UMIT
the health & life sciences university

Programmierung, Algorithmen und Datenstrukturen 2: Grundlagen der objektorientierten Programmierung Grundlegende Datenstrukturen und Algorithmen SS 2019

Martin Welk
Elias Tappeiner

Institut für Biomedizinische Bildanalyse
EWZ, Hall, Raum G2.020 (MW)/Raum G2.019 (ET)
E-Mail: martin.welk@umit.at, elias.tappeiner@umit.at

UMIT
the health & life sciences university

Umfang und Inhalt

- Vorlesung für Studierende im Bachelorstudiengang Mechatronik
  (VU, 2 SWS / 2,5 ECTS, Pflichtbereich)
- **Ziele:**
  - Erlernen grundlegender Konzepte der objektorientierten Programmierung
  - Umsetzung in der Programmiersprache C++
- **Inhalte:**
  - Klassendefinition in C++
  - Grundlegende Datenstrukturen und Algorithmen
  - Vertiefung des strukturierten Programmierens in C++ mit Datenstrukturen und Algorithmen der C++-Standardbibliothek

UMIT
the health & life sciences university

Gruppen/Termine

Im Sommersemester 2019 drei Gruppen:

- Hall: Termine 1, 3, 5, 7 getrennt für zwei Gruppen
  (Gruppe 1 montags, Gruppe 3 mittwochs)
- Hall: Termine 2, 4, 6 gemeinsam für beide Gruppen (freitags/einmal samstags)
- Lienz (Termine freitags) für 1. und 3. Semester

Genaue Termine: nach Bekanntgabe durch das Studienmanagement und in Moodle

Abweichungen von den zu Semesterbeginn bekannt gegebenen Terminen sind möglich und werden rechtzeitig in Moodle und/oder per Aussendung an die UMIT-Mail-Adresse angekündigt. Es ist in Ihrer Verantwortung, Moodle und UMIT-Mail regelmäßig abzurufen.

UMIT
the health & life sciences university

Lehrveranstaltungsform: Präsenztermine

Die Präsenztermine werden

Vorlesung (VU) mit integrierten praktischen Übungen

durchgeführt.

- Beamerpräsentation (Einführung/Zusammenfassung; theoretische Blöcke zu Algorithmen/Datenstrukturen)
- Verstehen von Beispielprogrammen in Gruppenarbeit
- Selbstständige Bearbeitung von Programmieraufgaben, Beispiellösungen werden im Nachgang bereit gestellt

UMIT
the health & life sciences university

Lehrveranstaltungsform: Hausarbeit

- **Ergänzendes Selbststudium** zur Vertiefung und Erweiterung, Vorbereitung für Folgetermin
- **Hausübungen**
  - Fortführung der Programmierbeispiele aus der Präsenzveranstaltung, zusätzliche Programmieraufgaben zur selbstständigen Bearbeitung, Bleistift-und-Papier-Übungen zum vertieften theoretischen Verständnis
  - Nach Abgabe gegenseitige Beurteilung (für Programmierübungen)
  - Anregung zur selbstständigen Vertiefung des Stoffs
  - Regelmäßige Übungsbearbeitung (einschließlich Beurteilungsaufträge) ist Teilleistung zum Bestehen

UMIT
the health & life sciences university

Hinweise zum Ablauf der Hausübungen

- **Schritt 1:** Selbstständige Bearbeitung der Programmieraufgaben in Gruppen bis zu 3 Studierenden mit anschließender Abgabe
  - Die Gruppenarbeit wird von **einer/m** beteiligten Studierenden in Moodle abgegeben.
    Die Namen **aller** Beteiligten (maximal 3) müssen bei der Abgabe angegeben sein, sonst können keine Punkte vergeben werden!
  - Die Abgabefrist ist ausnahmslos einzuhalten, Nachreichungen sind aus organisatorischen Gründen nicht möglich.
- **Schritt 2:** Gegenseitige Beurteilung und Kommentierung durch Studierende
  - Jede/r Studierende, die/der an einer Gruppenabgabe mitgewirkt hat, bekommt eine Abgabe zur Beurteilung zugeordnet.
  - Wer bei einer Aufgabe nicht an einer abgegebenen Bearbeitung beteiligt war, kann bei dieser Aufgabe auch keine Beurteilung zugeordnet bekommen (und damit keine Beurteilungspunkte erwerben).
  - Die Beurteilungsleistung ist eine Einzelleistung, keine Gruppenarbeit!
  - Auch hier ist die Bearbeitungsfrist ausnahmslos einzuhalten.

UMIT
the health & life sciences university

Leistungsbewertung – kursbegleitende Prüfungsleistung

- Die kursbegleitende Prüfungsleistung wird durch die Hausübungen mit gegenseitiger Beurteilung erbracht.
- Ein Teil der Abgaben und Beurteilungen wird durch Dozenten/Studienassistenten bewertet
- Selbstständigkeit der Bearbeitungen: Erhebliche Übereinstimmung zwischen Abgaben mehrerer Gruppen wird mit Punktabzügen geahndet!
- Die erreichbare Gesamtpunktzahl setzt sich zusammen aus
  - ca. $25\,\%$: Bewertungen der abgegebenen Programmierleistung durch andere Studierende
  - ca. $10\,\%$: Ausführung der zugeteilten Beurteilungsaufträge
  - ca. $25\,\%$: Bewertungen ausgewählter Abgaben durch Dozenten/ Studienassistenten (unter Berücksichtigung der Gesamtzahl von Abgaben)
  - ca. $25\,\%$: Bewertungen ausgewählter Beurteilungsleistungen nach Kontrolle durch Dozenten/Studienassistenten
  - ca. $15\,\%$: Bewertungen der Theorieaufgaben
- Mit $50\,\%$ der Gesamtpunktzahl ist die kursbegleitende Prüfungsleistung positiv bewertet. Dies ist Voraussetzung zur Klausurteilnahme.

UMIT
the health & life sciences university

Leistungsbewertung – Abschlussprüfung

- Der Kurs wird mit einer Klausur abgeschlossen. Hierfür werden zwei Termine angeboten:
  - 15.07.2019, 11.00–12.30 Uhr
  - 30.09.2019, 13.00–14.30 Uhr (Ersatztermin)
- Die positiv absolvierte kursbegleitende Prüfungsleistung ist Voraussetzung zur Klausurteilnahme.
- Die Endnote ergibt sich grundsätzlich aus der Klausur.
- Bei deutlich über die Mindestvoraussetzung hinausgehenden Übungsleistungen (mindestens $75\,\%$ der Übungspunkte) können Bonuspunkte zur Aufwertung der Endnote erteilt werden (bis zu einer Notenstufe Aufbesserung).

UMIT
the health & life sciences university

Leistungsbewertung – Bemerkungen

- Wird die Klausurzulassung nicht erreicht oder trotz erreichter Klausurzulassung die Klausur nicht bestanden, so ist nach den studienrechtlichen Bestimmungen der Kurs (VU) **als Ganzes** im Folgejahr zu wiederholen.
- Insbesondere kann auch eine positive kursbegleitende Prüfungsleistung **nicht** ins Folgejahr „mitgenommen“ werden.
- Es gibt maximal drei Antritte; der gesamte Kurs mit beiden Klausurterminen zählt als ein Antritt.

UMIT
the health & life sciences university

Unterlagen und Materialien

- Zum Download bzw. Nachlesen auf `http://moodle.umit.at`,
  - Folien (nach der Veranstaltung)
  - Beispiele
  - Aufgaben, Übungen, Arbeitsaufträge
  - Aktuelle Informationen

UMIT
the health & life sciences university

Unterlagen und Materialien

- Literatur zur Programmierung
  - S. B. Lippman, J. Lajoie, B. E. Moo: C++ Primer. Fifth Edition, Addison-Wesley, 2013, ISBN 978-0-321-71411-4
  - T. Will: C++ – Das umfassende Handbuch. Rheinwerk, 2018, ISBN 978-3836243605
  - . . .
- Literatur zu Algorithmen und Datenstrukturen
  - K. Mehlhorn, P. Sanders: Algorithms and Data Structures. The Basic Toolbox. Springer, 2008.
  - T. Ottman, P. Widmayer. Algorithmen und Datenstrukturen. Spektrum, 4. Aufl. 2002.
  - T. H. Cormen, C. E. Leiserson, R. Rivest, C. Stein: Algorithmen – eine Einführung. Oldenbourg, 2004.

UMIT
the health & life sciences university

Vorlesung 1:
Klassen und Objekte

- Grundlagen der objektorientierten Programmierung
- Klassen, Methoden, Instanzen
- Überladen und Signatur
- Konstruktoren

UMIT
the health & life sciences university

Prozedural versus objektorientiert

- **Prozedurale Programmierung:**
  - Trennt Daten von Methoden
  - Korrektheit und Kompatibilität von Daten und Funktionen muss durch Programmierer/in sicher gestellt werden
  - Werden Daten(strukturen) geändert, so müssen alle abhängigen Funktionen geändert werden und umgekehrt
- **Objektorientierte Programmierung:**
  - Fasst Datenstrukturen und zugehörige Funktionen in Objekten zusammen
  - Ermöglicht Abstraktion von Objekttypen in Klassenbeschreibungen und davon abhängige Konzepte, z. B. Vererbung
  - **Weniger fehleranfällig, besser wiederverwendbar, einfachere Wartung**

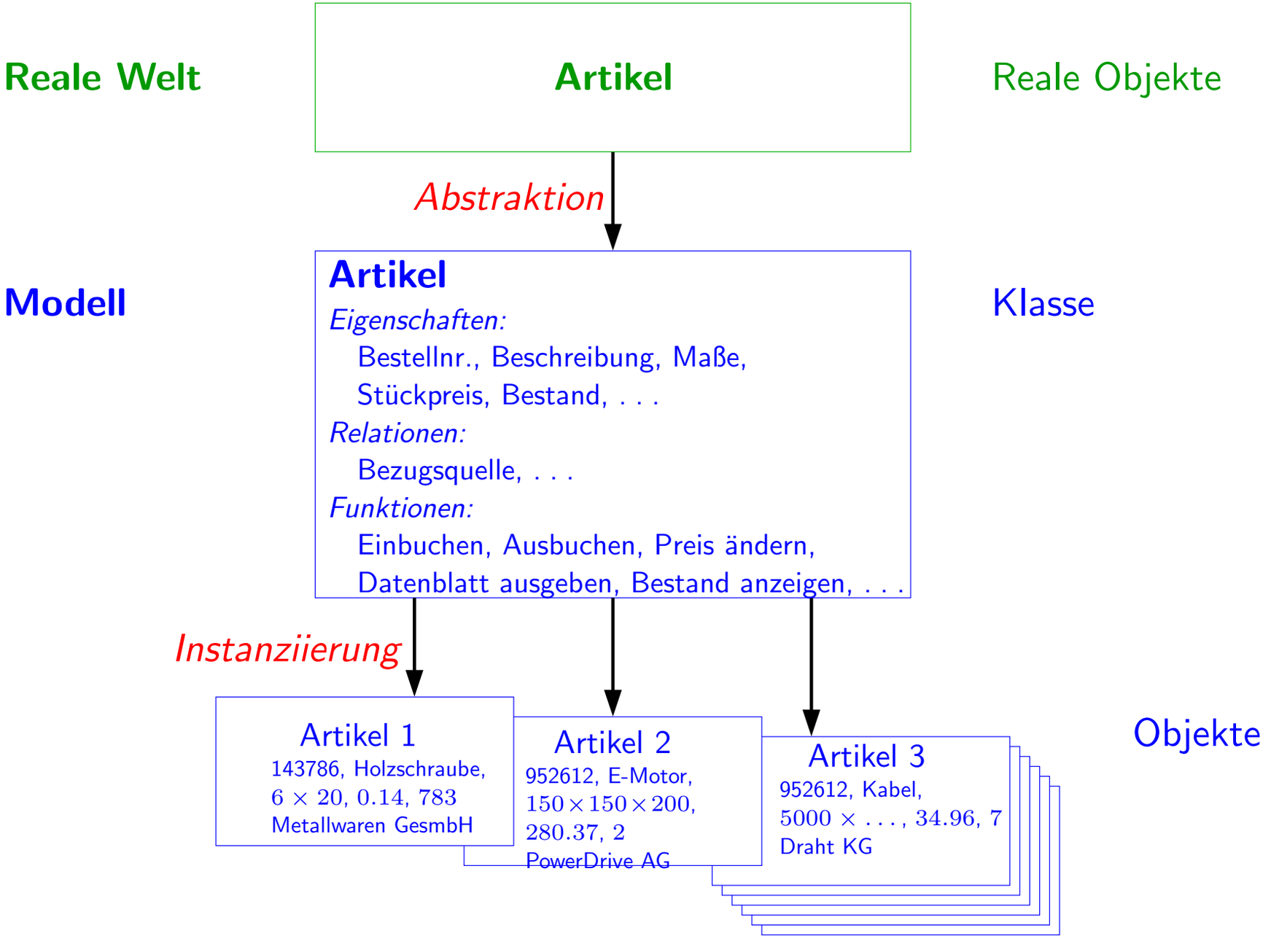
UMIT
the health & life sciences university

Klassen und Objekte

- Hauptkomponente, die C zu C++ macht
- Grundlegende Konzepte: Datenabstraktion, Datenkapselung und Vererbung
- Effizientere Programmierung, bessere Erweiterbarkeit und Wartbarkeit, damit geringere Fehleranfälligkeit
- Entscheidender Schritt in der objektorientierten Programmierung: *Modellierung* eines realen Problems in einer Klassenabstraktion

UMIT
the health & life sciences university

Beispiel eines Objektmodells: Sortimentsartikel für ein Lagerhaltungssystem



UMIT
the health & life sciences university

Datenabstraktion

- Unter **Abstraktion** versteht man die Schaffung einer spezifizierten *Schnittstelle (Interface)* zu den Daten.
- Die Schnittstelle spezifiziert, auf welche Weise die Daten *angesprochen* werden und damit, in welcher Weise sie „nach außen“ sichtbar sind

UMIT
the health & life sciences university

Datenabstraktion

- Die Schnittstelle ist der (einzige) Weg, auf dem die übrigen Programmteile mit dem Objekt kommunizieren, das heißt Daten abfragen und ändern können.
- *Beispiel:* `string`
  - Schnittstelle umfasst Indexzugriff, Methoden `at`, `length`, `substr`, `insert`, den Operator + etc.
  - Zugriffe auf den Dateninhalt der Zeichenkette geschehen ausschließlich über diese Methoden (und Operatoren)
- *Beispiel:* `vector<int>`
  - Schnittstelle umfasst Indexzugriff, Methoden `at`, `size`, `push_back`, `pop_back`, `clear` etc.
  - Zugriffe auf die Elemente des Vektors geschehen ausschließlich über diese Methoden
- *Beispiel:* `ofstream`
  - Schnittstelle umfasst Methoden `open`, `close`, `put`, `setwidth`, den Operator `<<` etc.
  - Zugriffe auf die Datei geschehen ausschließlich über diese Methoden

UMIT
the health & life sciences university

Datenkapselung

- Unter **Datenkapselung** versteht man, dass die Daten abgeschirmt sind – wie sie tatsächlich im Speicher repräsentiert werden, muss nach außen nicht sichtbar sein.

  Beispiel: `vector<int>`

  - Die interne Repräsentation der Daten (z. B. Allokation von Speicherbereichen, Umkopieren der Daten bei Vergrößerung/Verkleinerung des Feldes) ist für die Benutzung des `vector<int>` unerheblich.
    (Wir werden in der Vorlesung „Algorithmen und Datenstrukturen“ Möglichkeiten besprechen, wie ein solches „dynamisches (in der Größe veränderliches) Array“ realisiert werden kann.)

  Beispiel: `string`

  - Wie beim Vektor-Container sind beispielsweise Speicherallokation und interne Umkopiervorgänge nach außen unsichtbar. Auf welche Weise bei `length` die Größe ermittelt wird, ist ebenfalls nicht sichtbar.

  Beispiel: `ofstream`

  - Der Zugriff auf das Dateisystem, ein allfälliger Ausgabepuffer, die vom System vergebene Posix-Dateinummer (Handle) usw. sind nach außen nicht sichtbar.

UMIT
the health & life sciences university

Definition von Klassen in C++

- Schlüsselwort `class` gefolgt vom Namen und einem Deklarationsblock

```
class Demo
{
  private:
    int secret; // Datenfeld (Beispiel)
    ... // nur intern sichtbare Deklarationen
  public:
    int get_secret (); // Methode (Beispiel)
    int compute_something (); // Methode (Beispiel)
    ... // nach aussen sichtbare Deklarationen
};
```

Hinweis: Unbedingt auf den Strichpunkt am Ende der Deklaration achten!

- Der Deklarationsblock enthält Deklarationen von *Member-Variablen* (Datenelementen) und *Methoden* (der Klasse zugeordneten Funktionen).
- Mit den Labels `private:` und `public:` werden Elemente (Daten, Methoden) in nur intern sichtbare und öffentlich zugängliche unterschieden.

  Die Labels können beliebig oft und abwechselnd benutzt werden. Default ist `private`.

UMIT
the health & life sciences university

Definition von Klassen in C++

- Bei der Implementation der im `class`-Block deklarierten Methoden außerhalb dieses Blocks wird ihnen der Klassenname mit `::` vorangestellt, also

```
int Demo::get_secret ()
{
  ...
}

int Demo::compute_something ()
{
  ...
}
```

UMIT
the health & life sciences university

Selbstständiges Üben: Quellcode verstehen

→`2101-calendar.cpp`

- Legen Sie ein neues **C++**-Konsolenprojekt mit dem Quelltext aus der (in Moodle bereit gestellten) Datei `2101-calendar.cpp` an
- Bringen Sie dieses Programm zum Laufen
- Erarbeiten Sie sich anhand dieses Programms die darin vorkommenden neuen C++-Sprachelemente

UMIT
the health & life sciences university

Selbstständiges Üben: Quellcode verstehen

→2101-calendar.cpp

```
// ===== Declaration of an object class for calendar dates =====
class CalendarDate {
  private:
    unsigned int day;
    unsigned int month;
    unsigned int year;
  public:
    bool set (unsigned int day_in, unsigned int month_in,
              unsigned int year_in);
    unsigned int get_day ();
    unsigned int get_month ();
    unsigned int get_year ();
    string get_formatted ();
    void set_from_userinput (string prompt = "Datum: ");
    int days_after_1900 ();
};
```

UMIT
the health & life sciences university

Selbstständiges Üben: Quellcode verstehen

→`2101-calendar.cpp`

- Eine Klasse wird ähnlich definiert wie ein `struct`, jedoch mit dem Schlüsselwort `class`
- Neu ist, dass außer Datenelementen (Membervariablen) auch Funktionen Bestandteil einer Klasse sein können; diese heißen dann *Methoden*
- Die Methoden werden in der Klassendeklaration normalerweise nur deklariert
- Die Definition (Implementation) der Methoden erfolgt dann gesondert außerhalb der Klassendeklaration
- Die Definition einer Klassenmethode erfolgt durch Voranstellen des Klassennamens und des *Bereichsoperators* `::`, z. B. `void CalendarDate::get_formatted() {...}`).
- Methodennamen sind daher klassenspezifisch.

UMIT
the health & life sciences university

Instanziierung

- Eine Klassendefinition ist vergleichbar mit einer Datentypdefinition.

  Häufig verwendete Konvention für selbst deklarierte Klassen:
  Klassennamen beginnen mit Großbuchstaben.

- Objekte vom Typ der Klasse heißen **Instanzen.**

  Instanzen werden definiert wie schon von Variablendeklarationen bekannt:

  Klassenname Objektname1 [, *Objektname2, . . .*];

  Beispiel:

  ```
  CalendarDate start, stop;
  ```

  Damit sind `start` und `stop` Instanzen der Klasse `CalendarDate`.

UMIT
the health & life sciences university

Verwendung von Klassenmethoden und -variablen

- Grundsätzlich können außerhalb der Klasse nur die Public-Methoden und -Variablen verwendet werden.

- Diese werden über ein instanziiertes Objekt mittels Punktoperator aufgerufen und wirken dann auf dieses Objekt:

```
CalendarDate start; // in main()
start.set_from_userinput();
cout << start.get_formatted();
```

Aufruf der Methoden für das Objekt `start`, die Methoden arbeiten also mit den Datenfeldern dieses Objekts.

Analog:

```
string text = "Lienz";
text.insert (text.end(), " -- Hall in Tirol");
cout << text.substr (9, 4); // -> Ausgabe: Hall
```

Aufruf der Methoden für das Objekt `text`, die Methoden arbeiten also mit den Datenfeldern dieses Objekts.

UMIT
the health & life sciences university

Verwendung von Klassenmethoden und -variablen

- Klassenelemente können direkt, also ohne Objektnamen und Punktoperator, nur innerhalb der zugehörigen Klasse aufgerufen werden.

  Der Aufruf einer Membervariable in einer Klassenmethode greift auf die entsprechende Variable in der Instanz zu, mit der die Methode aufgerufen wird.

  - `if (set (day_user, month_user, year_user)) break;`
    in `CalendarDate::set_from_userinput ()` ruft `CalendarDate::set (...)` auf, und die Methode `CalendarDate::set (...)` arbeitet mit den Daten derselben Instanz, für die `CalendarDate::set_from_userinput ()` aufgerufen wurde.
  - `return day;`
    in `CalendarDate::get_day ()` gibt das Datenfeld `day` der Instanz zurück, für die `CalendarDate::get_day ()` aufgerufen wurde

UMIT
the health & life sciences university

Verwendung von Klassenmethoden und -variablen

- Private Methoden und Membervariablen einer Klasse sind außerhalb der Klasse nicht sichtbar.

  Wird in der `main`-Funktion ein Zugriff mittels `start.year` auf das „geheime“ Datenfeld versucht, so tritt ein Compilerfehler auf.

- *Empfehlung:* Datenelemente sollten privat sein, der Zugriff sollte nur mittels Methoden erfolgen.

- *Es gibt außer `private` und `public` noch die Zugriffsklasse `protected`. So deklarierte Elemente sind in der Klasse selbst und in davon abeleiteten Klassen zugänglich.*

UMIT
the health & life sciences university

Verwendung von Klassenmethoden und -variablen

- *Typisches und empfehlenswertes Vorgehen:* Membervariablen sind privat.

  Alle Zugriffe sollen über Methoden erfolgen.

- Sollen die Werte der Membervariablen direkt öffentlich abgefragt werden können, so benutzt man *Getter-Methoden*

  Im Beispielprogramm zB `CalendarDate::get_day ()`

  Welche Getter-Methoden gibt es noch im Beispiel?

- Das Setzen der Werte erfolgt über *Setter-Methoden*; dabei sollte stets geprüft werden, ob die übergebenen Werte zulässig sind. Damit kann erreicht werden, dass das Objekt nur „sinnvolle“ Werte enthalten kann

  Im Beispiel: `CalendarDate::set (...)`

  Welche Setter-Methoden gibt es noch im Beispiel?

UMIT
the health & life sciences university

Vergleich von `class` und `struct`

- Eine Klasse (`class`) ist offenbar einer Verbundvariablen (`struct`), wie wir sie aus C kennen, sehr ähnlich.
- Im Unterschied zu einem C-`struct` kann die C++-Klasse Methoden enthalten.
- Im Unterschied zu einem C-`struct` darf die C++-Klasse direkt über ihren Namen (ohne den Vorsatz `class`) angesprochen werden.
- Im Unterschied zu einem C-`struct` sind bei einer C++-Klasse alle Datenelemente und Methoden, sofern nicht anders deklariert, privat.

In C++ ist dies der einzige echte Unterschied zwischen `class` und `struct`.

Wie eine Klasse darf nämlich auch ein `struct` in C++ ohne das Schlüsselwort `struct` angesprochen werden, und es ist in C++ auch möglich, ein `struct` mit Methoden auszustatten.

Ein C++-`struct` ist also nichts weiter als eine Klasse, deren Elemente, sofern nicht anders deklariert, öffentlich sind. *Davon sollte aber im Normalfall kein Gebrauch gemacht werden – „richtige“ Klassen (mit Methoden) sollten als Klassen und nicht als Verbundtypen definiert werden!*

UMIT
the health & life sciences university

Selbstständiges Üben: Quellcode schreiben

→`2102-calendar` *(Angabe)*;
→`2103-calendar` *(Beispiellösung, nach der Vorlesung)*

- Legen Sie ein neues C++-Konsolenprojekt mit den Quelltextdateien aus dem (in Moodle bereit gestellten) Ordner `2102-calendar-src` an
- Machen Sie sich mit der Projektgliederung vertraut
- Arbeiten Sie die neu hinzugekommenen Methoden und die Erweiterungen in der Funktion `main()` durch und verstehen Sie, was diese machen
- Ergänzen Sie die fehlenden Codeteile in den Methoden `CalendarDate::set_from_daysafter1900()` und `CalendarDate::dow()`
- Testen Sie das Programm mit verschiedenen Daten und prüfen Sie, ob korrekte Ergebnisse berechnet werden

UMIT
the health & life sciences university

Selbstständiges Üben: Quellcode verstehen

→`2104-calendar`

- Legen Sie ein neues C++-Konsolenprojekt mit den Quelltextdateien aus dem (in Moodle bereit gestellten) Ordner `2104-calendar-src` an
- Bringen Sie dieses Programm zum Laufen
- Vergleichen Sie die Funktion des Programms und die Implementation mit dem vorigen Beispiel

UMIT
the health & life sciences university

Abstraktion

- Das Programm hat dieselbe Funktionalität wie das vorangegangene Beispiel.
- Die öffentliche Schnittstelle der Klasse `CalendarDate` in `2104-calendar` (Datei `calendar.h`) ist identisch mit der in `2102-calendar`.

  Folglich kann die Funktion `main()` komplett unverändert bleiben.
- Die privaten Membervariablen sind aber jetzt komplett anders gewählt: Ein Datum wird intern nicht mit Tag/Monat/Jahr, sondern mit der Tageszahl seit 31.12.1900 abgespeichert!

  Dadurch sind die Methodenimplementationen (Datei `calendar.cpp`) deutlich anders aufgebaut; es ist eine private Methode `get_dmy(...)` hinzugekommen.
- Dies ist ein Beispiel für Abstraktion:

  Für Benutzer/innen einer Klasse ist nur die Schnittstelle – die öffentlichen Methoden mit ihrer Spezifikation – ausschlaggebend.

  Wie die in diesen Methoden bereitgestellte Funktionalität intern realisiert wird, welche Datenrepräsentation gewählt wird usw., ist allein Sache der/des Programmierers/in der Klasse.

UMIT
the health & life sciences university

Überladen von Funktionen

- In einem C++-Programm (nicht in C!) kann für mehrere Funktionen der gleiche Name verwendet werden. Dies nennt man **Überladen.**
- Voraussetzung ist, dass die Funktionen gleichen Namens verschiedene Parametertypen haben:

```
int sum (int a, int b);
int sum (int a, int b, int c);
float sum (float a, float b, float c);
```

- Verschiedene Rückgabetypen reichen nicht aus!

  So sind obige Deklarationen nicht verträglich mit einer weiteren Funktion

```
float sum (int a, int b);
```

- Passen die Typen der Parameter in einem Funktionsaufruf nicht genau zu einer der verfügbaren Funktionen, versucht der Compiler, den passenden Aufruf herauszufinden.

Übung: 1. Was passiert bei `sum (1, 2.1, 3);` ? 2. Parameter ist double, nicht float -> deshalb "gewinnen" die beiden int und obige Funktion 2 wird verwendet (da int in überzahl)

2. Was passiert bei `sum (1.0, 2.1, 3.0);` ? Fehler, da 3 double und nicht 3 float

UMIT
the health & life sciences university

Signatur einer Funktion

- Die **Signatur** einer Funktion ist gegeben durch die Anzahl und Typen ihrer Parameter.

```
int sum (int a, int b);                    →  (int, int)
int sum (int a, int b, int c);             →  (int, int, int)
float sum (float a, float b, float c);     →  (float, float, float)
float sum (int a, int b);                  →  (int, int)
```

- An Namen und Signatur erkennt der Compiler eine Funktion.
- Für überladene Funktionen erzeugt der Compiler separate Funktionen gleichen Namens, aber unterschiedlicher Signatur.
- Funktionen gleichen Namens und verschiedenen Rückgabetyps mit gleicher Signatur sind nicht zulässig.
- Überladene Funktionen dürfen sich nicht nur in Referenz-/Wertübergabe ihrer Argumente unterscheiden. Dagegen sind Unterschiede nur in der `const`-Qualifikation von Parametern erlaubt.

Default-Parameter

- Für Parameter (allerdings nicht für Referenzparameter) können Defaultwerte angegeben werden:

```
void set_coordinate_origin (int x = 0, int y = 0)
```

  Mit dieser Deklaration sind folgende Aufrufe erlaubt:

```
set_coordinate_origin (3, 5);
set_coordinate_origin (3); // --> (3, 0)
set_coordinate_origin (); // --> (0, 0)
```

  Jedoch geht nicht `set_coordinate_origin (, 5);`

- Eine Deklaration mit Defaultparametern entspricht der gleichzeitigen Deklaration mehrerer überladener Funktionen.

- Alles hier Gesagte gilt sinngemäß auch für Methoden.

UMIT
the health & life sciences university

Vorüberlegungen

- Ein Prinzip der objektorientierten Programmierung ist, sicherzustellen, dass Objekte jederzeit in einem „definierten Zustand“ sind und gültige Werte enthalten, sodass alle Klassenmethoden korrekt funktionieren.
- In den Setter-Methoden der Klasse `CalendarDate` haben wir uns viel Mühe gegeben, um sicherzustellen, dass Objekte der Klasse nur mit gültigen Werten (Daten zwischen 1.1.1901 und 31.12.2099) belegt werden können
- Unmittelbar nach der Deklaration `CalendarDate date;` enthält `date` jedoch undefinierte Werte!
- Um das Prinzip, dass ein Objekt zu jeder Zeit nur gültige Werte enthalten soll, umzusetzen, müssen wir noch sicher stellen, dass dies von Anfang an der Fall ist.
- Man könnte dafür zB eine Initialisierungsmethode `init()` einführen, die man für jedes neu deklarierte `CalendarDate`-Objekt sofort aufruft und die zB immer erst einmal den 15.3.2019 als Datumswert speichert.

  Dies ist jedoch keine gute Lösung, da immer noch der Aufruf der Initialisierungsmethode im Programm durch den/die Programmierer/in beachtet werden muss. Besser wäre eine „automatische“ Lösung!

UMIT
the health & life sciences university

Konstruktoren

- **Konstruktoren** sind spezielle Methoden einer Klasse.
- Mit ihnen kann die Ausführung initialer Zuweisungen und/oder die Ausführung bestimmter Methodenaufrufe beim Erzeugen eines Objektes erreicht werden.
- Sie sind daher in der Regel die bessere Alternative zu `init()`-Methoden!
- Der Name eines Konstruktors ist gleich dem Klassennamen, z. B. `CalendarDate()`.
- Mehrere Konstruktoren mit unterschiedlichen Signaturen sind möglich (Überladen):

```
CalendarDate();     // Default-Konstruktor
CalendarDate (unsigned int day, unsigned int month,
              unsigned int year); // mit Anfangswerten
```

Nach Deklaration und Implementation dieser Konstruktoren kann man z. B. schreiben

```
CalendarDate date1 (15, 3, 2019);
CalendarDate date2;
```

Für `date1` wird so der Konstruktor mit Initialisierungsparametern aufgerufen, für `date2` dagegen der Default-Konstruktor.

UMIT
the health & life sciences university

Besonderheiten von Konstruktoren

- Konstruktoren werden *nur* bei der Erzeugung von Objekten ausgeführt (einmal pro Instanz), niemals durch direkten Aufruf.
- Aus diesem Grunde hat ein Konstruktor *keinen* Rückgabetyp (noch nicht einmal `void`!).
- Abgesehen davon werden Konstruktoren wie alle anderen Klassenmethoden deklariert und implementiert, also muss die Implementation eines Default-Konstruktors zB mit

  ```
  CalendarDate::CalendarDate() { ... }
  ```

  erfolgen.

Konstruktoren

- Konstruktoren können *public* (Normalfall) oder *private* (selten) sein.
- Wird kein Konstruktor angegeben, so stellt der Compiler einen impliziten Default-Konstruktor (ohne Parameter) bereit. Dieser nimmt keinerlei Initialisierungen vor, d. h. bei Erzeugung einer Instanz wird lediglich der Speicher bereit gestellt.
- Sobald *irgendein* Konstruktor definiert wird, gibt es den impliziten Default-Konstruktor nicht mehr! Mit der Definition von

```
CalendarDate (unsigned int day, unsigned int month,
              unsigned int year);
```

(und nur diesem Konstruktor) wäre die Deklaration ohne Parameterangabe nicht mehr möglich.

UMIT
the health & life sciences university

Selbstständiges Üben: Quellcode schreiben

→`2105-calendar` *(Beispiellösung, nach der Vorlesung)*

- Erweitern Sie die Klasse `CalendarDate` (wahlweise aus `2102-calendar` oder `2104-calendar`) um einen Default-Konstruktor, um sicher zu stellen, dass ein deklariertes `CalendarDate`-Objekt von Anfang an ein gültiges Datum enthält. Beispielsweise könnte nach der Deklaration `CalendarDate date;` das Datum in `date` immer auf den 15. 3. 2019 gesetzt sein.

- Fügen Sie außerdem einen Konstruktor mit Anfangswerten ein, mit dem beispielsweise die Deklaration `CalendarDate date (30, 12, 1955);` den Geburtstag des C++-Erfinders in `date` als Anfangswert speichert.

Sie können bei der Implementation von Konstruktoren auf vorhandene Klassenmethoden zurückgreifen!

Konstruktor kann const-variablen setzen

Konstante Objekte

- Wie andere Variablen können auch Objekte mittels `const` als konstant deklariert werden:

  ```
  const CalendarDate staatsvertrag (15, 5, 1955);
  ```

- Auf so deklarierte Objekte kann nur *lesend* zugegriffen werden. Alle schreibenden Methoden sind nicht aufrufbar.

  (Insbesondere muss die Zuweisung von Werten durch einen Konstruktor erfolgen!)

- *Lesende Methoden* müssen dazu durch den Zusatz `const` als solche gekennzeichnet sein:

  ```
  unsigned int CalendarDate::get_formatted () const;
  ```

  Wäre `get_formatted ()` wie bisher ohne `const` deklariert, würde der Aufruf `staatsvertrag.get_formatted ()` als Compilerfehler gemeldet.

- Das Schlüsselwort `const` ist Teil der Signatur. Eine Methode mit `const` und eine Methode ohne `const` können also bei ansonsten gleicher Parameterliste überladen werden.

- Dies muss konsequent auf alle lesenden Methoden übertragen werden.

UMIT
the health & life sciences university

Selbstständiges Üben: Quellcode schreiben

→`2105-calendar` *(Beispiellösung, nach der Vorlesung)*

- Prüfen Sie in der Klasse `CalendarDate`, die Sie zuvor um Konstruktoren ergänzt haben, welche Klassenmethoden als `const` deklariert werden können, und setzen Sie dies um.
- Testen Sie, ob alles so funktioniert wie vorher.
- Testen Sie mit `const CalendarDate`-Objekten in `main()`.

UMIT
the health & life sciences university

Selbstständiges Üben: Quellcode verstehen

→2106-images-class

- Legen Sie ein neues C++-Konsolenprojekt mit den Quellcode- und Header-Dateien aus dem (in Moodle bereit gestellten) Ordner 2106-images-class-src an
- Bringen Sie dieses Programm zum Laufen
- Erarbeiten Sie sich anhand dieses Programms die darin vorkommenden neuen C++-Sprachelemente

Dynamische Datenelemente in Klassen

- Die Klasse `Image` speichert die Grauwerte des Bildes nicht mehr in einem `vector`-Container. Stattdessen ist ein Datenelement `double *p` vorgesehen, das einen dynamisch allozierten Speicherbereich adressieren soll

- In dieser Situation kommt den Konstruktoren eine wesentliche Aufgabe zu: Sie müssen für die Allokation von Speicherbereichen sorgen!

  Im Beispiel erfolgt dies

  - in der Methode `readpgm` mittels `p = new double [nx * ny];`
  - im Konstruktor `Image::Image (const int, const int)` ebenso;
  - im Konstruktor `Image::Image (const string)` durch Aufruf von `readpgm`

- Wird kein Speicherbereich alloziert, sollte zumindest der Nullzeiger zugewiesen werden. In C++11 wird dafür `nullptr` verwendet.

  Dies geschieht in `Image::Image (const string)` für den Fall, dass `readpgm` fehlschlägt (Rückgabewert `false`)

UMIT
the health & life sciences university

Die Operatoren `new` und `delete`

- Mit dem Operator `new` können in C++ Instanzen beliebiger Objektklassen oder elementarer Datentypen zur Laufzeit angelegt werden:

```
double *pd = new double; // alloziert ein double
```

- Mittels `new` allozierter Speicher wird mittels `delete` freigegeben:

```
delete pd;
```

- Ein Array mit mehreren Instanzen (Variablen) kann mittels `new []` alloziert werden:

```
double *pd_a = new double [nx * ny];
    // Feld fuer nx * ny double-Werte
```

- Ein Speicherbereich, der mittels `new[]` alloziert wurde, muss mit `delete[]` freigegeben werden:

```
delete[] pd_a;
```

Destruktoren

- Wenn ein Objekt Speicherbereiche über Zeiger verwaltet – die dann normalerweise durch Konstruktoren mittels `new` alloziert werden –, so muss dafür Sorge getragen werden, dass diese Speicherbereiche am Ende der Lebenszeit des Objektes auch wieder mittels `delete` freigegeben werden.

- Dafür benötigt man einen **Destruktor.**

  So wie ein Konstruktor dafür da ist, bei Erzeugung eines Objektes nötige Initialisierungen vorzunehmen, ist ein Destruktor dazu da, alles Erforderliche für eine ordnungsgemäße „Entsorgung“ des Objektes zu veranlassen.

- Der Name eines Destruktors ist stets `~`*Klassenname*, also etwa

  ```
  Image::~Image(){...}
  ```

  (quasi ein mit `~` negierter Konstruktor)

- Ein Destruktor hat keinen Rückgabewert und keine Parameter.