4) ", pDerived_2->nonVirtualMethod());
22
23    println("5", static_cast<Base*>(pDerived_2)->nonVirtualMethod());
24
25    println("7) ", static_cast<Derived_1*>(pDerived_2)->nonVirtualMethod());
26
27    println("8) ", pDerived_1->nonVirtualMethod());
```

```
28
29    println("9) ", pDerived_1->virtualMethod());
30
31    println("10a) ", pDerived_1->Base::virtualMethod());
32
33    println("10b) ", pDerived_2->Derived_1::virtualMethod());
34
35    println("11 ", pDerived_2->overloadedMethod(17));
36
37    println("12 ", pDerived_2->overloadedMethod());
38
39    println("13 ", pDerived_2->overloadedMethod(std::string("x"));
40
41    println("14 ", dynamic_cast<Derived_2*>(pBase)->virtualMethod());
42
43    print("15 ");
44    Base* baseObject    = new Base();
45    Derived_2* d2Object = dynamic_cast<Derived_2*>(baseObject);
46    if(d2Object){
47      println("+");
48    }else{
49      println("-");
50    }
51
52    println("6b) ");
53    delete pR;
54  }
55
56  int main(int argc, const char * argv[]) {
57    foobar();
58    return 0;
59  }
```

Dies ergibt folgende Ausgabe:

```
1  6a) Derived_1::virtualMethod
2  1) Derived_2::virtualMethod
3  2) Base::nonVirtualMethod(void)
4  3) Derived_2::virtualMethod
5  4) Derived_2::nonVirtualMethod
6  5) Base::nonVirtualMethod(void)
7  7) Derived_1::nonVirtualMethod
8  8) Derived_1::nonVirtualMethod
9  9) Derived_2::virtualMethod
10 10a) Base::virtualMethod
11 10b) Derived_1::virtualMethod
12 11 Base::overloadedMethod(int)
```

```
12 Base::overloadedMethod(void)
13 Base::overloadedMethod(std::string)
14 Derived_2::virtualMethod
15 -
6b) Derived_1::virtualMethod
```

Zur Ermittlung der zur Laufzeit aufgerufenen Methode müssen zwei Typen betrachtet werden: 1) der statische Typ, den der Compiler zur Übersetzungszeit ermittelt; 2) der dynamische Typ, also der konkrete Typ des Objektes, auf das ein Zeiger (oder eine Referenz) verweist. Die obigen Ausgaben sind so begründet:

- bei 1, 3, 9 und 14 wurde der Aufruf über den Virtual-Mechanismus abgewickelt – unabhängig vom statischen Typ des Zeigers auf das Objekt. In allen Fällen wird `Derived_2::virtualMethod` aufgerufen, da das Zielobjekt vom Typ `Derived_2` ist
- bei 2, 4, 5, 7 und 8 wird jeweils diejenige Implementierung aufgerufen, die sich in der Klasse findet, die der Compiler als Typ des Zielobjektes des Pointers links des `->`-Operators. Sprich: hat der Compiler als statischen Zeigertyp `X *` ermittelt, so wird `X::nonVirtualMethod()` aufgerufen
- die Aufrufe in Konstruktoren und Destruktoken werden statisch gebunden, da es sonst geschehen könnte, dass Methoden auf nicht fertig konstruierten Objekten ausgeführt werden (6a, 6b)
- bei 10a und 10b wird der `virtual`-Mechanismus explizit durch Vollqualifizierung umgangen

Es existieren zwei Arten von Typumwandlungen (type casts):

1. upcast: ein Zeiger vom Typ einer abgeleiteten Klasse wird auf den Typ einer Basisklasse gecastet. Dies kann implizit durch den Compiler geschehen
2. downcast:ein Zeiger vom Typ einer Basisklasse wird in einen Typ einer abgeleiteten Klasse gecastet. Dies muss explizit geschehen (mit `static_cast<>()` oder `dynamic_cast<>()`)

3 Material zum aktiven Lernen

3.1 Aufgabe: Grundgerüst

Nehmen Sie als Grundgerüst den Quelltext aus dem vorherigen Materialpaket mit mindestens den Klassen `Shape`, `Rectangle` und `Cicle`.

Ihr Grundgerüst für das Testat darf die Modifikationen nicht enthalten!

3.2 Aufgabe: Modifikationen

Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (5 - 10) vor Zuschauern (Testatsituation) umsetzen können. Konkret sollen Sie im Testat in der Lage sein, das gegebene Grundgerüst um mindestens eine zufällig ausgewählte Modifikation zu erweitern. Bereiten Sie dazu auf ihrer Arbeitsumgebung ein Verzeichnis vor, welches ausschließlich das Grundgerüst enthält.

Modifikationen:

1. Fügen Sie den Klassen `Shape`, `Rectangle` und `Circle` eine nicht virtuelle Methode `void nonVirtual(void)` hinzu. Implementieren Sie eine Funktion `void invokeVirtually(Shape* theShape)`, welche dafür sorgt, dass abhängig von `theShape`s Laufzeittyp entweder `Rectangle::nonVirtual()` oder `Circle::nonVirtual()` aufgerufen wird (also die Implementierung aufgerufen wird, die in der Klasse definiert ist, zu welchem das Objekt gehört, auf das `theShape` zeigt)
2. Fügen Sie der Klasse `Shape` eine nicht-virtuelle Methode hinzu, welche eine Methode aufruft, welche in `Shape` rein virtuell ist und in den abgeleiteten Klassen `Rectangle` und `Circle` überschrieben wird.
3. Freiwillig für Fortgeschrittene: fügen Sie in `Shape` einen Funktionszeiger hinzu, der in den Konstruktoren von `Circle` etc. auf jeweils eine Funktion gesetzt wird, die ein per `Shape *` gegebenes Objekt zeichnen kann. Effektiv bilden Sie damit `virtual void draw()` nach
4. Nehmen Sie in `main_mp5_POLY.cpp` als Vorlage und erstellen eine Funktion `void foobar_2()` mit den gleichen lokalen Variablen wie in `void foobar()`. Erstellen Sie nun die folgenden Aufrufe:
   - `Derived_2::virtualMethod()` über `pBase`
   - `Derived_2::virtualMethod()` über `pDerived_1`
   - `Derived_1::nonVirtualMethod()` über `pDerived_2`
   - `Derived_2::nonVirtualMethod()` über `pDerived_2`
   - `Base::virtualMethod()` über `pDerived_1`

3.3 Verständnisfragen

Nach Bearbeitung des Kapitels “Konzepte”, der Erstellung des Grundgerüsts sowie dem Üben der Modifikationen sollten Sie in der Lage sein, die folgenden Fragen zu beantworten.

1. Wozu wird das statische Binden des Aufrufs einer virtuellen Methode zwingend benötigt?
2. Zu welchem Zeitpunkt wird festgelegt, aus welcher Klasse die Implementierung einer virtuellen Methode stammt, die aufgerufen wird?
3. Zu welchem Zeitpunkt wird festgelegt, welche Methodensignatur verwendet wird, um einen Aufruf einer überladenen Methode zu generieren?

4. Muss für einen upcast `static_cast` oder `dynamic_cast` verwendet werden? Warum ja/nein?
5. Muss für einen downcast `static_cast` oder `dynamic_cast` verwendet werden? Warum ja/nein?
6. Kann der Compiler einen downcast implizit vornehmen? Begründung angeben.
7. Kann über den Zeiger auf ein Objekt einer abgeleiteten Klasse die Basisklassenimplementierung einer virtuellen Methode aufgerufen werden (z.B. `this` in einer überschriebenen Methode)? Wenn ja: wie?
8. Kann über den Zeiger auf ein Objekt einer Basisklasse eine Methode aufgerufen werden, welche in einer abgeleiteten Klasse definiert ist und nicht virtuell ist? Wenn ja: wie?
9. Für Fortgeschrittene: Wie geht der Compiler beim Aufruf einer Methode vor, welche überladen *und* virtuell ist?
10. Diskutieren Sie den Zusammenhang von Funktionszeigern und Interrupt Handlern.

4 Nützliche Links

- Wikipedia: Tabelle virtueller Methoden[2]
- C++ FAQ Lite[3]

5 Literatur

- [PPP] Stroustrup, Bjarne: Programming - Principles and Practice using C++
- [TCPL] Stroustrup, Bjarne: The C++ Programming Language, Fourth Edition

[2] https://de.wikipedia.org/wiki/Tabelle_virtueller_Methoden
[3] http://yosefk.com/c++fqa/index.html