Materialpaket 10_PITF – Pitfalls

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 einige C++-Stolperfallen umgehen.

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

- Fehler vermeiden, die sich durch ungünstige Kombination von Zeigern und Compiler-generierten Methoden ergeben können (z. B. double delete)
- Fehler vermeiden, die sich durch Object-Slicing ergeben

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

- aus eigener Erfahrung die Komplexität der manuellen Speicherverwaltung (`new/delete`) und die damit einhergehenden Probleme in Programmen abschätzen
- Smart Pointer der C++-Standardbibliothek verwenden

2 Konzepte

Bei Programmiersprachen bedeutet der Begriff *Orhtogonalität*, dass Sprachkonstrukte beliebig miteinander kombiniert werden können. Sprachen, die sehr orthogonal sind, beherzigen das *Prinzip der geringsten Überraschung*. C++ hat viele Stellen, bei denen Nichtorthogonales durchscheint – gewissen Kombinationen sind gar nicht möglich, andere hingegen sehr fehlerträchtig.

2.1 Pointer und Objektkopien

Die unten angegebene Klasse `LifeCycleProbe` verfügt über einen `int`-Wert, einige Konstruktoren, den Zuweisungsoperator (`=`) und eine Methode `printID()`. Im wesentlichen dient diese Klasse dazu, vom Compiler generierte Kopiervorgänge sichtbar zu machen. Hier die Deklaration:

```
1  class LifeCycleProbe {
2    int id;
3  public:
4    LifeCycleProbe();
5    LifeCycleProbe(int id);
6    LifeCycleProbe(const LifeCycleProbe&);
7    LifeCycleProbe& operator=(const LifeCycleProbe&);
8    ~LifeCycleProbe();
9    void printID();
10 };
```

Die einzelnen Methoden sind wie folgt implementiert:

```
1  //  LifeCycleProbe.cpp
2  //  DoubleDelete
3
4  #include "LifeCycleProbe.hpp"
5  #include <iostream>
6
7  LifeCycleProbe::LifeCycleProbe(){
8    this->id=0;
9    std::cout << "default ctor id=" << id << std::endl;
10 }
11
12 LifeCycleProbe::LifeCycleProbe(int id){
```

```
13    this->id=id;
14    std::cout << "ctor(int) id=" << id << std::endl;
15  }
16
17  LifeCycleProbe::LifeCycleProbe(const LifeCycleProbe& other){
18    std::cout << "copy ctor (source id=" << other.id
19              << " to id=" << other.id + 9 << ")" << std::endl;
20    this->id=other.id + 9;
21  }
22
23  LifeCycleProbe& LifeCycleProbe::operator=(const LifeCycleProbe& other){
24    std::cout << "assignment (old id=" << this->id
25              << " source id=" << other.id
26              << " new id=" << other.id + 11 << ")" << std::endl;
27    this->id = other.id + 11;
28    return *this;
29  }
30
31  LifeCycleProbe::~LifeCycleProbe(){
32    std::cout << "dtor id=" << id << std::endl;
33  }
34
35  void LifeCycleProbe::printID(){
36    std::cout << "printID() id=" << id << std::endl;
37  }
```

In `main()` wird ein Objekt dieser Klasse instantiiert und bei drei Funktionsaufrufen als Parameter verwendet – jedoch jedes mal mit unterschiedlicher Aufrufart:

```
1   //  main.cpp
2   //  DoubleDelete
3
4   #include "LifeCycleProbe.hpp"
5   #include <iostream>
6
7   void callByPointer(LifeCycleProbe* p){
8     p->printID();
9   }
10
11  void callByReference(LifeCycleProbe& r){
12    r.printID();
13  }
14
15  void callByValue(LifeCycleProbe v){
16    v.printID();
17  }
```

```
18
19 int main(int argc, const char * argv[]) {
20   LifeCycleProbe probe(1000);
21   std::cout << "three invocations ===========" << std::endl;
22   callByPointer(&probe);
23   callByReference(probe);
24   callByValue(probe);
25   std::cout << "The End. ===========" << std::endl;
26     return 0;
27 }
```

Das obige Programm erzeugt folgende Ausgabe:

```
ctor(int) id=1000
three invocations ===========
printID() id=1000
printID() id=1000
copy ctor (source id=1000 to id=1009)
printID() id=1009
dtor id=1009
The End. ===========
dtor id=1000
```

Obige Ausgabe macht deutlich, dass für den Aufruf `callByValue()` vom Compiler eine temporäre, anonyme Kopie des Objekts mit der ID `1000` erzeugt wird (neue ID=`1009`), welche gleich nach dem Aufruf wieder zerstört wird.

Da bei den beiden Aufrufen `callByPointer()` und `callByReference()` nicht das Objekt selbst, sondern dessen Adresse übergeben wird, ist keine Kopie nötig. Bei Aufruf per Pointer wird die Adresse explizit übergeben, beim Aufruf implizit durch den Compiler.

Nun wird `main()` erweitert um diese Zeilen:

```
1  {
2    std::cout << "assigment ===========" << std::endl;
3    LifeCycleProbe blockProbe(2000);
4    blockProbe = probe;
5    blockProbe.printID();
6  }
7
8  std::cout << "The End. ===========" << std::endl;
9    return 0;
10 }
```

In einem begrenzten Gültigkeitsbereich (scope) wird eine weitere Variable `blockprobe` erzeugt welche mit dem `operator=` den Wert der Variable `probe` zugewiesen bekommt. In der Ausgabe des Programms ist dieser Vorgang sichtbar:

```
assigment ===========
```

```
ctor(int) id=2000
assignment (old id=2000 source id=1000 new id=1011)
printID() id=1011
dtor id=1011
```

Bis hier treten keine unerwarteten Effekte auf. Wird jedoch nun die Klasse `LifeCycleProbe` um eine Ressource erweitert, die mit einem Pointer referenziert wird, können die vom Compiler generierten Kopiervorgänge Probleme bereiten.

Der Pointer `int *resource;` ist Teil eines jeden `LifeCycleProbe`-Objekts. Er wird in den Konstruktoren mit einer Adresse versehen, welche im Destruktor mit `delete` wieder freigegeben werden muss:

```
1  LifeCycleProbe::LifeCycleProbe(){
2    this->id=0;
3    std::cout << "default ctor id=" << id << std::endl;
4  #ifdef HAS_RESOURCE_POINTER
5    resource = new int;
6  #endif
7  }
8
9  LifeCycleProbe::LifeCycleProbe(int id){
10   this->id=id;
11   std::cout << "ctor(int) id=" << id << std::endl;
12 #ifdef HAS_RESOURCE_POINTER
13   resource = new int;
14 #endif
15 }
16
17 LifeCycleProbe::~LifeCycleProbe(){
18   std::cout << "dtor id=" << id << std::endl;
19 #ifdef HAS_RESOURCE_POINTER
20   std::cout << "deleting resource" << *resource << std::endl;
21   delete resource;
22 #endif
23 }
```

Wird nun das Programm erneut gestartet (mit `#define HAS_RESOURCE_POINTER` in `LifeCycleProbe.hpp`), so erzeugt es einen Fehler und wird abgebrochen (es stürzt ab):

```
ctor(int) id=1000
three invocations ===========
printID() id=1000
printID() id=1000
copy ctor (source id=1000 to id=1009)
printID() id=1009
dtor id=1009
```

```
deleting resource0
assigment ===========
ctor(int) id=2000
assignment (old id=2000 source id=1000 new id=1011)
printID() id=1011
dtor id=1011
deleting resource0
The End. ===========
dtor id=1000
deleting resource0
DoubleDelete(27773,0x100082000) malloc: *** error for object 0x100101c10: pointer being free
```

Das Problem hier ist: der Destruktor des Objektes mit der `ID=1000` versucht die Ressource an `resource` per `delete` zu löschen – diese wurde jedoch bereits von dem Objekt mit der `ID=1009` gelöscht (die Kopie für die Parameterübergabe). Die Ursache des Problems ist, dass der Copy Constructor `LifeCycleProbe::LifeCycleProbe(const LifeCycleProbe& other)` den Wert von `other.resource` kopiert hat – genau wie es der Compiler-generierte Copy Constructor macht.

Fazit: Sobald dynamisch allokierte Ressourcen (z.B. per `new`) in einem Objekt gespeichert sind, so müssen mindestens diese Methoden sorgfältig die Verwiese auf die Ressourcen Pflegen:

- Copy Constructor
- Zuweisungsoperator
- Destruktor (ggf. `virtual`)
- (ab C++11: Move Constructor[2])
- (ab C++11: Move Assignment)

Diese Regel ist auch als “The big Three” bekannt (Seit C++11 als “The big Five”).

1

2.2 Vererbung und Objektkopien

Der Compiler erzeugt häufig Kopien von Objekten beispielsweise, um diese als Parameter an Funktionen zu übergeben oder in Arrays bzw. Containern zu speichern.

In folgenden soll nun beleuchtet werden, wie sich dieser Sachverhalt mit Vererbung verträgt. Gegeben seien die beiden Klassen `Base` und `Derived`:

```
1 class Base{
2   int _a;
3 public:
```

[2] http://en.cppreference.com/w/cpp/language/rule_of_three

```
4     Base(){}
5     virtual ~Base() {}
6     Base(int aa) :_a(aa) {}
7     int value_a();
8     virtual int value_virtual();
9   };
10
11  int Base::value_a(){
12    return _a;
13  }
14
15  int Base::value_virtual(){
16    return _a;
17  }
18
19  class Derived : public Base {
20    int _b;
21  public:
22    Derived(int x, int y) : Base(x), _b(y) {}
23    virtual int value_virtual();
24  };
25
26  int Derived::value_virtual(){
27    return _b;
28  }
29
30  void passByPointer(Base* bp){
31    std::cout << "->value_a()        = " << bp->value_a() << std::endl;
32    std::cout << "->value_virtual()  = " << bp->value_virtual() << std::endl;
33  }
34
35  void passByValue(Base b){
36    std::cout << ".value_a()        = " << b.value_a() << std::endl;
37    std::cout << ".value_virtual()  = " << b.value_virtual() << std::endl;
38  }
```

Die Klasse `Base` verfügt über einen `int _a`, welcher von den beiden Methoden `value_a()` und `value_virtual()` zurückgegeben wird. Die Klasse `Derived` unterscheidet sich. Sie verfügt über ein weiteres Feld `int _b`, welches von der virtuellen Methode `value_virtual` zurückgegeben wird.

Übergeben wir nun einen Zeiger auf ein Objekt der Klasse `Derived` an eine Funktion, die einen `Base`-Zeiger erwartet, so zeigt sich das erwartete Verhalten:

```
1  Derived d(1,2);
2  std::cout << "d.value_virtual() = " << d.value_virtual() << std::endl;
3  std::cout << std::endl << "pass pointer to base class " << std::endl;
```

```
4  passByPointer(&d);
```

gibt aus:

```
d.value_virtual() = 2

pass pointer to base class
->value_a()        = 1
->value_virtual()  = 2
```

Übergeben wir das Objekt `d` nun nicht per Zeiger, sondern als Wert (*call by value*), so wird der Compiler das Objekt der Klasse Derived in ein Objekt der Klasse Base "umwandeln". Hierbei geht die `Derived`-Information des Objektes verloren. Da also nur ein Teil des Objekte kopiert wird, wird dieser Sachverhalt *object slicing* genannt.

```
1  std::cout << std::endl << "pass by value (i.e. object of class Base) " << std::endl;
2  passByValue(d);  // pass Base-object by value -> object slicing: d is "transformed" int an
```

gibt aus

```
pass by value (i.e. object of class Base)
.value_a()        = 1
.value_virtual()  = 1
```

Es wird also nun nicht mehr der in `d` gespeicherte Wert `2` ausgegeben, sondern der Wert `1`.

Ebenso verhält es sich, wenn `Derived`-Objekte an `Base`-Objekte zugewiesen werden (beispielsweise, weil sie in Containern gespeichert werden):

```
1  std::cout << std::endl << "store Derived objects in std::vector<Base>" << std::endl;
2  std::vector<Base> v;
3  v.push_back(d); // object slicing
4  std::cout << ".value_a()        = " << v[0].value_a() << std::endl;
5  std::cout << ".value_virtual()  = " << v[0].value_virtual() << std::endl;
```

gibt aus

```
store Derived objects in std::vector<Base>
.value_a()        = 1
.value_virtual()  = 1
```

Object Slicing stellt kein Problem dar, wenn die sich ergebenden Objekte ohne die fehlenden Teile funktionsfähig sind. Heißt: es existieren keine logischen Zusammenhänge, die nur in Objekten abgeleiteter Klassen korrekt gegeben sind (Invarianten).

Was die Speicherung von polymorphen Objekten angeht, ist Object Slicing ein Problem. Schließlich sollen die Objekte abgeleiteter Klassen vollständig gespeichert werden. Hier verbleibt als Lösung nur das Speichern der Objekte im free store (heap) ggf. indirekt.

Fazit: sollen Objekte polymorph verwendet werden (insbes. Verwendung von virtuellen Methoden), so *muss* die Handhabung der Objekte per Referenz oder Pointer (auf Basisklasse) stattfinden.

2.3 Smart Pointers

Es ist nicht einfach, in der Programmlogik sicherzustellen, dass zu jedem `new` auch das passende `delete` geschieht – und dies genau einmal. Analog zum RAII-Idiom wurden daher Smart Pointers entwickelt, da in beiden Fällen der Compiler dafür sorgt, dass Objekte garantiert gelöscht werden und somit der Aufruf des Destruktors garantiert ist. In Destruktor eines Smart Pointers lässt sich nun sicherstellen, dass nicht mehr benötigte (unreferenzierte) Objekte gelöscht werden.

Die C++-Standardbibliothek bietet diese Smart Pointer an (siehe z. B. *Using C++11's Smart Pointers* [3]):

- `std::unique_ptr<>` exclusive einmalige Ownership. Wird erzeugt mit `std::make_unique<>()`
- `std::shared_ptr<>` Ownership ist auf mehrere Smart Pointer verteilt. Mit Reference Counting. Wird erzeugt mit `std::make_shared<>()`
- `std::weak_ptr<>` Schwache Referenz (nicht Ownership). Wird bei der Erzeugung an ein `std::shared_ptr<>`-Objekt gebunden

Dieses Beispiel (Quelle https://msdn.microsoft.com/de-de/library/hh279674.aspx) zeigt die Verwendung eines `std::unique_ptr<>`:

```
1  class LargeObject
2  {
3  public:
4      void DoSomething(){}
5  };
6
7  void ProcessLargeObject(const LargeObject& lo){}
8  void SmartPointerDemo()
9  {
10     // Create the object and pass it to a smart pointer
11     std::unique_ptr<LargeObject> pLarge(new LargeObject());
12
13     //Call a method on the object
14     pLarge->DoSomething();
15
16     // Pass a reference to a method.
17     ProcessLargeObject(*pLarge);
```

[3] http://www.umich.edu/~eecs381/handouts/C++11_smart_ptrs.pdf

```
18
19 } //pLarge is deleted automatically when function block goes out of scope.
```

Smart Pointers bieten neben den Vorteilen von RAII (s.o.) auch die Möglichkeit, Heapobjekte (indirekt) in Standardcontainern zu speichern.

3 Material zum aktiven Lernen

Da eine Programmiersprache nur durch aktive Verwendung erlernt werden kann, werden im folgenden Aufgaben zum praktischen Üben vorgestellt. Zunächst wird ein Grundgerüst (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 ihr Grundgerüst von Materialpaket 08_PTRN als Ausgangspunkt und nehmen folgende Änderungen vor:

1. die Basisklasse für Bargeld (Scheine und Münzen) erbt von `HeapObject`
2. verschieben/verteilen Sie sämtlichen Code aus `main()` in die drei Funktionen `setup()`, `simulate()` `tearDown()` und rufen diese in `main()` auf. Die Parameterlisten können Sie frei gestalten
3. fügen Sie vor das `return` in `main()` die Anweisung `HeapObject::assertionsHold();` ein. Die Funktion `main()` enthält also ausschließlich die Aufrufe der oben genannten drei Funktionen, `HeapObject::assertionsHold();` und die `return`-Anweisung
4. verwenden Sie zur Handhabung von Bargeld im Programm raw pointers, die sie mit dem Operator `new` erhalten und die Sie später an `delete` übergeben
5. Bargeld wird von einem Konsumenten zu einem anderen übertragen (Schenkung)
6. ein Konsument kann gelegentlich Bargeld verlieren. Das Objekt soll also gelöscht werden

Stellen Sie sicher, dass die Assertions in der Klasse `HeapObject` nicht anschlagen, das Programm also sauber beendet. Anmerkung: Sie können in `HeapObject.cpp` die Prüfung `assert(ctorCount == newCount);` auskommentieren, falls hier die Kombination von überladenem `new`-Operator und `std::make_shared<>` für Probleme sorgt.

Nehmen Sie nun folgende Änderungen vor:

1. Stellen Sie die Handhabung von Bargeld in ihrem Programm von raw pointer auf `std::`-Smart Pointers um. Verwenden Sie

also `std::shared_ptr<>`, `std::weak_ptr<>`, `std::unique_ptr<>`, `std::make_unique`, und/oder `std::make_shared`. Hinweis: falls Sie einen Container mit `std::unique_ptr<>` verwenden, benötigen Sie `std::move()`, um die Ownership von einem Container zu einem anderen zu übertragen – eine einfache Zuweisung funktioniert nicht, da sonst mehrere Verweise auf einzelne Zielobjekte entstehen könnten.

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. Simulieren Sie einen Bargeldtransport von einer Bank zu einer anderen, bei dem ein Teil der Scheine und Münzen verloren geht.
2. Bei der Einzahlung von Bargeld auf ein Konto werden die Scheine und Münzen vernichtet und der Wert wird dem Guthaben aufgeschlagen.
3. Erstellen Sie einen `vector<>`, der Scheine und Münzen mit zufälligen Werten (Beträgen) enthält (bzw. Verweise darauf). Lassen Sie den `vector` von `std::sort()` sortieren.

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. Gegeben ist ein Klasse `Derived`, welche von `Base` abgeleitet ist. Kann die freistehende Funktion `void foo(Base b)` aufgerufen werden mit einem Objekt der Klasse `Derived`? Was passiert dabei genau?
2. Durch welche Änderung an `foo` wird sichergestellt, dass `foo` die virtuelle Funktion `virtual void bar()` der Klasse `Derived` bei `b` aufrufen kann?
3. Lassen sich polymorphe Objekte in Containern der Standardbibliothek aufbewahren? Begründung!
4. Unter welche Umständen ist ein Objekt, welches durch Slicing entstanden ist, nicht funktionsfähig?
5. Eine Klasse enthält einen Pointer auf ein mit `new` allokiertes Objekt. Woran müssen Sie auf jeden Fall denken?
6. Sie haben eine Klasse/Struktur `C` definiert und dazu einige Operatoren überladen. Sie haben keinen `operator=` und keinen `C::C(const C&)`-

Konstruktor definiert. Lassen sich die Anweisungen `C a; C b(a); C d; d=a;` übersetzen? Begründung!

7. Wäre es geschickt, obige Klasse `C` mit einem Feld auszustatten, welches ein Zeiger ist? Begründung!
8. Was ist der wesentliche Unterschied zwischen `std::shared_ptr<>` und `std::unique_ptr<>`? Was ist die Gemeinsamkeit?
9. Wozu dient `std::weak_ptr<>`?
10. Begründen Sie die im Paket 4_OO_b vorgestellte Unterscheidung von Stackobjekten und Heapobjekten.

4 Nützliche Links

- Object Slicing http://www.bogotobogo.com/cplusplus/slicing.php
- All About Better Software, C++ 11 / C++14 Smart Pointers, Part 1 The shared pointer: https://www.youtube.com/watch?v=_Sk9JT_gTV4
- MS, Intelligente Zeiger: https://msdn.microsoft.com/de-de/library/hh279674.aspx
- Babu Abdulsalam, C++11 Smart Pointers: https://www.codeproject.com/articles/541067/cplusplus11-smart-pointers
- http://www.acodersjourney.com/2016/05/top-10-dumb-mistakes-avoid-c-11-smart-pointers/

5 Literatur

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