C/C++ Materialpaket (Level AB)
05c_OO_CYCL – Object Orientation (Lifecycle)

Prof. Dr. Carsten Link

Zusammenfassung

Abbildung 1: Der Lebenszyklus von Werten und Entitäten

Inhaltsverzeichnis

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten und 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]:

- 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 Konstruktion und Destruktion einfacher Objekte

Die Konstruktion und Destruktion einfacher Objekte wird an diesem Beispiel erläutert:

```
1  struct S {
2    int a;
3    int b;
4    S();  // default ctor
5    S(int aa, int bb);
6    ~S();
7  };
8
9  class C {
10   int x;
11   int y;
12 public:
13   C();  // default ctor
14   C(int xx, int yy);
```

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

```
15   ~C();
16 };
17
18 int main(){
19   S s1;
20   S s2(1,2);
21   C* c1 =  new C(3,4);
22   C* c2 = nullptr;
23   delete c1;
24   return s1.a + s2.a;
25 }
```

Werte vom Typ `S` und vomt Typ `C` werden durch den Compiler angelegt. Dazu wird Speicher bereitgestellt, der im zweiten Schritt mit dem Aufruf eines Konstruktors initialisiert wird. Unterschiedlich ist jedoch der jeweilige Auslöser für das Anlegen eines Wertes oder Objekts.

Das Anlegen von Ausprägungen von Werttypen (wie beispielsweise lokale Variablen `s1` und `s2`) wird automatisch durch den Programmablauf ausgelöst. Vom Compiler wird dafür gesorgt, dass Speicherplatz für den Wert vorhanden ist und ein Konstruktor aufgerufen wird, damit dieser Speicher initialisiert wird. Das Löschen eines Wertes von `S` wird ebenso durch den Compiler ausgelöst (beispielsweise am Ende einer Funktion werden die lokalen Variablen gelöscht).

Das Anlegenen einer Instanz des Entitätstypens `C` wird explizit durch das Programm mit `new` ausgelöst. Auch hier wird ein Konstruktor aufgerufen, dem ggf. Parameter übergeben werden können (wie bei `c1`). Das Löschen einer Instanz von `C` wird typischerweise explizit durch das Programm mit `delete` ausgelöst.

Im Beispiel oben existieren `s1` und `s2` länger als das Objekt, auf das `c1` verweist. Da `s1` und `s2` für die Berechnung der Rückgabewertes nötig sind, kann der Compiler die beiden Werte nicht vorher löschen.

2.2 Konstruktionsreihenfolge bei komplexen Objekten

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:

1. vor Ablauf eines Konstruktors sind member fields konstruiert worden (d.h. die Konstruktoren der Member sind durchlaufen worden)
2. vor Ablauf eines Konstruktors ist die Basisklasse konstruiert worden (d.h. der Konstruktor der Basisklasse ist durchlaufen worden); wobei dort Regel 1 eingehalten wurde

Die sich ergebende Konstruktionsreihenfolge ist also: zuerst die Member der Basisklasse, Basisklassenkonstruktor, Member der abgeleiteten Klasse, Konstruktor der abgeleiteten Klasse; wie im folgenden Beispiel zu sehen:

```
#include <iostream>

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

class BaseEntity {
    ValueType _value;
public:
    BaseEntity();
    virtual ~BaseEntity();
};

class DerivedEntity : public BaseEntity {
public:
    DerivedEntity();
    ~DerivedEntity();
};
```

Mit den folgenden Implementierungen:

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

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

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

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

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

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

Mit folgender Hauptfunktion:

```
int main(int argc, const char * argv[]) {

    BaseEntity *obj = new DerivedEntity();

    delete obj;
    return 0;
}
```

Gibt dies aus:

```
ValueType::ValueType()
BaseEntity::BaseEntity()
DerivedEntity::DerivedEntity()
DerivedEntity::~DerivedEntity()
BaseEntity::~BaseEntity(
ValueType::~ValueType()
```

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 bei den Membern der Klasse ganz oben in der Vererbungshierarchie).

Das Zerstören zieht Destruktoraufrufe in umgekehrter Reihenfolge nach sich.

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

Die `struct ValueType` 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 `BaseEntity` wird erweitert um einen zweiten Konstruktor, der das Feld `_value` mit einem gegebenen `int` per member initialiser list initialisiert:

```
1 class BaseEntity {
2     ValueType _value;
3 public:
4     BaseEntity();
5     BaseEntity(int initialValue);
6     virtual ~BaseEntity();
7 };
```

Implementierung:

```
1 BaseEntity::BaseEntity(int initialValue)
2 : _value(initialValue)
3 {
4     std::cout << "BaseEntity::BaseEntity(int)" << std::endl;
5 }
```

Soll nun im Konstruktor von `class DerivedEntity` dafür gesorgt werden, dass nicht der default Konstruktor `BaseEntity::BaseEntity()` 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:

```
1 DerivedEntity::DerivedEntity()
2 : BaseEntity(17)
```

```
{
    std::cout << "DerivedEntity::DerivedEntity()" << 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 `DerivedEntity`:

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

Mit den Implementierungen:

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

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:

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

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

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);
    HeapObject();
    virtual ~HeapObject();
    static bool assertionsHold();
protected:
private:
    static int ctorCount;
    static int dtorCount;
    static int newCount;
    //    static void remove(HeapObject *);
    HeapObject(const HeapObject&) = delete;
    HeapObject& operator=(const HeapObject&) = delete;
};

int HeapObject::ctorCount = 0;
int HeapObject::dtorCount = 0;
int HeapObject::newCount  = 0;

void*   HeapObject::operator new (size_t size){
    newCount++;
    return new char[size];
}

HeapObject::HeapObject(){
    ctorCount++;
}

HeapObject::~HeapObject(){
    dtorCount++;
}

bool HeapObject::assertionsHold(){
    assert(ctorCount == newCount);   // all objects have been allocated on heap
    assert(ctorCount == dtorCount);  // all objects have been deleted
    return true;
}
```

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

```
struct StackObject {
private:
    void* operator new(size_t size) noexcept {
        bool noStackObjectOnHeap = false;
        assert(noStackObjectOnHeap);
        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

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).

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 (`{ ... }`) zum Einsatz kommen.
3. 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`
4. ändern Sie vorherigen Code, so dass sich `+K+B+M+L-L-M-B-K` ergibt (ändern Sie nicht die Vererbungshierarchie)
5. Nehmen Sie den Quelltext aus dem vorherigen Materialpaket und verschieben `Shape::_position` sowie `ColoredShape::_color` in den `private`-Bereich der jeweiligen Klassen. Stellen Sie nun sicher, dass keine Fehlermeldungen oder Warnungen des Compilers auftreten und die Objekte effizient initialisiert werden

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 X()` ausgeführt wird?
8. Warum ist die Laufzeit der Operatoren `new` und `delete` womöglich nicht konstant?
9. Freiwillig: Zeigen Sie auf, inwiefern das Piktogram auf der Titelseite dieses Materialpaketes den Inhalt zusammengefasst darstellt.
10. 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[2]
- [HADM] Murphy, Niall: How to Allocate Dynamic Memory Safely[3]

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

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

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