C/C++ Materialpaket (Level C)
05d_OO_BIND – Object Orientation (Function Binding)

Prof. Dr. Carsten Link

Zusammenfassung

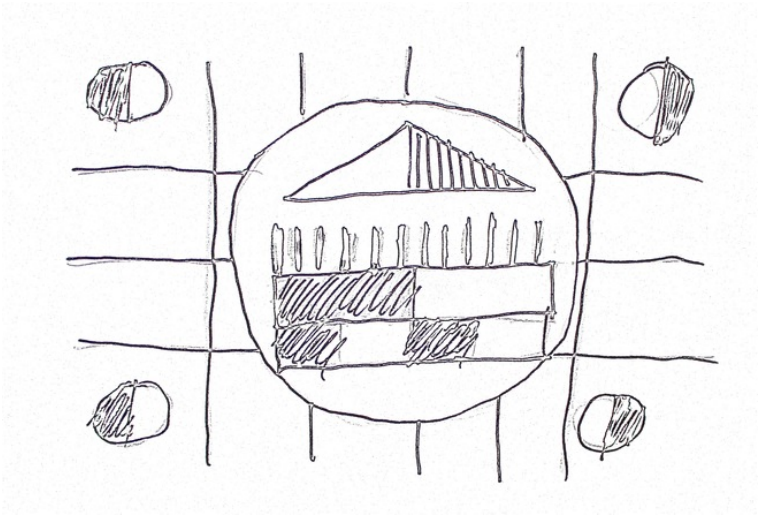

Abbildung 1: Testbild

Inhaltsverzeichnis

1 Kompetenzen und Lernegebnisse **2**

2 Konzepte **2**
2.1 Frühe Bindung von Funktionen und Methoden 2
2.2 Späte Bindung mit Funktionspseudonymen 3
2.3 Späte Bindung mit virtuellen Methoden 7
2.4 Umsetzung virtueller Methoden 7
2.5 Frühe und späte Bindung von Methodenaufrufen im Vergleich . . 8

3 Material zum aktiven Lernen **10**
3.1 Aufgabe: Grundgerüst 10
3.2 Aufgabe: Modifikationen 11

3.3 Verständnisfragen 12

4 Nützliche Links **12**

5 Literatur **13**

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten und 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
- 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 Frühe Bindung von Funktionen und Methoden

In Kapitel 04_UDEF 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 (overloading) genannt.

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

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 (daher frühe Bindung; wird auch statische Bindung genannt) und ad-hoc polymorphism genannt. Die beiden `foo()`-Funktionen könnten auch Teil einer Klasse (member functions) sein – Function overloading ist auch in diesem Fall möglich.

2.2 Späte Bindung mit Funktionspseudonymen

Mit Pseudonymen ist es im Alltag möglich, echte Namen zu verbergen. Der in diesem Kapitel vorgestellte C++-Mechanismus macht sich eine andere Eigenschaft von Pseudonymen zu Nutze: hinter einem Pseudonym muss sich nicht immer dasselbe Ding oder dieselbe Person befinden. So können hinter dem Pseudonym `The Dude` zu verschiedenen Zeitpunkten verschiedene Personen(namen) stecken. Der Begriff Funktionspseudonym meint genau das: hinter einem Funktionspseudonym können sich zur Laufzeit zu verschiedenen Zeitpunkten unterschiedliche Funktionen verbergen (Funktionspseudonyme sind bei vielen Programmiersprachen unter dem Namen *Funktionszeiger* bekannt).

Bei später Bindung von Funktionsaufrufen wird die konkrete Funktion, die aufgerufen werden soll, erst zur Laufzeit festgelegt. Dadurch wird die Stelle des Aufrufs entkoppelt von der Stelle der Festlegung (nicht nur syntaktisch, sondern auch was den (logischen) Zeitpunkt angeht).

Diese Entkopplung macht Programme dynamisch; da nicht bereits zur Übersetzungszeit die konkret aufzurufende Funktion festgelegt werden muss. Weiterhin ist es möglich, zur Laufzeit mehrfach zu ändern, welche konkrete Funktion bei Aufrufen verwendet wird. Beispielsweise könnte in einem Programm an vielen Stellen `sort()` aufgerufen werden, zur Laufzeit jeweils abhängig von den eingelesenen Daten tatsächlich `quick_sort()` oder `merge_sort()` aufgerufen.

Im folgenden Beispiel wird mit Hilfe von Funktionszeigern die Aufruf- von der Festlegungsstelle entkoppelt. Um Fehlersuche und andere Analysen zu erleichtern, wird im Hauptprogramm die Logging-Funktion `log_msg()` aufgerufen. Über die

Kommandozeilenparameter `-l0`, `-l1`, `-l2` oder `-l3` lässt sich festlegen, ob und auf welche Art die Informationsstrings ausgegeben werden, indem an den Funktionsnamen `log_msg` jeweils unterschiedliche Funktionen gebunden werden. Diese Festlegung geschieht in `setup_logger()` abhängig von den Kommandozeilenparametern (die von `main()` durchgereicht werden).

Betrachtet wird folgendes Hauptprogramm:

```
#include "logger.hpp"
#include <iostream>
#include <cassert>

int binary_search(std::string s, char c);
int linear_search(std::string s, char c);

int main(int argc, char* argv[]){
  std::cout << "Welcome! You may append append -l0 -l1 -l2 or -l3 to a.out" << std::endl;
  setup_logger(argc, argv);
  //test_search();
  std::string text = "jklmnqpo"; //"abcdefghijklmnopqrstuvwxyz";
  char c = 'p';
  int pos1 = linear_search(text, c);
  int pos2 = binary_search(text, c);
  if (pos1 == pos2) {
    log_msg(DEBUG, "binary_search OK!     text=" + text + " c=" + c);
  } else {
    log_msg(ERROR, "binary_search BROKEN! text=" + text + " c=" + c);
  }
  return 0;
}
```

Das obige Programm ist noch nicht fertiggestellt. Da noch kleinere Fehler behoben werden müssen, wird oft gebrauch gemacht von `log_msg(DEBUG...`, um Diagnosen zu vereinfachen[2].

Wird das erzeugte Programm ohne Kommandozeilenparameter gestartet, erscheint nur eine Willkommensnachricht – von den Logging-Nachrichten ist nichts zu sehen:

```
$ ./a.out
Welcome! You may append append -l0 -l1 -l2 or -l3 to a.out
```

Wird das erzeugte Programm mit dem Kommandozeilenparameter `-l1` gestartet, erscheint diese Ausgabe:

```
$ ./a.out -l1
Welcome! You may append append -l0 -l1 -l2 or -l3 to a.out
```

[2]Der vollständige Code befindet sich in `src-cpp-student/05_OOd/z_funcpointers_logging/`

```
ERROR: binary_search broken! 1
ERROR: binary_search broken! 2
```

Wird das erzeugte Programm mit dem Kommandozeilenparameter `-l2` gestartet, erscheint diese Ausgabe:

```
$ ./a.out -l2
Welcome! You may append append -l0 -l1 -l2 or -l3 to a.out
DEBUG: binary_search s=jklmnqpo
ERROR: binary_search broken! 1
DEBUG: linear_search s=jklmnqpo
DEBUG: linear_search s[i]=j
DEBUG: linear_search s[i]=k
DEBUG: linear_search s[i]=l
DEBUG: linear_search s[i]=m
DEBUG: linear_search s[i]=n
DEBUG: linear_search s[i]=q
DEBUG: linear_search found it! i=5
DEBUG: binary_search s=jklmnqpo
ERROR: binary_search broken! 2
```

Bei der Angabe von `-l3` erscheint keine Ausgabe; es wird stattdessen in die Datei `log.txt` geschrieben.

Bei den verschiedenen Durchläufen oben wird ersichtlich, dass erst zur Laufzeit entschieden wird, welche Logging-Funktion aufgerufen werden soll. Die oben gezeigte Umsetzung mittels eines Funktionspseudonyms (Funktionszeiger) hat folgende Vorteile: 1) um den Aufruf von `log_msg()` ist kein `if-else` nötig, um eine aufzurufende Funktion auszuwählen; 2) die aufrufende Stelle muss die effektiv aufgerufene Funktion nicht kennen (es besteht daher keine Abhängigkeit dazu).

Das oben beschriebene Verhalten wird im Programm wie folgt implementiert: In `logger.hpp` deklariert `typedef void (*logger_func)(std::string category, std::string message);` einen Funktionszeigertyp. Der Typ `logger_func` steht demnach für Zeiger auf Funktionen, welche `(std::string, std::string)` als Argumentenliste haben und `void` als Rückgabewert. Die vier Funktionen `no_logging()`, `log_level1()`, `log_level2()`, `log_level3()` habe eine solche Signatur, so dass ihre Adresse in einer Variablen vom Typ `logger_func` gespeichert werden kann. Die in `logger.hpp` deklarierte Variable `log_msg` hat diesen Typ und wird in `main.cpp` für Funktionsaufraufe verwendet. In der Funktion `setup_logger()` ist zu sehen, wie die Bindung der Variablen `log_msg` zur Laufzeit an eine der vier Funktionen von statten geht:

```
#include "logger.hpp"

logger_func log_msg = no_logging;

```

```
5  void no_logging(std::string , std::string){
6  }
7
8  void log_level1(std::string category, std::string message){
9    if (category == ERROR || category == WARNING) {
10     std::cout << category + message << std::endl;
11   }
12 }
13
14 void log_level2(std::string category, std::string message){
15   std::cout << category + message << std::endl;
16 }
17
18 void log_level3(std::string category, std::string message){
19   write_to_file(category + message);
20 }
21
22 // do the wiring of logging functions in effect
23 // (based on log level provided at the command line)
24 void setup_logger(int argc, char* argv[]){
25   if(argc == 2){
26     if (std::string(argv[1]) == std::string("-l3")) {
27       log_msg = log_level3;
28     } else if (std::string(argv[1]) == std::string("-l2")) {
29       log_msg = log_level2;
30     } else if (std::string(argv[1]) == std::string("-l1")) {
31       log_msg = log_level1;
32     } else { /// "-O" is also the default case
33       log_msg = no_logging;
34     }
35   }
36 }
```

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 nicht

kennt. 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). Da C keine Operatorenüberladung unterstützt, wird an `qsort` ein Funtkionszeiger zum Vergleich zweier Elemente übergeben.

Level C: Das Programm mit der Funktion `binary_search()` enthält einen Fehler. Finden Sie diesen. Hinweis: die Ursache des Fehlers legt es nahe, am Anfang der Funktion `binary_search()` mindestens ein `assert()` einzubauen, um Vorbedingungen zu prüfen.

2.3 Späte Bindung mit virtuellen Methoden

Bei der späten Bindung (auch dynamische Bindung genannt) von Methodenaufrufen wird erst zur Laufzeit bestimmt, welche Methodenimplementierung (also welcher Methodenrumpf) aufgerufen wird. Zur Übersetzungszeit wird nur bestimmt, welche Signatur die Methode hat. Eine Oberklasse deklariert eine `virtual`-Methode, die von abgeleiteten Klassen überschrieben werden kann (d.h. die abgeleitete Klasse liefert eine eigene Implemtierung der Methode). Zur Laufzeit bestimmt dann der konkrete Typ des Objektes, welche Methodenimplementierung für den Aufruf verwendet wird.

Der Mechanismus zur Umsetzung der späten Bindung mit virtuellen Methoden wird in den nachfolgenden Kapiteln erläutert.

2.4 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.

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

2.5 Frühe und späte Bindung von Methodenaufrufen im Vergleich

Gegeben sind die folgenden Klassen:

```
#include "println.hpp"

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

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

Die Methoden sind so definiert:

```
void    Base::nonVirtualMethod(void){
  std::cout << "Base::nonVirtualMethod" << std::endl;
}

void    Base::virtualMethod(void){
  std::cout << "Base::virtualMethod" << std::endl;
}

void    Derived::nonVirtualMethod(void){
  println("Derived::nonVirtualMethod");
}
void    Derived::virtualMethod(void){
  println("Derived::virtualMethod");
}
```

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

```
void lookupMethods(void){
  Derived* theObject = new Derived();

```

```
4   Base*      pBase  = theObject;
5   Derived* pDerived = theObject;
6
7   print("2) ");
8   pBase->nonVirtualMethod();              // 2)
9
10  print("8) ");
11  pDerived->nonVirtualMethod();         // 8)
12
13  print("5) ");
14  static_cast<Base*>(pDerived)->nonVirtualMethod(); // 5)
15
16  print("1) ");
17  pBase->virtualMethod();                 // 1)
18
19  print("9) ");
20  pDerived->virtualMethod();            // 9)
21
22  print("10_1) ");
23  static_cast<Base*>(pDerived)->virtualMethod();           // 10_1)
24
25  print("10a) ");
26  pBase->Base::virtualMethod();           // 10a)
27
28
29  print("15 :");
30  Base* baseObject    = new Base();
31  Derived* d2Object = dynamic_cast<Derived*>(baseObject);
32  if(d2Object){                           // 15)
33    println("ok, lets invoke methods on d2Object!");
34    d2Object->virtualMethod();
35  }else{
36    println("I'm sorry, Dave. I'm afraid I can't do that.");
37  }
38
39  delete theObject;
40 }
```

Dies ergibt folgende Ausgabe:

```
2) Base::nonVirtualMethod
8) Derived::nonVirtualMethod
5) Base::nonVirtualMethod
1) Derived::virtualMethod
9) Derived::virtualMethod
10_1) Derived::virtualMethod
```

```
10a) Base::virtualMethod
15 :I'm sorry, Dave. I'm afraid I can't do that.
```

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

- bei 2, 5 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 `T *` ermittelt, so wird `T::nonVirtualMethod()` aufgerufen
- bei 1, 9 und 10.1 wurde der Aufruf über den Virtual-Mechanismus abgewickelt – unabhängig vom statischen Typ des Zeigers auf das Objekt. In allen Fällen wird `Derived::virtualMethod` aufgerufen, da das Zielobjekt vom Typ `Derived` ist
- bei 10a wird der `virtual`-Mechanismus explizit durch Vollqualifizierung umgangen
- **Level C:** 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)

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

Regelmäßiger Hinweis: Da eine Programmiersprache nur durch aktive Verwendung erlernt werden kann, werden im Folgenden Aufgaben zum praktischen Üben vorgestellt. Zunächst wird ein Grundgerüst (C/C++Programm) erstellt, welches dann auf mehrere Arten modifiziert wird. Insbesondere die Modifikationen ermöglichen es dem Lernenden (und auch dem Lehrenden), die Qualität des Kompetenzerwerbs bzgl. dieses Materialpakets bewerten zu können.

3.1 Aufgabe: Grundgerüst

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

Hinweis: Ihr Grundgerüst für das Testat darf die unten aufgeführten Modifikationen nicht enthalten!

3.2 Aufgabe: Modifikationen

Regelmäßiger Hinweis: Weiter unten ist eine Liste mit Modifikationen gegeben, die zwei Zwecken dienen: 1) Sie dienen als Richtschnur für das Praktizieren und Üben der Inhalte dieses Materialpakets. 2) Die Modifikationen können im Rahmen eines Testats als Aufgabe verwendet werden, durch deren Lösung Studierende nachweisen können, dass sie den Stoff dieses Materialpakets beherrschen. Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (ca. 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. **Arbeiten Sie also auf einer Kopie des Verzeichnisses `src-cpp-student/`!** Achten Sie darauf, dass der Text auf Ihrem Bildschirm in heller Umgebung aus einem Meter Abstand heraus gut lesbar ist (light mode, große Schrift).

Modifikationen:

1. Fügen Sie den Klassen `Shape`, `Rectangle` und `Circle` eine nicht virtuelle Methode `void nonVirtual(void)` hinzu. Implementieren Sie eine freistehende 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. Nehmen Sie in `main_06_BIND_levelAB.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::virtualMethod()` über `pBase`
   - `Derived::virtualMethod()` über `pDerived`
   - `Base::nonVirtualMethod()` über `pBase`
   - `Base::nonVirtualMethod()` über `pDerived`
3. **Level C:** Fügen Sie in `Shape` einen Funktionszeiger hinzu, der in den Konstruktoren von `Circle` etc. auf jeweils eine globale Funktion gesetzt wird, die ein per `Shape *` gegebenes Objekt zeichnen kann (d.h. die Funktionen erhalten jeweils einen `Shape`-Pointer auf konkretere Objekte der Klassen `Rectangle`, `Cirlce` etc.). Effektiv bilden Sie damit `virtual void draw()` nach (ggf. benötigen Sie eine `friend`-Deklaration dazu)
4. **Level C:** erstellen Sie `invokeVirtually_a()` und `invokeVirtually_b()`, so dass a) der Operator `typeid()` (aus `<typeinfo>`) und b) ein Typkennungs-`int` in `Shape` verwendet wird. Vergleichen Sie die Ergebnisse, insbesonderen, wenn Sie Pointer auf `Triangle`-Objekte übergeben. Hierbei ist `Triangle` eine Klasse, die von `Shape` abgeleitet ist, aber in `invokeVirtually_a()` und `invokeVirtually_b()` nicht berücksichtigt wird, da Sie erst später erstellt wurde).

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 einen Basisklassenzeiger eine Methode aufgerufen werden, welche in einer abgeleiteten Klasse definiert ist und nicht virtuell ist? Wenn ja: wie?
9. Freiwillig: Zeigen Sie auf, inwiefern das Piktogram auf der Titelseite dieses Materialpaketes den Inhalt zusammengefasst darstellt.
10. **Level C:** Vergleichen Sie die beiden Lösungsmöglichkeiten für `invokeVirtually(Shape* theShape)` (mit `dynamic_cast<>()` oder mit `typeid()`) miteinander. Gibt es Unterschiede im Verhalten? Was passiert, wenn Sie jeweils einen Zeiger auf eine Objekt hineingeben, dessen Klasse von `Rectangle` abgeleitet ist?
11. **Level C:** Wie geht der Compiler beim Aufruf einer Methode vor, welche überladen *und* virtuell ist?
12. **Level C:** Diskutieren Sie den Zusammenhang von Funktionszeigern und Interrupt Handlern.
13. **Level C:** Diskutieren Sie Alternativen zu Funktionszeigern am Beispiel des Loggings. Als Mittel zur Umsetzung bieten sich Kontrollstrukturen, der Präprozessor oder der Linker an.

4 Nützliche Links

- Wikipedia: Tabelle virtueller Methoden[3]
- C++ FAQ Lite[4]

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

5 Literatur

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