Objektorientierte Programmierung
(Winter 2006/2007)

Kapitel 9: Klassen / Strukturen

- Strukturen
- Klassen: Methoden (Member Functions)
- Daten-Kapselung: public, private
- Konstruktoren
- Klassen-Attribute und Methoden (Static Members)

Strukturen: Motivation (1)

- Manchmal möchte man mehrere Datenwerte zu einer Einheit zusammenfassen.
- Bei Arrays gilt:
  - ◇ Alle Array-Elemente haben den gleichen Typ.
  - ◇ Die Elemente werden über Zahlen identifiziert.
- Im Unterschied dazu gilt bei Strukturen:
  - ◇ Die Elemente einer Struktur können unterschiedlichen Typ haben.
  - ◇ Die einzelnen Elemente werden über Namen (Bezeichner) identifiziert.

Strukturen: Motivation (2)

- Strukturen sind ein Mittel zur Abstraktion:
  - ◇ Man kann auf einer höheren Programmebene von den einzelnen Komponenten (Feldern, Attributen) der Struktur abstrahieren.
  - ◇ Man behandelt die Struktur dann als Ganzes.
  - ◇ Bei Bedarf kann man natürlich immer noch auf die Komponenten zugreifen.
- In Pascal heißen Strukturen Records.
- In relationalen Datenbanken werden Strukturen als (Tabellen-)Zeilen (oder "Tupel") verwendet.

Strukturen: Motivation (3)

- Allgemein geht es jetzt darum, neue Datentypen zu definieren, die über die eingebauten Typen wie `int`, `float` hinausgehen.
- Stroustrup (The C++ Prog. Lang., 2000):
  - ◇ A type is a concrete representation of a concept.
  - ◇ A program that provides types that closely match the concepts of the application tends to be easier to understand and easier to modify than a program that does not.

Strukturen (1)

```
...
struct date {
        int day;
        int month;
        int year;
};
int main()
{
        date d;
        d.day = 24; d.month = 12; d.year = 2006;
        cout << d.day << "." << d.month << "."
                << d.year;
        return 0;
}
```


Strukturen (2)

- Vereinfacht besteht eine Strukturdeklaration aus:
  - ◇ Dem Schlüsselwort "`struct`",
  - ◇ dem Namen des neuen Strukturtyps,

    Der Name sollte möglichst nicht mit dem Namen einer Variablen, Funktion etc. übereinstimmen. Aus Kompatibilitätsgründen zu C ist das zwar möglich, aber dann muß man den Strukturtyp später immer mit dem vorangestellten Schlüsselwort "`struct`" ansprechen. In C ist das immer erforderlich, aber man kann mit "`typedef`" einen Namen für den Strukturtyp definieren.
  - ◇ "Variablendeklarationen" in `{...}`.

    Eigentlich werden Komponenten deklariert, aber die Deklarationen sehen syntaktisch aus wie Variablendeklarationen (mit Einschränkungen, z.B. kein `const`, `static`, etc.).

Strukturen (3)

- Von dem neuen Strukturtyp kann man anschließend Variablen deklarieren, natürlich auch mehrere, und auch Arrays, Pointer, u.s.w.:

```
date d1, d2;
date d3;
date a[5];
date *p;
```

- Man kann Strukturen bei der Deklaration wie Arrays initialisieren:

```
date d1 = { 24, 12, 2006 };
```


Strukturen (4)

- Jeder Wert von Typ "`date`" hat die Komponenten (engl. "component", "field", "member") "`day`", "`month`", "`year`".
- Man greift auf eine Komponente mit dem Operator "." zu, z.B.:

```
a[i].year
```

- Wenn man auf die Komponenten zugreifen will, muß man einen Wert vom Typ `date` angeben.

  Man kann nicht einfach nur "`year`" schreiben (außer innerhalb von "Member Functions", s.u.). Obwohl die Deklarationen der Komponenten wie Variablendeklarationen aussahen, gibt es keine solche Variablen außerhalb der Struktur.

Strukturen (5)

- Da man häufig mit Zeigern auf Strukturen arbeitet. gibt es die Abkürzung ->. Z.B. steht

  `p->year`

  für `(*p).year`.

- Man kann Strukturen des gleichen Typs an einander zuweisen.

  Auf diese Art werden eventuell größere Datenmengen kopiert. Insbesondere könnte eine Struktur auch ein Array enthalten, was dann mitkopiert wird. Es ist etwas inkonsistent, daß man Arrays nicht direkt zuweisen kann. Vielleicht liegt das auch daran, daß der Name des Arrays immer für die Basisadresse steht, also sofort in einen Pointer umgewandelt wird. Den kann man natürlich kopieren. In alten C-Varianten konnte man auch keine Strukturen zuweisen.

Strukturen (6)

- Entsprechend können Strukturen auch an Funktionen "by value" übergeben werden, und von Funktionen als Ergebniswert zurückgegeben werden.

  Man sollte sich bewußt sein, daß dies bei großen Strukturen nicht besonders effizient ist. "Effizient" heißt geschickter Umgang mit den Resourcen, so daß eine Lösung für die gegebene Aufgabe möglichst schnell ist, bzw. möglichst wenig Speicherplatz verbraucht.

- Oft ist es besser, mit Zeigern auf Strukturen zu arbeiten, oder mit Referenzen (insbesondere `const`).

  Kleine Strukturen, die insgesamt nicht größer als 4 oder 8 Byte sind, können dagegen noch mit einzelnen Maschinenbefehlen kopiert werden. Z.B. könnte man `day` und `month` in einem `char` abspeichern, und `year` als `short int`.

Strukturen (7)

- Zwei getrennt deklarierte Struktur-Typen sind nicht gleich, selbst wenn sie die gleichen Komponenten haben:

```
struct date2 {
        int day;
        int month;
        int year;
};
```

- Die Typen `date` und `date2` sind verschieden, Zuweisungen dazwischen sind ausgeschlossen.

- Verschiedene Strukturen dürfen Komponenten mit gleichem Namen haben.

Strukturen (8)

- Man kann Strukturen nicht mit "`==`" und "`!=`" vergleichen (außer durch Überladen dieser Operatoren, aber dann muß man es sich selbst definieren, s.u.).

  Es scheint etwas inkonsistent, daß die Zuweisung automatisch funktioniert, aber nicht der Vergleich.

  Ein Grund könnte sein, daß es für Strukturen zwei Gleichheits-Begriffe gibt: Gleiche Adresse oder Gleichheit aller Komponenten. Wenn hier die Gleichheit aller Komponenten zunächst natürlicher erscheint, was ist dann mit einer Struktur, die einen Zeiger auf eine andere Struktur enthält? Sollen hier rekursiv die Komponenten verglichen werden?

  Ein anderer Grund ist, daß Strukturen "Lücken" aufweisen können (s.u.), in denen beliebiger Datenmüll stehen kann. Zwei Strukturen sind gleich, selbst wenn in den Lücken verschiedene Bitmuster stehen. Dann sind effiziente Vergleiche in großen Einheiten nicht möglich, sondern es müssen die Komponenten einzeln verglichen werden.

`struct`: Implementierung (1)

- Der Compiler berechnet für jede Komponente der Struktur einen "Offset" von der Startadresse der Struktur:

0	4	8	12
`day`	`month`	`year`	

- Angenommen, der Typ `int` ist wie heute üblich vier Byte groß (32 Bit).
- Die Struktur `date` ist dann 12 Byte groß.

`struct`: Implementierung (2)

- Falls die Variable `d` vom Typ `date` z.B. bei Adresse 2000 beginnt, würde
  - ◇ die Komponente `day` auch bei Adresse 2000
  - ◇ die Komponente `month` bei Adresse 2004
  - ◇ die Komponente `year` bei Adresse 2008

  stehen.
- Ein Ausdruck wie `p->year` wird also ausgewertet, indem die ganze Zahl gelesen wird, die im Hauptspeicher an der Adresse "Inhalt von `p` plus 8" steht.

`struct`: Implementierung (3)

- Viele CPUs haben Einschränkungen hinsichtlich der Adressen, an denen größere Werte stehen können.
- Z.B. können 32-Bit (4 Byte) Werte nur an durch 4 teilbaren Adressen stehen.

  Wenn der Bus zwischen CPU und Hauptspeicher 32 Bit groß ist, geschieht der Datenaustausch normalerweise in 32-Bit Einheiten an durch 4 teilbaren Adressen. Außerdem gibt es vier Steuerleitungen, mit denen die CPU einzelne Bytes selektieren kann (besonders wichtig zum Schreiben in den Hauptspeicher). Wenn man nun einen `int`-Wert ab Adresse 3 lesen wollte, wären zwei Speicherzugriffe nötig: Einer auf die Bytes `0..3`, von denen nur das letzte Byte verwendet wird, und einer auf die Bytes `4..7`, von denen die ersten drei verwendet werden. Viele CPUs können die 32-Bit-Größen nicht auf diese Art in Teilen lesen, und wenn sie es können, führt es zu einer Verlangsamung.

`struct`: Implementierung (4)

- Entsprechend können `double`-Werte möglicherweise nur an durch 8 teilbaren Adressen stehen.
- Englisch spricht man von "alignment requirement".
- Sei folgende Struktur betrachtet:

```
struct date3 {
        char day;
        char month;
        int year;
};
```

- Die Komponente `day` hat den Offset 0, `month` den Offset 1, und `year` z.B. den Offset 4.

`struct`: Implementierung (5)

- In diesem Fall führt die Einschränkung, daß der `int`-Wert `year` an einer durch vier teilbaren Adresse stehen muß, zu einer Lücke in der Struktur:

0	1	2	4	8
`day`	`month`		`year`	

- Die beiden Bytes mit Offset 2 und 3 in der Struktur werden nicht genutzt.
- Die ganze Struktur ist entsprechend 8 Bytes groß, obwohl die Summe der Größen der Komponenten nur 6 Byte ist.

`struct`: Implementierung (6)

- Test mit MS Visual C++ 6.0 auf Intel Pentium III:

```
date3 d;
cout << (long) &d          << '\n'; // 7077360
cout << (long) &(d.day)    << '\n'; // 7077360
cout << (long) &(d.month)  << '\n'; // 7077361
cout << (long) &(d.year)   << '\n'; // 7077364
cout << sizeof(date3)      << '\n'; // 8
cout << ((long) &d) % 4    << '\n'; // 0
```

- Natürlich müssen Variablen des Strukturtyps selbst an durch vier teilbaren Adressen stehen.

  Der Compiler verwaltet für jeden Typ die Anforderung an die Ausrichtung der Variablen.

`struct`: Implementierung (7)

- C garantiert, daß die Komponenten einer Struktur in der deklarierten Reihenfolge im Speicher stehen.
- Manchmal kann man die Lücken durch Umsortierung der Komponenten verringern.

  Eine verständliche Anordnung ist aber meist wichtiger.
- Der Compiler legt bei Bedarf auch am Ende eine Lücke an, so daß die Strukturen hintereinander in einem Array stehen können.

  Es würde nichts bringen, `year` zuerst zu deklarieren. Die Struktur bleibt 8 Byte groß, so daß, wenn der Anfang auf einer durch 4 teilbaren Grenze liegt, daß Ende auch an einer solchen Speichergrenze liegt.

Union (1)

- Benutzt man das Schlüsselwort `union` statt `struct`, bekommen alle Komponenten den Offset `0`.

  Der `union`-Typ ist so lang wie der längste Komponenten-Typ, plus ggf. einige "Padding"-Bytes, um die Ausrichtung wieder herzustellen.
- Dann darf man natürlich nur eine Komponente einer `union`-Variable gleichzeitig benutzten.

  Man darf nur auf die Komponente lesend zugreifen, die man zuletzt geschrieben hat. Beim Schreiben einer Komponente wird der Inhalt aller anderen Komponenten zerstört.
- Deswegen der Name "`union`": Der Typ ist gewissermaßen die Vereinigung der Komponenten-Typen.

Union (2)

- Der Compiler kann die korrekte Verwendung von `union`-Komponenten nicht prüfen.
- Typisch ist die Verwendung einer Struktur, die zuerst einen Code enthält, welche Komponente ("Variante") der Union gerade benutzt wird:

```
struct number {
        bool  is_int;
        union {
                int    int_val;
                double double_val;
        };
};
```


Union (3)

- Ist `x` eine Variable vom Typ `number`, so sollte man vor dem Zugriff auf `x.int_val` prüfen, daß `x.is_int` wahr ist.

  Diese Konstruktion entspricht den varianten Records in Pascal.

- Während `union` in C relativ wichtig war, benutzt man in C++ eher Subklassen (s.u.).

  Auch in C führt `union` nur zu einer Reduktion des Speicherbedarfes. Ansonsten könnte man auch `struct` verwenden. Bei Strukturen ist wie bei Arrays keineswegs verlangt, daß alle Komponenten initialisiert sind (natürlich darf man nur initialisierte Komponenten auslesen).

  Klassen kann man nicht als `union`-Komponente verwenden (unklar, welcher Konstruktor etc. aufgerufen werden soll, s.u.).

Geschachtelte Strukturen (1)

- Eine Komponente einer Struktur kann selbst wieder eine Struktur sein:

```
struct person {
        char  first_name[20];
        char  last_name[20];
        date  birth_date;
};
```

- Die Struktur `date` muß schon vorher deklariert sein.
- Der Zugriff auf Komponenten geschieht wie vorher, dabei entstehen Pfadausdrücke, z.B. (wenn `x` eine Variable vom Typ `person` ist): `x.birth_date.month`

Geschachtelte Strukturen (2)

- Eine häufige Datenstruktur sind verkettete Listen.
- Hier enthält die Struktur einen Zeiger auf eine weitere Struktur vom gleichen Typ:

```
struct person_list {
        char  first_name[20];
        char  last_name[20];
        date  birth_date;
        person_list  *next;
};
```

- Nachdem der Compiler `struct person_list` gelesen hat, ist die Struktur vorläufig/partiell deklariert.
- Dann kann man schon Zeiger darauf deklarieren.

Geschachtelte Strukturen (3)

- Das funktioniert, weil alle Zeiger gleich groß sind, egal auf welchen Typ sie zeigen.
- Die genauen Komponenten der Struktur oder ihre Größe sind also nicht nötig, um eine Komponente von dem Zeigertyp einführen zu können.
- Selbstverständlich kann eine Struktur vom Typ T aber nicht eine Komponente vom gleichen Strukturtyp T enthalten (ohne Zeiger).

  Einerseits ist die Größe der Struktur zu diesem Zeitpunkt nicht bekannt, und andererseits würde sie durch eine solche Rekursion auch unendlich.

Geschachtelte Strukturen (4)

- Man kann die vorläufige Strukturdeklaration auch getrennt aufschreiben. Das ist z.B. nötig, wenn zwei Strukturen sich gegenseitig referenzieren:

```
struct department; // incomplete declaration
struct employee {
        char  first_name[20];
        char  last_name[20];
        department  *works_in;
};
struct department {
        char  dept_name[80];
        employee  *head;
};
```


Exkurs: Aufzählungstypen (1)

- Ein Aufzählungstyp ist ein Datentyp, der nur eine kleine Anzahl möglicher Werte hat, die durch explizite Aufzählung definiert werden.
- Die Werte sind Bezeichner (symbolische Konstanten), haben in C++ aber auch einen Integer-Wert.
- Beispiel:

```
enum weekdays {monday, tuesday, ..., sunday};
```

- Dieser Typ hat sieben mögliche Werte, nämlich die angegebenen Konstanten für die Wochentage Montag, Dienstag, . . . , Sonntag (genauer: s.u.).

Exkurs: Aufzählungstypen (2)

- Wie von anderen Datentypen auch, kann man von dem Aufzählungstyp Variablen deklarieren, ihn als Ergebnis- oder Argumenttyp einer Funktion verwenden, oder als Komponententyp einer Struktur.
- Beispiel (Variablendeklaration):

```
weekdays w;
```

  In C müßte man "`enum weekdays w;`" schreiben. Dies kann man sich dann mit einer `typedef`-Deklaration vereinfachen, s.u.

- Der Variablen `w` kann man dann einen der Werte zuweisen, z.B. `w = sunday;`

Exkurs: Aufzählungstypen (3)

- Sehr typisch ist auch ein Switch über allen möglichen Werten des Aufzählungstyps:

```
switch(w) {
case monday:
        cout << "Montag";
        break;
...
case sunday:
        cout << "Sonntag";
        break;
}
```

- Manche Compiler geben eine Warnung aus, wenn nicht alle Fälle behandelt sind.

Exkurs: Aufzählungstypen (4)

- Die Konstanten eines Aufzählungstyps haben einen Zahlwert, und werden in arithmetischen Ausdrücken implizit nach `int` umgewandelt.

  U.U. auch `unsigned int` oder `long`, falls die Zahlwerte dieses Typs zu groß für `int` sind.

- Normalerweise sind sie von 0 beginnend durchnumeriert:
  - ◇ `monday` hat den Wert 0.
  - ◇ ...
  - ◇ `sunday` hat den Wert 6.

Exkurs: Aufzählungstypen (5)

- Man kann aber auch explizit Zahlwerte festlegen:

```
enum weekdays {monday = 1, ..., sunday = 7};
```

  Man braucht nicht unbedingt für alle Konstanten des Aufzählungstyps einen Zahlwert festlegen: Es wird dann der Wert der letzten Konstante plus 1 verwendet. Im Beispiel würde man also den gleichen Effekt erreichen, wenn man nur für `monday` einen Wert festlegt.

- Während Aufzählungstypen automatisch nach `int` umgewandelt werden, gilt das Umgekehrte nicht.
- Bei Bedarf kann man folgende Notation verwenden:

```
w = weeksdays(3);
```

  In C muß man "`w = (weeksdays) 3;`" schreiben.

Exkurs: Aufzählungstypen (6)

- Ein häufiger Trick ist die Verwendung von Zweierpotenzen, um auch Mengen der Aufzählungswerte durch "Bit-oder" repräsentieren zu können:

```
enum weekdays {
        monday    =  1,
        tuesday   =  2,
        wednesday =  4,
        thursday  =  8,
        friday    = 16,
        saturday  = 32,
        sunday    = 64
}
```

Exkurs: Aufzählungstypen (7)

- Nun kann man z.B. mit `saturday|sunday` das Wochenende repräsentieren.

  In C++ gilt deswegen die Regel, daß der legale Wertebereich eines Aufzählungstyps (bei positiven Konstanten) sich immer bis $2^n - 1$ erstreckt, wobei n minimal so gewählt wird, daß alle Werte der Konstanten in den Bereich passen. Falls es negative Konstanten gibt, wird entsprechend ein Bereich der Form -2^n bis $+2^n - 1$ gewählt. Dies entspricht nicht der in anderen Sprachen üblichen Auffassung eines Aufzählungstyps (dort sind wirklich nur die angegebenen Konstanten legal), aber die zusätzlichen "anonymen" Werte kann man jedenfalls nur durch eine explizite Umwandlung einer Zahl in den Aufzählungstyp bekommen.

- Entsprechend kann man mit `w & sunday` testen, ob `w` den Sonntag enthält.

Von Strukturen zu Klassen (1)

- Ein Datentyp legt nicht nur eine Wertemenge fest, sondern es gehören immer auch Operationen dazu.
- Nachdem man die Strukturtypen wie oben gezeigt deklariert hat, ist alles, was man damit machen kann, auf die Komponenten zuzugreifen.
- Zu `date` sollten weitere Funktionen gehören, z.B.:
  - ◇ Man sollte den Wochentag zu einem Datum bestimmen können.
  - ◇ Man sollte das Datum des nächsten Tages bestimmen können (oder Anzahl Tage addieren).

Von Strukturen zu Klassen (2)

- Früher hat man die Funktionen zu einem Strukturtyp getrennt deklariert (allerdings in der gleichen Include-Datei), z.B.

```
weekday day_of_week(date d);
```

- Bei Bedarf konnten sich auch die Anwender des Typs `date` in gleicher Weise noch weitere Funktionen für Datumsangaben definieren.

  Die sind dann natürlich beliebig über ein großes Programm verteilt. Es wäre vernünftig, sie an einer Stelle zu versammeln, aber niemand ist dazu gezwungen: Alle haben den vollen Zugriff auf die Struktur-Komponenten, und es gibt keinen prinzipiellen Unterschied zwischen den "mitgelieferten" und "nachträglich hinzugefügten" Funktionen.

Von Strukturen zu Klassen (3)

- Angenommen, es stellt sich später heraus, daß eine andere Repräsentation der Datumsangaben besser wäre, z.B. als Anzahl Tage seit einem festen Startdatum (1.1.1970).
- Dann ist es sehr schwer, das Programm entsprechend zu ändern: Die Zugriffe auf die Komponenten sind über das ganze Programm verteilt.
- Hier setzt das Klassenkonzept an, indem es eine explizit definierte Schnittstelle für Strukturen einführt.

Von Strukturen zu Klassen (4)

- Mit Klassen ist es möglich, die Komponenten der Struktur selbst zu verstecken:
  - ◇ Man kann von außen nicht mehr direkt auf die Komponenten zugreifen,
  - ◇ sondern nur noch über die explizit definierten Zugriffsfunktionen.
- Durch den indirekten Zugriff auf die Komponenten kann man die Implementierung des Datentyps (sein "Innenleben") nachträglich ändern, aber die Schnittstelle (nach außen) stabil halten.

Von Strukturen zu Klassen (5)

- Natürlich wird man im Datums-Beispiel Funktionen haben, die den Tag, Monat, Jahr liefern.
- Damit ist aber zunächst nur ein lesender Zugriff auf diese Komponenten möglich.
- Man wird sicher auch Funktionen zum Setzen des Datums anbieten (zumindest zur Initialisierung).
- Diese Funktionen sollten prüfen, daß die Datumsangabe korrekt ist, z.B. nicht 35.20.2007.

Von Strukturen zu Klassen (6)

- Man beachte den Unterschied:
  - ◇ Wenn mit Strukturen ein ungültiges Datum auftritt, kann der Fehler an beliebiger Stelle im Programm sein: Überall hat man Zugriff auf die Komponenten der Struktur.
  - ◇ Wenn mit der Klasse ein ungültiges Datum auftritt, hat eine der Funktionen der Klasse es verursacht oder zumindest "durchschlüpfen lassen".

    Wenn die Funktion mit ungültigen Eingabewerten aufgerufen wird, hätte sie das Programm normalerweise mit einer Fehlermeldung beenden sollen (oder eine "Exception auslösen", s.u.).

Klassen: Weitere Konzepte (1)

- Werte eines "Klassen-Datentyps" heißen Objekte.

  Manche Leute legen wert darauf, daß Objekte eben keine Werte sind. Siehe die Diskussion zur Objektidentität auf Folie 9-44.
- Ein Objekt einer Klasse heißt auch Instanz dieser Klasse. Man erhält ein Objekt durch Instanziierung einer Klasse.
- Die Komponenten heißen auch Instanz-Variablen, Attribute oder Datenelemente (der Klasse).
- Der Zustand eines Objektes ist die Belegung seiner Komponenten/Attribute mit konkreten Werten.

Klassen: Weitere Konzepte (2)

- Bei Klassen gibt es auch einen klaren syntaktischen Unterschied zwischen
  - ◇ den offiziellen Funktionen der Klasse: Hier sieht der Funktionsaufruf ähnlich zu einem Komponentenzugriff aus:

    ```
    d.day_of_week()
    ```

  - ◇ und getrennt definierten Funktionen, die auch Objekte der Klasse verarbeiten: Hier wird der Datumswert als normales Argument übergeben:

    ```
    day_of_week(d)
    ```


Klassen: Weitere Konzepte (3)

- Der syntaktische Unterschied des Funktionsaufrufs motiviert vielleicht, alle Funktionen auf Datumswerten in der Klassendefinition zusammenzufassen.
- Die Funktionen, die die Schnittstelle der Klasse ausmachen, werden auch Methoden genannt.

  Mindestens in Smalltalk wird der Aufruf einer Methode auch als das Senden einer Nachricht an das Objekt aufgefasst. Stroustrup verwendet den Begriff "Methode" nur für virtuelle Funktionen (die beim Vererben überschrieben werden können), siehe nächstes Kapitel.
- Stroustrup nennt die Methoden "member functions" und die Komponenten "data members".

  Beides sind "member" der Klasse.

Klassen: Weitere Konzepte (4)

- Das Neue, was das Klassenkonzept bringt, ist also die Möglichkeit, Daten und Programmcode zu "verkapseln", so daß man auf die Daten nur über die definierten Funktionen/Methoden zugreifen kann.

  Tatsächlich gab es das auch schon mit dem Konzept der abstrakten Datentypen, bei denen man von der Implementierung des Datentyps abstrahiert, und sich nur für die Schnittstelle, also die Operationen interessiert, und die Axiome, die diese Operationen erfüllen müssen.
- Die andere Syntax für Methodenaufrufe ist nur eine andere Syntax: Jede Methode hat ein implizites "0-tes" Argument für das Objekt der Klasse, auf das die Methode angewendet wird.

Klassen: Weitere Konzepte (5)

- Zu den objektorientierten Konzepten gehört üblicherweise noch die Objektidentität: Zwei Objekte können auch dann verschieden sein, wenn sie in allen Komponenten übereinstimmen.
- Dem gegenüber sind Datenwerte gleich, wenn sie in allen Komponenten übereinstimmen.
- Wenn ein Datenwert geändert wird, ist es ein neuer Datenwert. Ein Objekt kann dagegen geändert werden, aber das gleiche Objekt bleiben.
- `date` ist hier ein schlechtes Beispiel.

Klassen: Erstes Beispiel

```
class Date {
    private:
        int day;
        int month;
        int year;
    public:
        int get_day()   { return day; }
        int get_month() { return month; }
        int get_year()  { return year; }
        void init(int d, int m, int y);
        weekday day_of_week();
};
```


Klassen: private/public (1)

- Es sind jetzt klar von einander getrennt:
  - ◇ `private`: Der von außen nicht zugreifbare Bereich. Nur die Funktionen (Methoden) der Klasse können diese Komponenten lesen/schreiben.
  - ◇ `public`: Die Schnittstelle nach außen.
- Die Voreinstellung ist `private`, dieses Schlüsselwort hätte also nicht angegeben werden müssen.

  Tatsächlich ist in C++ der einzige Unterschied zwischen `struct` und `class`, daß bei `struct public` voreingestellt ist, und bei `class private`. Das klassische `struct` in C hat dagegen keine Funktionskomponenten.

Klassen: private/public (2)

- In C++ können die Methoden der Klasse auf die `private`-Komponenten aller Objekte der Klasse zugreifen,
  - ◇ d.h. selbst wenn eine Methode für ein Objekt X der Klasse aufgerufen wurde,
  - ◇ kann sie auch auf die `private`-Komponenten eines Objektes Y der gleichen Klasse zugreifen.
- In Smalltalk geht das nicht: Eine Methode kann immer nur auf die `private`-Komponenten des Objekts zugreifen, für das sie aufgerufen wurde.

Klassen: private/public (3)

- Selbstverständlich kann man sich auch in C++ an die einschränkendere Regel von Smalltalk halten (Programmierstil).

  Das ist wohl stärker objektorientiert. Man muß dann auch für andere Objekte der gleichen Klasse eine Methode dieser Objekte aufrufen, anstatt direkt auf ihre Komponenten zuzugreifen.

- Man darf `public`/`private` in beliebiger Reihenfolge und auch mehrfach verwenden.

  Für den Anwender der Klasse ist ja nur der `public`-Teil interessant. Deswegen macht es Sinn, den zuerst anzugeben.

Klassen: private/public (4)

- Die Deklaration der Komponenten als "`private`" schützt nicht vor bewußten Umgehungsversuchen.

  Wenn man einen Zeiger auf das Objekt hat, und die Offsets der Komponenten kennt, kann man sie natürlich auch von außen auslesen oder sogar verändern. Das ist aber völlig inakzeptabeler Stil.

- Wenn es auch üblich ist, daß die Daten private und die Funktionen public sind, ist das keine Vorschrift.

  Z.B. kann man lokale Hilfsfunktion auch als `private` deklarieren.

inline-Funktionen (1)

- Im Beispiel ist für drei besonders einfache Methoden der Funktionsrumpf direkt in der Klassendeklaration mit angegeben.
- Diese Funktionen sind dann automatisch sogenannte "`inline`"-Funktionen:
  - ◇ Der Compiler versucht den Overhead eines expliziten Funktionsaufrufs zu vermeiden (Rücksprungadresse auf Stack legen u.s.w.).
  - ◇ Statt dessen ersetzt er den Funktionsaufruf im Prinzip durch eine Kopie des Funktionsrumpfes (mit Ersetzung der Parameter).

inline-Funktionen (2)

- Man kann beliebige (nicht-rekursive) Funktionen durch Voranstellen des Schlüsselwortes `inline` als Inline-Funktionen kennzeichnen.

  Es würde auch nichts schaden, die in der Klasse definierten Methoden explizit als "`inline`" zu kennzeichnen.

- Der Compiler wird dann den Funktionsaufruf auf diese Art "expandieren".
- Das ist schneller, verbraucht aber mehr Speicherplatz für Programmcode.

  Wenn die Funktion nicht sehr kurz ist: Dann könnte es sogar Speicherplatz sparen.

Methoden-Definition (1)

- Weil beim Aufruf der Methoden immer ein Objekt gegeben ist, kann im Methodenrumpf auf die Komponenten (Attribute, Instanzvariablen) ohne Angabe eines Objektes zugegriffen werden.

  D.h. die Komponenten können jetzt so benutzt werden wie Variablen. Man darf aber natürlich ein Objekt angeben, wenn man sich auf die Komponenten des eines anderen Objektes der Klasse beziehen will (möglicherweise schlechter Stil).

- Falls man das aktuelle Objekt explizit ansprechen muß, gibt es dafür das Schlüsselwort "`this`".

  In anderen objektorientierten Sprachen (Smalltalk) heißt es "`self`".

Methoden-Definition (2)

- Bei der Definition von Methoden in der Klassendeklaration selbst gibt es eine Ausnahme zu der Regel, daß alles vor der Verwendung deklariert sein muß:
  - ◇ Man kann auch auf erst später deklarierte Komponenten zugreifen.
  - ◇ Im Beispiel könnte man den `private`- und den `public`-Teil also vertauschen.
  - ◇ Formal wird ein Methodenrumpf erst verarbeitet, nachdem der Compiler die eigentliche Klassendeklaration vollständig gesehen hat.

Methoden-Definition (3)

- Wenn man eine Methode außerhalb der Klasse definiert, muß man den Klassennamen explizit als Namensraum angeben, z.B.

```
inline void Date::init(int d, int m, int y)
{
        // Prüfung der Parameter:
        ...
        day = d; month = m; year = y;
}

weekday Date::day_of_week()
{
        ...
}
```


Methoden-Definition (4)

- Selbstverständlich dürfen verschiedene Klassen Methoden mit dem gleichen Namen haben.
- Deswegen ist es wichtig, die Methode durch Voranstellen des Klassennamens und des Namensraum-Operators "`::`" eindeutig einer Klasse zuzuordnen.

  Der Klassenname wird sozusagen Bestandteil des Funktionsnamens.
- Wenn die Methode aufgerufen wird, kennt der Compiler den Typ (die Klasse) des Objektes, für das die Methode aufgerufen wird, und kann so die richtige Variante auswählen (ohne Namensraum-Angabe).

const-Methoden

- Man kann explizit angeben, daß eine Methode den Zustand des Objektes nicht verändert:

```
class Date {
        ...
    public:
        int get_day()   const { return day; }
        int get_month() const { return month; }
        int get_year()  const { return year; }
        void init(int d, int m, int y);
        weekday day_of_week() const;
};
```

- Wenn man `day_of_week` später definiert, muß `const` noch einmal mit angegeben werden.

Konstruktoren (1)

- So wie die Klasse `date` bisher deklariert ist, gibt es keine Garantie, daß die Methode `init` auch aufgerufen wird.
- Auch sonst können nicht initialisierte Variablen vorkommen, aber bei Typen wie `int` kann man die Initialisierung gleich bei der Deklaration vornehmen.

  Wenn man das nicht tut, ist man selber schuld, wenn man bei eventuellen Fehlern lange suchen muß.
- So wie oben deklariert, kann `init` aber nicht gleich bei der Deklaration mit aufgerufen werden.

Konstruktoren (2)

- Daher kann man in C++ Konstruktor-Methoden deklarieren (kurz: Konstruktoren).

  Die obige Lösung mit `init` wurde nur aus pädagogischen Gründen gewählt. Konstruktor-Methoden sind normalerweise besser. Ggf. kann es `set` oder verschiedene `set_*`-Methoden für nachträgliche Änderungen geben.
- C++ garantiert, daß für jede Variable der entsprechenden Klasse ein Konstruktor aufgerufen wird.

  Es kann eventuell mehrere mit unterschiedlichen Argumenttypen geben (s.u.).

Konstruktoren (3)

- Konstruktoren sind dadurch gekennzeichnet, daß sie den gleichen Namen wie die Klasse haben:

```
class Date {
    private:
        ...
    public:
        Date(int d, int m, int y);
        int get_day()   const { return day; }
        ...
};
```


Konstruktoren (4)

- Für Konstruktor-Methoden wird kein Resultattyp angegeben.

  Der Aufruf wird innerhalb der Deklaration ausgeführt, nicht in einem Wertausdruck.
- Ansonsten werden Konstruktor-Methoden wie andere Methoden auch definiert:

```
Date::Date(int d, int m, int y)
{
        day = d;
        month = m;
        year = y;
}
```

Konstruktoren (5)

- Man kann die Konstruktoren auch direkt in der Klassendeklaration vollständig definieren:

```
class Date {
    private:
        int day; int month; int year;
    public:
        Date(int d, int m, int y)
        {
                day = d;
                month = m;
                year = y;
        }
        ...
};
```


Konstruktoren (6)

- Bei der Variablendeklaration müssen dann die Parameter für den Konstruktor angegeben werden:

```
Date d(24, 12, 2006);
```

- Dies deklariert die Variable `d`, die eine Instanz der Klasse `Date` enthält (ein `Date`-Objekt), initialisiert mit dem 24.12.2006.
- Auch folgende Syntax ist möglich (erinnert an die übliche Initialisierung):

```
Date d = Date(24, 12, 2006);
```


Konstruktoren (7)

- Wenn man einen Konstruktor ohne Parameter definiert, funktioniert die normale Deklaration `Date d;` (und ruft diesen Konstruktor auf).

  Falls nur ein Konstruktor mit Parametern deklariert ist, geht diese Variablendeklaration dagegen nicht. Man bekommt die Fehlermeldung "No appropriate default constructor available".

- Deklariert man gar keinen Konstruktor, legt C++ einen Konstruktor ohne Argumente an, der (im Beispiel) nichts tut (`day` etc. bleiben uninitialisiert).
- Konstruktoren ohne Argumente (C++ generiert oder selbst definiert) heißen Default-Konstruktoren.

Konstruktoren (8)

- Selbstverständlich werden Konstruktoren auch bei einer Array-Deklaration aufgerufen:

```
Date a[3];
```

  Das funktioniert natürlich nur, wenn es einen Konstruktor ohne Argumente gibt.

- Man kann probehalber eine Ausgabe-Anweisung in den Konstruktor schreiben, um das zu testen.

  Bei Bedarf könnte man darin auch die Adresse von `this` ausgeben, und mit den Adressen der Variablen vergleichen, um zu sehen, für welche Variable der Konstruktor aufgerufen wurde.

Konstruktoren (9)

- Konstruktoren für globale Variablen werden vor Beginn der Ausführung von `main` aufgerufen.
- Konstruktoren für normale lokale Variablen werden jedesmal aufgerufen, wenn die Deklaration ausgeführt wird.

  Dies ist vielleicht auch ein Grund, warum die Deklaration in C++ ein Statement ist. Falls man die Variable in einem Block in einer Schleife deklariert, wird der Konstruktor entsprechend häufig aufgerufen.
- Konstruktoren für statische lokale Variablen werden aufgerufen, wenn die Deklaration zum ersten Mal ausgeführt wird.

Konstruktoren: Member (1)

- Wenn Komponenten ("data members") selbst Objekte sind, garantiert C++ natürlich auch, daß sie durch Aufruf eines Konstruktors initialisiert werden.

  Dies gilt nicht für Komponenten, die einen normalen Datentypen wie `int` haben. Das ist etwas inkonsistent. Stroustrup schreibt, daß es so kompatibeler mit C ist, und der Overhead zu groß schien.
- Wenn es einen Konstruktor ohne Parameter gibt, braucht man sich um den Konstruktor-Aufruf nicht zu kümmern: Das geht automatisch.

Konstruktoren: Member (2)

- Soll die Komponente dagegen durch einen Konstruktor mit Parametern initialisiert werden, muß man eine spezielle Syntax verwenden:
  - ◇ Nach dem Funktionskopf des Konstruktors (Name, Parameterliste) kommt ein Doppelpunkt ":",
  - ◇ dann jeweils der Name einer Komponente zusammen mit den Werten für die Argumente des Konstruktors,

    Bei mehreren Komponenten durch Kommata getrennt. Als Argumente kann man z.B. die Parameter des Konstruktors verwenden.
  - ◇ dann der eigentliche Rumpf des Konstruktors.

Konstruktoren: Member (3)

- Beispiel: Klasse `person` hat Komponente vom Typ `date` (Geburtsdatum):

```
class Person {
        char first_name[20];
        char last_name[20];
        Date birthdate;
public:
        Person(const char *fn, const char *ln,
                        int d, int m, int y):
                birthdate(d, m, y)
        {
                ... // fn, ln kopieren
        }
... };
```

Konstruktoren: Member (4)

- Alternative: Namen als string-Objekte:

```
class Person {
        string first_name;
        string last_name;
        Date birthdate;
public:
        Person(const char *fn, const char *ln,
                        int d, int m, int y):
                first_name(fn), last_name(ln),
                        birthdate(d, m, y)
        { // alles schon erledigt
        }
... };
```


Konstruktoren: Member (5)

- Die Konstruktoren der Komponenten werden vor dem Rumpf des Konstruktors aufgerufen,
- und zwar in der Reihenfolge, in der die Komponenten in der Klasse deklariert sind.

  Nicht in der Reihenfolge, in der die Konstruktoraufrufe nach dem Doppelpunkt geschrieben sind. Um Verwirrungen zu vermeiden, sollte man natürlich am besten die gleiche Reihenfolge wählen.
- Bei Bedarf kann man auch zuerst einen Default-Konstruktor für die Komponente aufrufen lassen und sie dann im Rumpf modifizieren.

Notiz zum Programmierstil

- Viele Programmierer schreiben den ersten Buchstaben von Klassennamen groß (wie oben gezeigt).
- Häufiges Problem: Namenskonflikt Attribute (Komponenten) vs. Parameter (z.B. vom Konstruktor).
- Mein Stil ist folgender:
  - Attribute mit großem ersten Buchstaben.
  - Parameter dann entsprechend klein.
  - Klassennamen schreibe ich klein mit Suffix "`_c`".

    Z.B. `person_c`. Über `typedef` (s.u.) definierte Pointer-Typen mit Suffix "`_t`". Z.B. ist `person_t` ein Zeiger auf `person_c`.

Destruktoren (1)

- Wenn ein Objekt gelöscht wird, z.B. weil der Block verlassen wird, in dem es als lokale Variable angelegt wurde, ruft C++ automatisch eine Destruktor-Methode auf.

  Wenn man keine deklariert, legt der Compiler automatisch eine an, die nichts tut (außer ggf. Destruktoren von Komponenten aufzurufen).
- Destruktoren sind dadurch gekennzeichnet, daß ihr Methodenname der Klassenname mit vorangestellter Tilde "~" ist.

  Sie sind ja das Komplement des Konstrukors. Wie bei Konstruktoren gibt es keinen Ergebnistyp.

Destruktoren (2)

- Destruktoren sind zumindest in Anfängerprogrammen eher selten anzutreffen.
- Destruktoren sind aber wichtig, wenn der Konstruktor oder eine andere Methode Resourcen anfordert, z.B. dynamisch Speicherplatz reserviert, eine Datei eröffnet, etc. In diesem Fall muß der Destruktor die Resourcen wieder freigeben.
- Wenn das Objekt z.B. eine kompliziertere Datenstruktur verwaltet (verkettete Liste etc.), sollte der Destruktor die komplette Datenstruktur löschen.

Destruktoren (3)

- Im Zusammenhang mit Assertions (Zusicherungen, s.u.) kann man Destruktoren verwenden, um ein gelöschtes Objekt unbrauchbar zu machen.

  Es könnte ja möglicherweise noch Zeiger auf das Objekt geben, und falls der Compiler bzw. das Laufzeitsystem den Speicherplatz nicht gleich neu verwendet, würden Methodenaufrufe über diese Zeiger zufällig noch funktionieren. Es ist aber eindeutig ein Fehler, auf ein bereits gelöschtes Objekt noch zuzugreifen. Unter anderen Umständen könnte das unüberschaubare Konsequenzen haben (Absturz etc.). Deswegen sollte man den Fehler möglichst sicher bemerken.

- Falls Objekte wechselseitig verzeigert sind, könnte ein Destruktoraufruf den Zeiger auf der anderen Seite löschen (auf 0 setzen).

Klassen-Attribute (1)

- Statische "Data Member" (Komponenten) existieren nur einmal pro Klasse, und nicht einmal pro Objekt (entspricht globaler Variable in der Klasse).
- Man spricht dann auch von Klassenvariablen oder Klassenattributen im Gegensatz zu Instanzvariablen.
- Z.B. kann man so die Anzahl der Objekte einer Klasse mitprotokollieren:
  - ◇ Statische Komponente für die aktuelle Anzahl.
  - ◇ Im Konstruktor erhöht man sie um eins, im Destruktor verringert man sie entsprechend.

Klassen-Attribute (2)

```
class Person {
        static int num_persons;
        string first_name;
        string last_name;
        date birthdate;
public:
        Person(const char* f, const char *l,
                        int d, int m, int y)
                : first_name(f), last_name(l),
                  birthdate(d, m, y)
                { num_persons++; }
        ~Person()
                { num_persons--; }
... };
```

Klassen-Attribute (3)

- Die statischen Attribute sind in der Klasse nur deklariert, aber nicht definiert und initialisiert.
- D.h. es ist so noch kein Speicherplatz für sie reserviert, und man bekommt beim Linken den Fehler "unresolved external symbol".

  Für die normalen Attribute ist nach Deklaration der Klasse auch kein Speicherplatz reserviert, dies geschieht bei der Erzeugung eines Objektes der Klasse. Die statischen Attribute existieren aber unabhängig von Objekten (z.B. auch, wenn noch kein Objekt erzeugt wurde).
- Lösung (außerhalb der Klassendeklaration):

```
int Person::num_persons = 0;
```


Klassen-Attribute (4)

- Man beachte: Bei der Definition der Klassenvariable wird das Schlüsselwort "`static`" nicht wiederholt.

  Sonst bekommt man den Fehler "storage class illegal on members".
- Häufig schreibt man die Klassendeklaration in eine Header-Datei (`.h`), die Definition der Klassenvariable aber in eine `.cxx`/`.cpp`-Datei.

  Die Methodendefinitionen, die nicht bereits Teil der Klassendeklaration oder `inline` sind, kommen auch in die `.cxx`-Datei. Der Punkt ist, daß eine Variable oder Funktion nicht mehrfach definiert sein darf (es wäre dann mehrfach Speicherplatz reserviert), man die Klassendeklaration aber ggf. in verschiedenen Modulen braucht. Die Aufteilung wird im Kapitel über den Linker genauer erklärt. Bei kleinen Programmen kann man alles zusammen in eine `.cxx`-Datei schreiben.

Klassen-Methoden (1)

- Entsprechend kann man Methoden als "`static`" deklarieren:

```
class Person {
        static int num_persons;
        string first_name;
        string last_name;
        date birthdate;
public:
        static int get_num_persons()
                { return num_persons; }
... };
```

- Solche Funktionen haben dann nicht den impliziten Parameter für ein "aktuelles Objekt".

Klassen-Methoden (2)

- Man ruft statische Methoden (Klassen-Methoden) also nicht für ein bestimmtes Objekt, sondern für die Klasse auf, z.B.

```
cout << Person::get_num_persons();
```

  Tatsächlich darf man Sie auch wie gewöhnliche Methoden für ein Objekt aufrufen, aber der Compiler ignoriert dann das Objekt (er benutzt es nur zur Bestimmung der Klasse).
- Daher können statische Methoden natürlich nicht (ohne Angabe eines Objektes) auf normale Komponenten (Instanz-Variablen) zugreifen.
- Sie können nur auf die Klassen-Variablen zugreifen.

Copy-Konstruktoren (1)

- Bei obigem Programm kann die Anzahl `num_persons` der `Person`-Objekte negativ werden!
- Der Grund ist, daß es noch einen anderen implizit angelegt Konstruktor gibt, den Copy-Konstruktor.
- Dieser initialisiert ein Objekt, indem er alle Attribute eines anderen Objektes kopiert, aber er erhöht die Anzahl der Objekte `num_persons` bisher nicht.
- Es gibt dagegen nur einen Destruktor, der auch für die so konstruierten Objekte aufgerufen wird.

Copy-Konstruktoren (2)

- Lösung: Man muß den Copy-Konstruktor selbst explizit definieren und auch darin die Anzahl der Objekte erhöhen:

```
class Person {
        ...
public:
        Person(const Person& P) :
                first_name(P.first_name),
                last_name(P.last_name),
                birthdate(P.birthdate)
                { num_persons++; }
... };
```


Copy-Konstruktoren (3)

- Es kann für eine Klasse mehrere Konstruktoren mit unterschiedlichen Eingabetypen geben.
- Der Copy-Konstruktor zeichnet sich dadurch aus, daß er eine konstante Referenz auf ein Objekt der gleichen Klasse als Eingabe hat.
- Im Gegensatz zum Default-Konstruktor wird der Copy-Konstruktor auch dann automatisch angelegt, wenn man explizit einen Konstruktor deklariert hat.
- Man kann das nur vermeiden, indem man selbst einen Copy-Konstruktor deklariert.

Copy-Konstruktoren (4)

- Der Copy-Konstruktor wird z.B. verwendet, wenn ein Objekt “by value” an eine Funktion übergeben wird.

  Das ist ineffizient. Besser: Konstante Referenz oder Zeiger mit `const`.
- Für den formalen Parameter wird ein temporäres Objekt erstellt, das durch den Copy-Konstruktor aus dem aktuellen Parameter initialisiert wird.
- Am Ende der Funktion wird der Destruktor dafür aufgerufen.
- Entsprechend: Rückgabe von Objekten.

Zuweisungen (1)

- Bei Zuweisungen, die eine Initialisierung sind, wird der Copy-Konstruktor verwendet:
```
Person p("Antonio", "Vivaldi", 4, 3, 1678);
Person p2 = p; // Copy Construktor
```
- Dagegen müssen Zuweisungen an bereits initialisierte Objekte anders behandelt werden:
```
Person p("Antonio", "Vivaldi", 4, 3, 1678);
Person p2("Johann Sebastian", "Bach", 21, 3, 1685);
p2 = p; // Zuweisung
```
- Würde hier nur der Copy-Konstruktor aufgerufen, so würde die Objektzählung nicht funktionieren.

Zuweisungen (2)

- Wenn schon, müßte erst der Destruktor für den alten Wert von `p2` aufgerufen werden, und dann der Copy-Konstruktor.
- Tatsächlich wird aber eine spezielle Zuweisungs-Methode aufgerufen, die auch automatisch definiert wird, wenn man sie nicht explizit definiert.
- Die automatisch erzeugte Methode weist komponentenweise zu, das ist in diesem Fall richtig.

  Probleme würde es z.B. geben, wenn die Klasse einen Zeiger auf dynamisch angeforderten Speicher enthalten würde. Dann wäre eine einfache Zuweisung nicht richtig, s.u.

Zuweisungen (3)

- Bei Bedarf kann man die Zuweisungs-Operation auch selbst definieren:
```
class Person {
        ...
        Person& operator=(const Person& p)
        {
                if(this != p) { // wegen p = p
                        ... // Komponenten zuweisen
                }
                return *this;
        }
        ...
};
```


Implizite Methoden (1)

- Wenn man
```
class C { int n; };
```
schreibt, ist dies also äquivalent zu
```
class C {
private:
        int n;
public:
        C() {};         // Default Konstruktor
        C(const C& x) // Copy Konstruktor
                { n = x.n; }
        ~C() {}; // Destruktor
        C& operator=(const C& x) // Zuweisung
                { n = x.n; }
};
```

Implizite Methoden (2)

- Manchmal sollte es die von C++ automatisch generierten Methoden aber besser nicht geben.
- Z.B. werden Klassen manchmal auch im Sinne klassischer Module verwendet, wenn man zusammengehörige Daten und Prozeduren kapseln möchte, aber es keinen Sinn machen würde, mehrere Objekte dieser Klasse anzulegen.
- Dann wären typischerweise alle Daten und Funktionen in der Klasse als `static` deklariert.

Implizite Methoden (3)

- Wenn es keinen Sinn macht, ein Objekt dieser Klasse anzulegen, sollte ein solcher Versuch zu einer Fehlermeldung beim Compilieren führen.

  Oder spätestens beim Linken. Allgemein gilt: Je früher ein Fehler bemerkt wird, desto billiger ist seine Beseitigung.

  Man muß immer damit rechnen, daß eine Klasse später in einem anderen Programm wiederverwendet wird, das ist ja gerade ein Ziel der objektorientierten Programmierung. Zu der Zeit überblickt der Programmierer die Einschränkungen aber nicht mehr so genau wie jetzt, wenn man die Klasse entwickelt und vermutlich selbst verwendet. Es können dann also leichter Fehler geschehen. Man sollte so robust programmieren, daß solche Fehler sofort bemerkt werden.

Implizite Methoden (4)

- Ein Trick, die Erzeugung von Objekten zu verhindern, ist, eine Konstruktor-Methode (z.B. ohne Parameter) zu deklarieren, aber als "`private:`".

  Ohne Objekte sind Copy-Konstruktor/Zuweisung nicht anwendbar.
- Ein Aufruf von außen würde dann zu einer Fehlermeldung führen.
- Außerdem gibt man keinen Rumpf (Implementierung) für den Konstruktor an (nur Deklaration).
- Dann würde ein Aufruf aus Methoden der Klasse selbst zu einem Fehler beim Linken führen.

Implizite Methoden (5)

- Den gleichen Trick kann man verwenden, wenn man zwar Objekte der Klasse erlauben will, das Kopieren von Objekten aber verhindern.

  Z.B. wenn das Objekt eine bestimmte Resource verwaltet (etwa dynamisch angeforderten Hauptspeicher), kann man durch Objektkopien mit der Freigabe dieser Resource durcheinander kommen. Zumindest wird es dann komplizierter, und wenn man sowieso nur über Zeiger und Referenzen mit den Objekten arbeitet, sollte man den Copy-Constructor und den Zuweisungsoperator unbenutzbar machen.

Konstanten (1)

- Beispiel: Klasse `Person` mit C-Strings:

```
class Person {
        char first_name[20];
        char last_name[20];
... };
```

- Explizit angegebene Werte (außer 0 und 1) sind meistens schlechter Stil.
- Der Wert 20 ist ganz willkürlich gewählt. Es ist sehr wohl möglich, daß man ihn später ändern möchte.
- Dann ist es aber schwierig, alle Stellen zu finden, die man ändern muß (z.B. ggf. auch 19, 40, etc.).

Konstanten (2)

- Natürlich könnte man eine Konstante außerhalb der Klasse deklarieren:

```
const int NAME_LENGTH = 20;
```

- Falls die Konstante nur im Zusammenhang mit der Klasse eine Bedeutung hat, ist das nicht schön.
- Die theoretisch saubere (aber etwas komplizierte) Lösung ist ein konstantes Klassen-Attribut:

```
class Person {
        static const int NAME_LENGTH = 20;
        char first_name[NAME_LENGTH];
        char last_name[NAME_LENGTH];
... };
```


Konstanten (3)

- In diesem Fall reicht die Deklaration der Konstanten leider nicht, und man braucht noch eine getrennte Definition (wie für andere Klassenvariablen, aber hier ohne Initialisierung):

```
const int Person::NAME_LENGTH;
```

  Das wirkt unnötig umständlich.
- Übrigens funktioniert diese Lösung ausschließlich für Konstanten von ganzzahlige Typen, z.B. nicht für `float`-Konstanten.

Konstanten (4)

- Ein Trick ist die Deklaration eines Aufzählungstyps mit einer einzigen Konstanten:

```
class Person {
        enum{ NAME_LENGTH = 20 };
        char first_name[NAME_LENGTH];
        char last_name[NAME_LENGTH];
... };
```

- Der Aufzählungstyp selbst hat keinen Namen, man kann ihn also nicht verwenden (wie beabsichtigt).
- Falls man den Typ `public` deklariert hat, kann man die Konstante mit `Person::NAME_LENGTH` ansprechen.

Konstante Attribute (1)

- Manche Attribute werden im Konstruktor initialisiert und dann nie mehr geändert.
- Es ist immer gut, wenn der Compiler solche Zusicherungen überwacht.

  Wenn es nur ein Kommentar ist oder in der Dokumentation steht, könnte ja doch bei späteren Änderungen dagegen verstoßen werden, und andere Programmteile (die sich auf die Aussage verlassen haben) funktionieren dann vielleicht nicht mehr zuverlässig.
- Man kann daher Attribute als `const` deklarieren:

```
class Person {
        const int id;
... };
```


Konstante Attribute (2)

- Die Initialisierung von konstanten Attributen muß im Konstrukor geschehen:

```
class Person {
      const int id;
      string first_name;
      string last_name;
public:
      Person(const char* f, const char *l, int n)
            : id(n), first_name(f), last_name(l)
              { }
... };
```

- Die sonst nur für Komponenten-Objekte übliche Syntax funktioniert für beliebige Attribute.

friend-Deklarationen (1)

- Eine normale Funktionsdeklaration in einer Klasse (Methoden-Deklaration) bewirkt drei Dinge:
  - ◇ Die Funktion bekommt Zugriff auf die privaten Komponenten der Klasse.
  - ◇ Die Funktion wird im Namensraum der Klasse deklariert.
  - ◇ Die Funktion hat ein spezielles Argument “`this`”.
- Bei statischen Funktionen entfällt der dritte Punkt.
- Mithilfe von `friend`-Deklarationen kann man auch nur den ersten Punkt haben.

friend-Deklarationen (2)

```
class A {
    private:
        int n;
        friend int f();
        friend int B::g(A* y);
};
A x;

int f() { return x.n; }

class B {
    public:
        int g(A* y) { return y->n; }
};
```

friend-Deklarationen (3)

- One die `friend`-Deklarationen würde man folgende Fehlermeldung bekommen: 'n': cannot access private member declared in class 'A'.
- Mit den `friend`-Deklarationen sind die Zugriffe dagegen möglich.
- Alternative:

```
class A { ... friend class B; ...}
```

Dann haben alle Methoden in Klasse `B` Zugriff auf die privaten Komponenten von `A`.

friend-Deklarationen (4)

- Allgemein sollte man die Menge der Funktionen, die Zugriff auf ein Datenelement haben, möglichst klein halten, um
  - ◇ Fehler schneller zu finden, und

    Wenn man von Zuweisungen über ungültige Pointer absieht (die beliebige Dinge im Speicher treffen können), muß der falsche Wert ja von einer Funktion zugewiesen sein, die Zugriff auf das Datenelement hatte. Natürlich ist es möglich, daß diese Funktion mit einem ungültigen Parameterwert aufgerufen wurde, den hätte sie aber eigentlich melden sollen.
  - ◇ Änderungen leicher durchführen zu können.

    Wenn man das Datenelement löscht oder die Repräsentation des Wertes darin ändert, können zunächst nur die Funktionen betroffen sein, die Zugriff darauf hatten.

Exkurs/Anhang: typedef

- Mit `typedef` kann man einen Namen für einen Typ deklarieren (ähnlich wie Variablendeklaration):

```
typedef Person *person_t;
```

- Nun ist `person_t` im wesentlichen eine Abkürzung für `Person *`.
- Man kann z.B. zwei Variablen deklarieren:

```
person_t x;
Person *y;
```

Die beiden Variablen `x` und `y` haben den gleichen Typ (können einander zugewiesen werden).