Materialpaket 05_OO_b – Object Orientation

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 die Mechanismen zur Konstruktion und Zerstörung von Objekten einsetzen und deren Konsequenzen berücksichtigen.

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

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

- Unterschiede von Wertobjekten und Heapobjekten erläutern und begründen (also die Gründe und Konsequenzen deren Einsatzes darlegen und für gegebene Situationen den geeigneten Datentyp auswählen)
- diese C++-Mechanismen korrekt verwenden:
  - Konstruktoren
  - Destruktoren
  - Initialisierungsreihenfolge (Konstruktoren / Destruktoren; Reihenfolge bzgl. Basisklasse und Member)
  - member/base initializer list
  - operatoren `new` und `delete`
- für Rechner mit beschränkten Ressourcen (z. B. embedded) die geeigneten Allokationsmechanismen identifizieren

2 Konzepte

Im Folgenden werden Objekte in zwei verschiedene Arten eingeteilt, um einen einfacheren und pflegeleichteren Einsatz von C++-Sprachmitteln zu fördern.

2.1 Wertobjekte und Heapobjekte

Unter den möglichen Typen, die in die Kategorie *benutzerdefinierte Datentypen* fallen, lassen sich zwei besonders häufig verwendete ausmachen: *Werttypen* (mit Wertobjekten) und *Objekttypen* (mit Heapobjekten, also mit Objekten im Sinne der objektorientierten Programmierung, welche im Heap leben)

Wertobjekte werden im Wesentlichen für Berechnungen eingesetzt, die denen mit eingebauten Typen ähneln, oder um Datensätze zu verarbeiten. Heapobjekte hingegen agieren in einem Netzwerk aus objektorientiert programmierten Objekten (Heapobjekten).

Unterschiede zwischen Wertobjekten (dort *expanded types* genannt) und Heapobjekten und damit einhegende Konsequenzen werden in [OOSC] Kapitel 8 bzw. 8.7 beschrieben.

Table 1: Gegenüberstellung der einzelnen verschiedenen Eigenschaften von Wertobjekten und Heapobjekten bzw. Werttypen und Objektklassen.

Kategorie	Wertobjekt	Heapobjekt
Verwandte Begriffe	Heapobjekt, Stacktyp, Werttyp, Wertobjekt, Record, `struct`, Struktur, regular type (§ 24.3.1 [TCPL])	Heapobjekt, Objektklasse, `class` Objekttyp

Kategorie	Wertobjekt	Heapobjekt
Zugriffsoperator	`.` (auf member)	`->`
Einschränkungen	dürfen keine Zeiger enthalten	dürfen ausschließlich im heap (free store) erzeugt werden
(De-)/Allokation	durch Compiler (lokale Variablen, temporäre Werte)	durch das Programm/ den Programmierer mit `new` und `delete`
Destruktor	meist unwichtig (compiler-generated reicht)	sehr wichtig; virtual Rule of Three/Zero/Five
dynamischer Speicher	Aufbewahrung in Container oder Rekursion	implizit gegeben
Allokationsaufwand	sehr gering (`+=` auf stack pointer), konstant	mittelhoch (Fragmentierung), nicht konstant
Lebensdauer	solange wie der Kontext (Ausdruck, Block, Funktion, Heapobjekt, BdefTyp, container, ...)	beliebig lang (bis `delete`)
Erreichbarkeit	Variablenname (Bezeichner)	hat keinen Namen; Zugriff per getypter Adresse, welcher ein Name gegeben werden kann
Lebensraum	auf dem call stack bzw. im activation record	im Heap
Adressermittlung	Adressoperator (`&`) E	rgebnis `operator new`
Verwendung der Adresse	nicht möglich bei temporaries; risikoreich bei den anderen (z. B. lokale).	bis `delete`
Operator overloading	viel genutzt	unüblich / problematisch; ggf. in Basisklasse und konkrete Implementierung in virtuellen Methoden von abgeleiteten Klassen.
Vererbung	nicht/selten genutzt	stark genutzt
Virtuelle Methoden	nicht/selten genutzt	stark genutzt
Polymorphismus	ad-hoc und generisch	subtyping
Beziehungen	has-a, is-a (selten wg. slicing)	has-a, is-a, aquaintance, ownership

Kategorie	Wertobjekt	Heapobjekt
Container	lassen sich gut in Containern verwalten	lassen sich nicht in Containern speichern; nur deren Adressen (Stichworte: object slicing, Heap)
Kopien und Zuweisung	automatisch durch compiler (ggf. mittels compiler- generierten copy ctor, Zuweisungsoperator). Kopien sind unkritisch	durch spezielle Methoden (z. B. Idiom virtual constructor)
typische Parameterübergabe	by value, by reference, by const reference	Adresse by value
Kenntlichmachung	`struct`	`class`
Identität	nebensächlich	über Adresse
Äquivalenz	`operator =`	nebensächlich; ggf. virtuelle Methode `.equals()`
Java-Äquivalent	primitive Typen (z. B. `int`)	Instanzen (Ausprägungen) von Klassen
C#-Äquivalent	struct	class

Gemeinsamkeiten von Wertobjekten und Heapobjekten:

- beim Erzeugen bzw. Löschen werden Konstruktoren bzw. Destruktoren vom Compiler aufgerufen
- formal gesehen können beide als `struct` oder `class` realisiert werden
- beide Arten können Wertobjekte enthalten

Kopien sind unkritisch bei Wertobjekten, da diese im Wesentlichen einen Wert darstellen, der keine Identität hat. Eine Unterscheidung zwischen Original und Kopien ist nicht nötig, nicht möglich und nicht sinnvoll. Da Wertobjekte keine Zeiger enthalten, ziehen die naiven Implementierungen der Compiler-generierten Spezialfunktionen Kopierkonstruktor und Zuweisungsoperator (Feld-für-Feld Kopie) keine unerwarteten Ergebnisse nach sich.

Bei Wertobjekten sind enthaltene Felder Wert-gebend – das Verändern eines dieser Felder ändert den Wert. Bei Heapobjekten hingegen stellen die Felder Eigenschaften des Objektes dar. Diese können meist geändert werden, ohne dass sich das Wesen oder die Identität des Heapobjektes ändert. Oft bilden Heapobjekte zur Laufzeit einen oder mehrere Objektbäume (durch die Beziehungen zwischen den einzelnen Objekten).

In Java verhalten sich primitive Typen (`int` etc.) ähnlich wie Wertobjekte und Objekte (Instanzen von Klassen) wie C++-Heapobjekte. In beiden Fällen

gibt es jedoch einen wichtigen Unterschied: in Java werden keine Destruktoren aufgerufen und der (logische) Zeitpunkt der Zerstörung ist nicht vorhersagbar.

C++ erlaubt neben der hier im Kurs eingeführten Trennung von Typen in Heapobjekte und Wertobjekte auch Objekte bzw. deren Benutzung, die Mischformen darstellen. Hierbei entstehen allerdings vielerlei Probleme, auf die erst in späteren Kapiteln eingegangen wird (object slicing, operator overloading, double delete, nicht-erreichbare Objekte, etc. – siehe Materialpaket `10_PITF`).

Der Begriff regular type (§ 24.3.1 [TCPL]) fordert von Instanzen *independence* und *equality*-Eigenschaften. Das heißt, nach einer Zuweisung `a=b` sind `a` und `b` gleich bzgl. des Operators `==` und unabhängig voneinander: Änderungen an dem einen ziehen keine Änderungen an dem anderen nach sich. Instanzen von regular types können kopiert und default-konstruiert werden. Alle eingebauten C++-Typen sind regular types.

Moderner C++-Programmierstil bevorzugt Wertobjekte, da diese sehr effizient sind und sich gut mit der Standardbibliothek vertragen.

Heapobjekte eignen sich insbesondere, wenn ein Programm Entitäten aus der Realwelt modellieren soll und wenn Beziehungen zwischen diesen Entitäten sich nicht in Listen, Tabellen etc. darstellen lassen und sehr dynamisch sind. Vor allem, wenn Subtyping Polymorphism (Vererbung und `virtual`-Methoden) zum Einsatz kommt, sollten Heapobjekte gewählt werden.

Fazit (vereinfacht): verwenden Sie so oft es geht Werttypen, um von Operator Overloading und der automatischen Speicherverwaltung zu profitieren (durch den Compiler und die Standardbibliothek). Verwenden Sie Heapobjekte, um die Vorteile von Subtyping Polymorphism zu nutzen. Vermeiden Sie Mischformen!

2.2 Beziehungen zwischen Klassen und Objekten

Die Sprache C++ bietet mehrere Möglichkeiten, Beziehungen (Assoziationen[2]) zwischen Klassen und Objekten zu modellieren. Beziehungen können verschieden stark sein und spiegeln sich in unterschiedlichen C++-Sprachmitteln wider. Die wichtigsten Beziehungen sind:

- *is-a* (Vererbung): Ein Objekt eine abgeleiteten Klasse ist ein Objekt der Oberklasse (lässt sich also so verwenden)
  - a car is-a vehicle
  - generalization/specialization
- *has-a*, *consists-of* (Komposition):
  - whole-part relationship
  - a car has-a motor, a car consists of a motor
  - der Teil kann nicht ohne das Ganze existieren

[2] http://javapapers.com/oops/association-aggregation-composition-abstraction-generalization-realization-dependency/

  - in C++: expanded Types (Wertobjekte) als Member (ggf. auch (Smart) Pointer auf HeapObjekte, welche dieselbe Lebensdauer haben, wie das umschließende Objekt)
- *acquaintance* (Aggregation): eine schwächere Form der *has-a*-Beziehung. Das umschließende Objekt hat keine *ownership* an dem anderen Objekt.
  - a *knows* b
  - A has B, which belongs to someone else
  - a car has (knows) passengers, but is not responsible for them
  - der Teil kann ohne das Ganze existieren
  - in C++: weak references (STL smart Pointers, pointer to HeapObject without `delete` obligation)

2.3 Konstruktion komplexer Objekte (Konstruktionsreihenfolge)

In dem Fall, in dem der Compiler eine Instanz einer `struct` oder `class` anlegen muss, sorgt er für den Ablauf dieser zwei Vorgänge:

1. Speicher wird angefordert. Im dynamischen Fall (`new`) wird der Speicher von Standardbibliotheken im Zusammenspiel mit den Betriebssystem zur Verfügung gestellt. Im statischen Fall wird dies vom Linker und dem Betriebssystem erledigt.
2. Der frische Speicher wird mittels Compiler-generierter Aufrufe von Konstruktoren initialisiert, so dass sich eine gültige Instanz des zu erzeugenden Typens ergibt.

In dem Fall, in dem der Compiler eine Instanz einer `struct` oder `class` *freigeben* muss, sorgt er für den Ablauf dieser zwei Vorgänge:

1. Destruktoren werden aufgerufen, um Einklinkpunkte für Aufräumarbeiten zu bieten. Der von der zu löschenden Instanz belegte Speicher enthält ggf. Verweise auf Ressourcen (Handles, Pointer, etc.), welche explizit freigegeben werden müssen. Dies geschieht durch Code, den der Programmierer in Destruktoren platziert (z. B. `delete`, `Release...()`).
2. Speicher wird wieder freigegeben. Im dynamischen Fall (`delete`) wird der Speicher von Standardbibliotheken im Zusammenspiel mit den Betriebssystem freigegeben. Im statischen Fall wird dies vom Linker und dem Betriebssystem erledigt.

Die oben angegeben Regeln gelten sowohl für einfache oder komplex zusammengesetzte Objekte. Ebenso ist es unerheblich, ob Vererbung im Spiel ist. In den nicht-trivialen Fällen muss lediglich die Reihenfolge beachtet werden:

- vor Ablauf eines Konstruktors sind member fields konstruiert worden (d.h. die Konstruktoren der Member sind durchlaufen worden)
- vor Ablauf eines Konstrukotrs ist die Basisklasse konstruiert worden (d.h. der Konstruktor der Basisklasse ist durchlaufen worden)

Beispiel:

```
#include <iostream>

struct FooVal {
    FooVal(int initialValue=0);
    ~FooVal();
private:
    int _someValue;
};

class FooBase {
    FooVal _value;
public:
    FooBase();
    virtual ~FooBase();
};

class FooDerived : public FooBase {
public:
    FooDerived();
    ~FooDerived();
};
```

Mit den folgenden Implementierungen:

```
FooVal::FooVal(int initialValue)
: _someValue(initialValue)
{
    std::cout << "FooVal::FooVal()" << std::endl;
}

FooVal::~FooVal(){
    std::cout << "FooVal::~FooVal()" << std::endl;
}

FooBase::FooBase(){
    std::cout << "FooBase::FooBase()" << std::endl;
}

FooBase::~FooBase(){
    std::cout << "FooBase::~FooBase(" << std::endl;
}

FooDerived::FooDerived(){
    std::cout << "FooDerived::FooDerived()" << std::endl;
}
```

```
22
23 FooDerived::~FooDerived(){
24     std::cout << "FooDerived::~FooDerived()" << std::endl;
25 }
```

Mit folgender Hauptfunktion:

```
1  int main(int argc, const char * argv[]) {
2
3      FooBase *obj = new FooDerived();
4
5      delete obj;
6      return 0;
7  }
```

Gibt dies aus:

```
1  FooVal::FooVal()
2  FooBase::FooBase()
3  FooDerived::FooDerived()
4  FooDerived::~FooDerived()
5  FooBase::~FooBase(
6  FooVal::~FooVal()
```

Die oben angegebene Ausgabe illustriert die Konstruktionsreihenfolge: zuerst die Basisklasse, dann die Felder der aktuellen Klasse, dann erst der Konstruktor der aktuellen Klasse. Wobei diese Regel ebenso auf die Basisklasse anzuwenden ist (die Konstruktion beginnt also ganz oben in der Vererbungshierarchie).

Das Zerstören zieht Destruktoraufrufe in umgekehrter Reihenfolge nach sich.

Wäre in der Klasse `FooBase` der Destruktor nicht `virtual` deklariert, so würde der Aufruf von `FooDerived::~FooDerived()` entfallen: der Compiler generiert einen Destruktoraufruf anhand des ihm ersichtlichen Typens. Da dies in diesem Fall ein `FooBase *` ist, wird `FooBase::~FooBase()` aufgerufen. Ist diese member function nicht virtuell, unterbleibt der Aufruf des Destruktors der Klasse des Laufzeittypens (im Beispiel `FooDerived`).

Die `struct FooVal` macht Gebrauch einer *member initializer list* (`: _someValue(initialValue)`). Hier wird das Feld `_someValue` gleich mit einem Wert Konstruiert, statt in zwei Schritten default-konstruiert und dann überschrieben zu werden (mit `_someValue = initialValue` im Konstruktorrumpf).

In der *member initializer list* können nicht nur member initialisiert werden; vielmehr ist dies eine Liste zur Auswahl von Konstruktoren. Es können hier Konstruktoren von Basisklassen angegeben werden, die der Compiler statt des jeweiligen default constructors verwenden soll.

Beispiel: Die Klasse `Foobase` wird erweitert um einen zweiten Konstruktor, der das Feld `_value` mit einem gegebenen `int` per member initialiser list initialisiert:

```
class FooBase {
    FooVal _value;
public:
    FooBase();
    FooBase(int initialValue);
    virtual ~FooBase();
};
```

Implementierung:

```
FooBase::FooBase(int initialValue)
: _value(initialValue)
{
    std::cout << "FooBase::FooBase(int)" << std::endl;
}
```

Soll nun im Konstruktor von `class FooDerived` dafür gesorgt werden, dass nicht der default Konstruktor `FooBase::FooBase()` verwendet wird (z.B. um das `private` geerbte Feld `_value` mit einem Wert zu versehen), sondern der mit `int`-Parameter, kann dies in der *base initialiser list* angegeben werden:

```
FooDerived::FooDerived()
: FooBase(17)
{
    std::cout << "FooDerived::FooDerived()" << std::endl;
}
```

Destruktoren sind besonders wichtig, um sicherzustellen, dass zur Laufzeit angeforderte Ressourcen (Dateien, Datenbankverbindungen, Speicher, etc.) freigegeben werden. Da der Compiler Destruktoren automatisch beim Löschen aufruft, kann die Freigabe nicht vergessen werden.

Beispiel: Eine Klasse `Bar` verwendet ein Objekt der Klasse `FooDerived`:

```
class Bar {
    FooBase *_helperObject;
public:
    Bar();
    ~Bar();
};
```

Mit den Implementierungen:

```
Bar::Bar(){
    _helperObject =  new FooDerived();
}

Bar::~Bar(){
    delete _helperObject;
}
```

Bei Verwendung eines `Bar`-Objektes braucht der Nutzer sich keine Gedanken um das Löschen des Feldes `_helperObject` zu machen. Es wird automatisch erzeugt und gelöscht:

```
1 Bar * bar = new Bar();
2 // ...
3 delete bar; // implies "delete _helperObject";
```

2.4 Allokation von Objekten auf Rechnern mit eingeschränkten Ressourcen

Sollen C++-Programme auf Rechnern zum Einsatz kommen, die nur über stark eingeschränkte CPU- und Speicherressourcen verfügen (Microcontroller, Embedded Systems), so gilt es einige Besonderheiten zu beachten:

- die Laufzeit der Operatoren `new` und `delete` ist womöglich nicht konstant[3]
- automatische Allokation auf dem Stack offenbart zwar nicht das Problem der Fragmentierung – oftmals ist der Stack bei diesen Systemen allerdings zu klein
- es besteht die Notwendigkeit, Objekte an bestimmten Adressen oder in bestimmten Adressbereichen zu allokieren (platzieren)
- es kann Situationen geben, in denen `new` keinen Speicher allokieren kann und `nullptr` zurückgeben muss

C++ bietet Sprachmittel um mit den oben genannten Problemen umgehen zu können. Es ist möglich für benutzerdefinierte Typen (`struct, class`) die Operatoren `new` und `delete` zu überladen. Dadurch ist es möglich, die Speicherverwaltung selbst in die Hand zu nehmen. Ebenso existiert der *placement new*-Operator, der es ermöglicht, Objekte an ganz bestimmten Stellen im Speicher abzulegen, damit sie den Zugriff auf Hardware-Register moderieren oder den nichtflüchtigen (Flash-) Speicher zur Persistierung verwenden.

In [EMA] werden einige Besonderheiten bezüglich der Allokation auf eingebetteten Systeme mitsamt Lösungen vorgestellt; weitere Sachverhalte werden in [HADM] diskutiert. Bjarne Stroustrup geht auf dieses Thema in [PPP] in Kapitel 25 ein.

3 Material zum aktiven Lernen

3.1 Aufgabe: Grundgerüst

Achten Sie darauf, bei den Projekteinstellung den C++14-Standard zu aktivieren (unter Workspace - active project settings - global settings - C++ compiler options

[3] Aufgrund der Fragmentierung des dynamischen Speichers durch wiederholtes verzahntes Anfordern und Freigeben

- ...).

Verwenden Sie im Folgenden diese Werttypen:

```
struct A {
    A(){std::cout << "+A";}
    A(const A&){std::cout << "+A";}
    ~A(){std::cout << "-A";}
};

struct B {
    B(){std::cout << "+B";}
    B(const B&){std::cout << "+B";}
    ~B(){std::cout << "-B";}
};

struct C {
    C(){std::cout << "+C";}
    C(const C&){std::cout << "+C";}
    ~C(){std::cout << "-C";}
};
```

Und diese Objekttypen:

```
class L {
public:
    L(){std::cout << "+L";}
    ~L(){std::cout << "-L";}
};

class M {
public:
    M(){std::cout << "+M";}
    ~M(){std::cout << "-M";}
};

class K {
public:
    K(){std::cout << "+K";}
    ~K(){std::cout << "-K";}
};
```

Verwenden Sie die nachfolgend gegebene Klasse `HeapObject` als Basisklasse für die Objekttypen `K, L, M`. und rufen am Ende von `main()` die statische Methode `HeapObject::assertionsHold()` auf.

```
class HeapObject{
public:
    void*   operator new (size_t size);
```

```
4       HeapObject();
5       virtual ~HeapObject();
6       static bool assertionsHold();
7   protected:
8   private:
9       static int ctorCount;
10      static int dtorCount;
11      static int newCount;
12      //    static void remove(HeapObject *);
13      HeapObject(const HeapObject&) = delete;
14      HeapObject& operator=(const HeapObject&) = delete;
15  };
16
17  int HeapObject::ctorCount = 0;
18  int HeapObject::dtorCount = 0;
19  int HeapObject::newCount  = 0;
20
21  void*   HeapObject::operator new (size_t size){
22      newCount++;
23      return new char[size];
24  }
25
26  HeapObject::HeapObject(){
27      ctorCount++;
28  }
29
30  HeapObject::~HeapObject(){
31      dtorCount++;
32  }
33
34  bool HeapObject::assertionsHold(){
35      assert(ctorCount == newCount);   // all objects have been allocated on heap
36      assert(ctorCount == dtorCount);  // all objects have been deleted
37      return true;
38  }
```

Verwenden Sie die nachfolgend gegebene Klasse `StackObject` als Basisklasse für die Werttypen `A, B, C`.

```
1   struct StackObject {
2   private:
3       void* operator new(size_t size) noexcept {
4           bool noStackObjectOnHeap = false;
5           assert(noStackObjectOnHeap);
6           return nullptr;
7       }
8   };
```

Experimentieren Sie mit Allokation von Instanzen der Typen `A, B, C, K, L, M` auf dem Call Stack und dem Heap.

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.

Hinweis: verwenden Sie die `clang++`-Option `-fno-elide-constructors`, damit Fehler nicht von Compileroptimierungen verdeckt werden.

Modifikationen:

1. erstellen Sie eine Funktion `void pattern2()`, welche Objekte der Werttypen `A, B, C` nutzt, so dass sich bei Aufruf von `pattern2()` die Ausgabe `+B+C+A-A-C-B` ergibt
2. erstellen Sie eine Funktion `void pattern1()`, welche Objekte der Werttypen `A, B, C` nutzt, so dass sich bei Aufruf von `pattern1()` die Ausgabe `+B+C-C+A-A-B` ergibt. Hierbei soll ein Block (`{ /* put statements here */ }`) zum Einsatz kommen.
3. erstellen Sie eine Funktion `void pattern3()`, welche Objekte der Werttypen `A, B, C` nutzt, so dass sich bei Aufruf von `pattern3()` die Ausgabe `+A+A+B-B-A+C-C-A` ergibt. Hierbei soll eine Funktion `B AtoB(A a)` zum Einsatz kommen.
4. Verbinden Sie die Klassen `K, L, M` per Vererbung, dass ein `L * p = new L(); delete p;` die Ausgabe `+K+M+L-L-M-K`
5. ändern Sie vorherigen Code, so dass sich `+K+B+M+L-L-M-B-K` ergibt (ändern Sie nicht die Vererbungshierarchie)
6. Nehmen Sie den Quelltext aus dem vorherigen Materialpaket und verschieben `Shape::_position` sowie `ColoredShape::_color` in den `private`-Bereich
7. Fügen Sie eine Klasse `Sign` hinzu, die eine Schild repräsentiert (also ein Text mit Umrandung). Überlegen Sie, ob und wie Sie Vererbung sinnvoll einsetzen können

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. In welcher Reihenfolge werden die Konstruktoren entlang einer Vererbungslinie ausgeführt?
2. In welcher Reihenfolge werden die Destruktoren entlang einer Vererbungslinie ausgeführt?
3. Welchen Zweck hat die member initializer list?
4. Welchen Zweck hat die base initializer list?
5. Sollte der Destruktor eines Objekttypen `virtual` deklariert werden? Warum Ja/Nein?
6. Werden die Konstruktoren von Felder einer Klasse `X` vor `X::X()` aufgerufen? Warum Ja/Nein?
7. Was passiert, wenn der Operator `new` ausgeführt wird?
8. Wieviel Kopien des Typs `X` werden erzeugt beim Aufruf einer Funktion `X foo(X x)` erstellt? Lassen Sie außer Acht, dass der Compiler Kopien wegoptimieren kann.
9. Was sind die Unterschiede zwischen den Assoziationen *is-a* und *has-a* (im Allgemeinen)?
10. Wie setzen Sie die folgenden Beziehungen in C++ um: *is-a*, *consists-of*, *has-a*, *knows*?
11. Was sind die Unterschiede zwischen den Assoziationen *Aggregation* und *Komposition* (im Allgemeinen und in C++)?
12. Wählen Sie für jeden der folgenden Begriffe aus, ob Sie einen Werttypen oder einen Objekttypen zu dessen Implementierung wählen würden (Begründung): Temperatur, wissenschaftlicher Artikel, Farbton, elektronisches Bauteil.
13. Warum ist die Laufzeit der Operatoren `new` und `delete` womöglich nicht konstant?
14. Welche Probleme können sich bei der automatischen Allokation von Objekten auf dem Stack auf Rechnern mit eingeschränkten Ressourcen ergeben?
15. Wie können Sie im Code zwei Objekte auf Äquivalenz prüfen?
16. Wie können Sie im Code zwei Objekte auf Identität prüfen (Objekte sind per Zeiger gegeben)?
17. Gegeben sei die Klasse `Base` (s.u.). Erstellen Sie eine Klasse `Derived`, deren öffentlicher Konstruktor zwei `int` akzeptiert, die als Initialisierungswerte für `a` und `b` dienen und somit die Variablendefinition `Derived d(1,2);` ermöglicht.

```
1 class Base{
2   int a;
3 public:
4   Base(int aa) {a=aa;}
5 };
```

4 Nützliche Links

- Electronic Arts Standard Template Library (EASTL): https://github.com/electronicarts/EASTL und http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2271.html
- Embedded Template Library (ETL): http://www.etlcpp.com

5 Literatur

- [EMA] DAILEY, AARON: Effective C++ Memory Allocation[4]
- [HADM] Murphy, Niall: How to Allocate Dynamic Memory Safely[5]
- [PPP] Stroustrup, Bjarne: Programming - Principles and Practice using C++
- [TCPL] Stroustrup, Bjarne: The C++ Programming Language, Fourth Edition

[4] http://m.eet.com/media/1171524/f-dailey.pdf

[5] https://barrgroup.com/Embedded-Systems/How-To/Malloc-Free-Dynamic-Memory-Allocation