C/C++ Materialpaket (Level C)
05b_OO_ENTVAL – Object Orientation (Entity Types vs. Value Types)

Prof. Dr. Carsten Link

Zusammenfassung

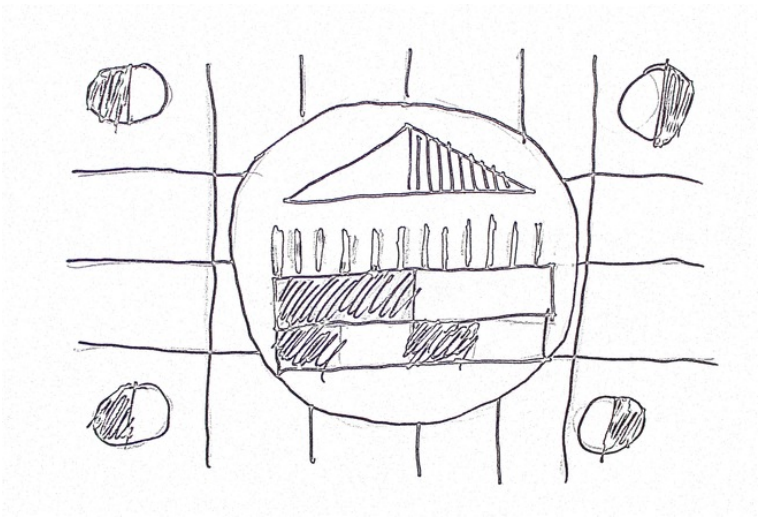

Abbildung 1: Testbild

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]:

- Unterschiede von Wertobjekten und Eintitätsobjekten erläutern und begründen (also die Gründe und Konsequenzen deren Einsatzes darlegen und für gegebene Situationen den geeigneten Datentyp auswählen)
- neben Vererbung auch die Beziehnungsarten Komposition und Aggregation einsetzen

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 Entitätsobjekte

Unter den möglichen Typen, die in die Kategorie *benutzerdefinierte Datentypen* fallen, lassen sich zwei besonders häufig verwendete ausmachen: *Werttypen* (mit Wertobjekten) und *Entitätstypen* (bzw. *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. Entitätsobjekte hingegen agieren in einem Netzwerk aus objektorientiert programmierten Objekten (Entitätsobjekten).

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

Beispiele für Werttypen aus diesem Kurs sind: `Color`, `RationalNumber`, `PascalString`.

Beispiele für Entitätstypen aus diesem Kurs sind: `Circle`, `Rectangle`, `Truck`.

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

Tabelle 1: Gegenüberstellung der einzelnen verschiedenen Eigenschaften von Wertobjekten und Eintitätsobjekten bzw. Werttypen und Entitätsklassen.

Kategorie	Werte	Entitäten
Verwandte Begriffe	Stacktyp, Werttyp, Wertobjekt, Record, `struct`, Struktur, regular type (§ 24.3.1 [TCPL])	Heapobjekt, Objektklasse, `class` Objekttyp
Kenntlichmachung in diesem Kurs	`struct`	`class`
Java-Äquivalent	primitive Typen (z. B. `int`)	Instanzen (Ausprägungen) von Klassen
Zugriffsoperator	`.` (auf member)	`->`
Einschränkungen	dürfen keine Zeiger enthalten	dürfen ausschließlich im heap (free store) erzeugt werden
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
Identität	nebensächlich	über Adresse
Äquivalenz	`operator ==`	nebensächlich; ggf. virtuelle Methode `.equals()`
Polymorphismus	ad-hoc und generisch	subtyping
(De-)/Allokation	durch Compiler (lokale Variablen, temporäre Werte)	durch das Programm/ den Programmierer mit `new` und `delete`
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
Destruktor	meist unwichtig (compiler-generated reicht)	sehr wichtig; virtual Rule of Three/Zero/Five

Kategorie	Werte	Entitäten
dynamischer Speicher	Aufbewahrung in Container oder Rekursion	implizit gegeben
Allokationsaufwand	sehr gering (`+=` auf stack pointer), konstant	mittelhoch (Fragmentierung), nicht konstant
Lebensraum	auf dem call stack bzw. im activation record	im Heap
Adressermittlung	Adressoperator (`&`)	Ergebnis `operator new`
Verwendung der Adresse	nicht möglich bei temporaries; risikoreich bei den anderen (z. B. lokale).	bis `delete`
Beziehungen	has-a, is-a (selten wg. slicing)	has-a, is-a, aquaintance, ownership
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
C#-Äquivalent	struct	class

Gemeinsamkeiten von Wertobjekten und Entitätsobjekten:

- 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 Entitätsobjekten hingegen stellen die Felder

Eigenschaften des Objektes dar. Diese können meist geändert werden, ohne dass sich das Wesen oder die Identität des Entitätsobjektes ändert. Oft bilden Entitätsobjekte 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 Java-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 Entitätsobjekte, 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[^association]) 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
  - 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)

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

(Level C) *Strings mit copy-on-write:* Bauen Sie die Klasse `PascalStr` so um, dass das `char[]` nicht vollständig im Objekt liegt, sondern nur ein Zeiger auf ein mit `new` allokoiertes. Um zusätzliche Effizienz zu gewinnen, wird beim Kopieren von `PascalStr_cow`-Objekten nicht alles kopiert, sondern mehrere Objekte teilen sich das `char[]`. Wird ein `PascalStr_cow`-Objekt verändert, so muss bei der jeweiligen Methode darauf geachtet werden, dass ein eigenes `char[]` angelegt wird.

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. (Level C) Erstellen Sie Testprozeduren, die ordnungsgemäße Funktion zeigen.
2. (Level C) Erstellen Sie Testprozeduren, die zeigen, dass `PascalStr_cow`-Objekte effizienter sind, als `PascalStr`-Objekte.

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. 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.
2. Wie können Sie im Code zwei Objekte auf Äquivalenz prüfen? Unterscheiden Sie Wertobjekte und Entitätsobjekte.
3. Wie können Sie im Code zwei Objekte auf Identität prüfen (Objekte sind per Zeiger gegeben)? Unterscheiden Sie Wertobjekte und Entitätsobjekte.
4. Was sind die Unterschiede zwischen den Assoziationen *is-a* und *has-a* (im Allgemeinen)?
5. Wie setzen Sie die folgenden Beziehungen in C++ um: *is-a*, *consists-of*, *has-a*, *knows*?
6. Was sind die Unterschiede zwischen den Assoziationen *Aggregation* und *Komposition* (im Allgemeinen und in C++)?
7. Freiwillig: Zeigen Sie auf, inwiefern das Piktogram auf der Titelseite dieses Materialpaketes den Inhalt zusammengefasst darstellt.

4 Nützliche Links

- Java Papers; Association, Aggregation, Composition, Abstraction, Generalization, Realization, Dependency: http://javapapers.com/oops/association-aggregation-composition-abstraction-generalization-realization-dependency/

5 Literatur

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