**Software-Technik: Vom Programmierer zur erfolgreichen …**

# 2. Funktionen und Datenstrukturen

2.1 Funktionale Abstraktion (Funktionen)
2.2 Funktionale Abstraktion bei der Netzplanung
2.3 Datenabstraktion: Strukturierte Datentypen
2.4 Generische Programmierung, 1. Teil
2.5 Zusammenfassung

Lehrbuch: 1.5
Kompendium, 3. Auflage: 1.8
Kompendium, 4. Auflage: 1.6

Folien mit gelben Punkten ● am oberen rechten Rand sind weniger wichtig für das Verständnis der nachfolgenden Kapitel.

## Variablen und Speicher

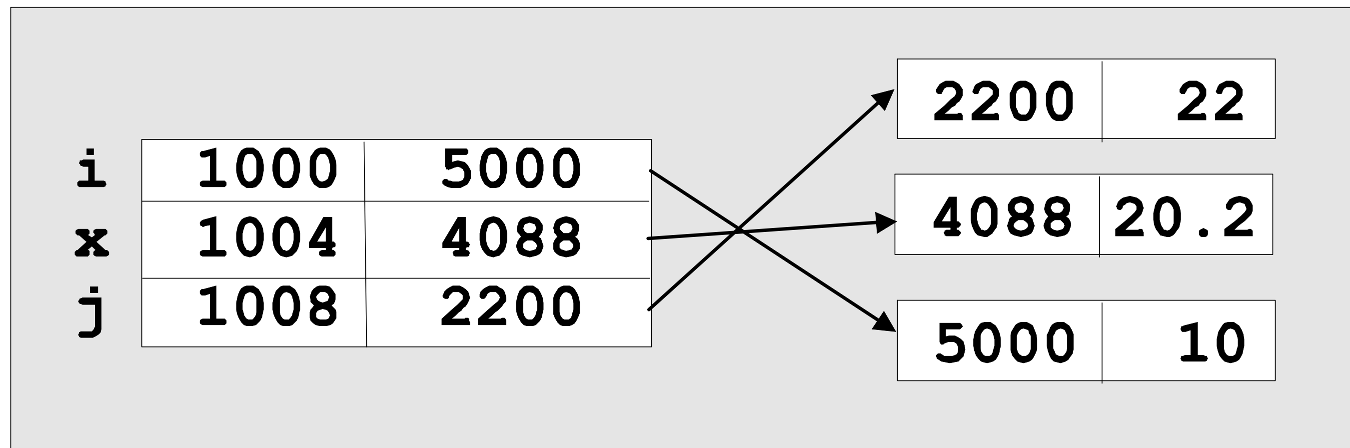
Die Variablen der einfachen Datentypen wie int oder double speichern direkt ihre Werte.
Eine Variable ist an einer bestimmten Speicherstelle im Arbeitsspeicher des Rechners abgelegt.

```
int i = 10;
double x = 20.2;
int j = 22;
```

| | Adresse | Wert |
|---|---|---|
| i | 1000 | 10 |
| x | 1004 | 20.2 |
| j | 1012 | 22 |

## Referenzen

In Zusammenhang mit benutzerdefinierten Datentypen sind insbesondere Referenzen von zentraler Bedeutung.
In einem Referenztyp wird im Gegensatz zu den einfachen Variablen nur eine Referenz auf den eigentlichen Wert der Variablen gespeichert.
Wären die Variablen i, x und j aus dem vorherigen Beispiel Referenzen, könnte die Speicherbelegung wie folgt aussehen.




Die Speicherstelle 1000 enthält somit nicht mehr direkt den Wert der Variablen i, sondern einen Verweis auf die Speicherstelle, die den Wert enthält; in diesem Fall steht der Wert von i in der Speicherstelle 5000.

## Strukturierte Datentypen

Bisher haben wir einfache Datentypen, Steueranweisungen und Funktionen kennen gelernt.

Hiermit ist es zwar zumindest im Prinzip möglich, jedes Programm zu schreiben, aber für größere Programme wäre dieses zumindest sehr, sehr mühselig, weil man z.B. für jede Variable einen eigenen Namen vergeben müsste. Wir werden deshalb in diesem Abschnitt zwei Verbesserungen kennen lernen:

- Ein Array/Vektor ist eine homogene Datenstruktur, die verschiedene Datenelemente gleichen Typs zusammenfasst; ein einzelnes Datenelement wird dann über einen Index als Selektor ausgewählt.
- Eine Struktur (Record) ist hingegen eine im Allgemeinen inhomogene Datenstruktur, die verschiedene Datenelemente unterschiedlichen Typs zusammenfassen kann; ein einzelnes Datenelement (häufig Komponente, Attribut oder Mitglied, engl. member genannt) wird dann über einen symbolischen Komponentennamen als Selektor ausgewählt.

# Strukturen

## Vorgang als struct / Record

C++

```
struct Vorgang {
   double dauer;
   double fruehanf;
   double spaetend;
};
```

Java

```
public class Vorgang {
   double dauer;
   double fruehanf;
   double spaetend;
}
```

C++

```
struct Netz {
   enum {MAX = 100};
   double startzeit, endzeit;
   int anzahl;
   Vorgang vorg[MAX];
   bool nachf[MAX][MAX];
};
```

Java

```
public class Netz {
   static final int MAX=100;
   double startzeit,endzeit;
   int anzahl;
   Vorgang vorg[]=new Vorgang[MAX];
   boolean nachf[][]=new
            boolean[MAX][MAX];
  }
```

## Strukturen / Klassen

C++

```
struct Vorgang {
   double dauer;
   double fruehanf;
   double spaetend;
};
```

```
class Vorgang {
   double dauer;
   double fruehanf;
   double spaetend;
};
```

Strukturen sind (in C++) das gleiche wie Klassen. Allerdings sind in Strukturen per Default alle Attribute (wie dauer) public und in Klassen alle Attribute in Methoden per Default private.

Auch Strukturen könnten Methoden haben.

Man verwendet Strukturen eigentlich nur als inner-class, auf die von außen dann kein Zugriff besteht.

Strukturen mit öffentlichen Attributen sind schlechter Programmierstil. Sind die Attribute nicht öffentlich, sollten besser gleich Klassen verwendet werden.

## Strukturen / Klassen

```
struct Vorgang {
   double dauer;
   double fruehanf;
   double spaetend;
};
```

Entspricht:

```
class Vorgang {
public:
   double dauer;
   double fruehanf;
   double spaetend;
};
```

```
struct Vorgang {
private:
   double dauer;
   double fruehanf;
   double spaetend;
};
```

Entspricht:

```
class Vorgang {
   double dauer;
   double fruehanf;
   double spaetend;
};
```

```
struct Vorgang {
public:
   double getDauer()
      {return dauer;}
private:
   double dauer;
};
```

Entspricht:

```
class Vorgang {
public:
   double getDauer()
      {return dauer;}
private:
   double dauer;
};
```

# Beispiel für ein Record / eine Struktur

```
typedef char Name[32];
struct Student {
   Name NachName, Vorname, Fach;
   int    MatNr:
   double Ergebnis;
   };

Student  s1, s2, s3;
```

Zugriff auf die einzelnen Record-Felder:

```
strcpy(s1.NachName, "Meier");
s2.MatNr = 1234567;
```

# Beispiel für ein Record / eine Struktur

```
typedef char Name[32];
struct Student {
   Name NachName, Vorname, Fach;
   int    MatNr:
   double Ergebnis;
   };

Student  s1, s2, s3;
```

Zugriff auf die einzelnen Record-Felder:

```
strcpy(s1.NachName, "Meier");
s2.MatNr = 1234567;
```

## Clicker-“Abstimmung“

```
struct Student {
   int    matNr;
   double ergebnis;
   };
struct Uni {
   Student viele[20];
   int anz;
};
Student  s1;
Uni ostfalia;
```

Was ist syntaktisch korrekt?

1. s1.matNr = 11;
2. matNr = 11;
3. matNr(s1) = 11;
4. 1 und 2
5. 2 und 3

Was ist syntaktisch korrekt?

1. ostfalia.viele = s1;
2. ostfalia.viele[1] = s1;
3. ostfalia.Student[2] = s1;
4. ostfalia.viele[1].matNr=s1;
5. 1 und 3

## Aufgabe

Implementieren Sie eine Datenstruktur Vektor:

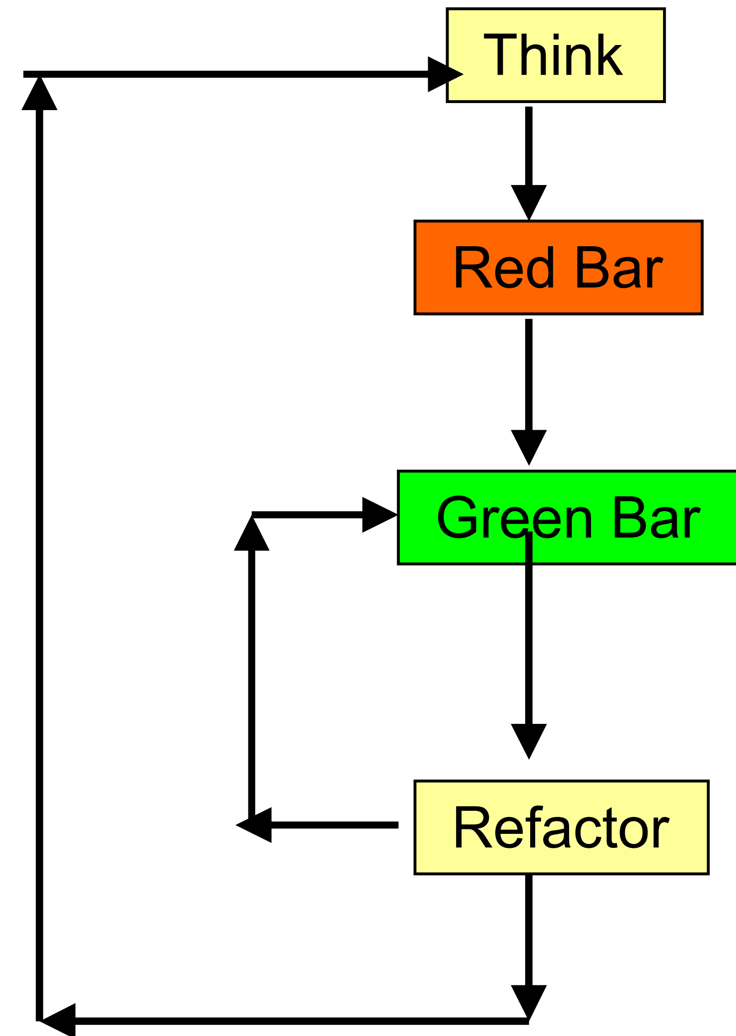
```
struct Vektor{
   int daten[10];
   int laenge; // wieviel der 10 wirklich belegt sind
};
```

Implementieren Sie noch die beiden Funktionen, die alle Elemente des Vektors mit wert initialisieren, bzw. Plus addiert die Werte zweier Vektoren.
Gehen Sie nach „test first“ vor und implementieren Sie automatisch ablaufende Tests, die die Funktionalität zeigen.

```
void Init(Vektor& v1, int wert);
void Plus(… );
```

Für die Schnellen, das gleiche für eine Matrix mit zanz und spanz. Die Daten werden in einem eindimensionalen Array intern gespeichert. Index wird aus Zeilen-Index und Spalten-Index berechnet.

## Testbasierte Software-Entwicklung



## Funktionale Abstraktion

Dieses Refactoring führen wir nun nach und nach für alle Funktionen durch.
„Never change a running system“ gilt nicht.
Wir haben automatisch ablaufende Tests.

Wir dokumentieren die neue Funktion zuerst.
Dann schreiben wir noch einen Test für die Funktion oder auch mehrere.

Anschließend implementieren wir die Funktion und ersetzen den Code durch den Funktionsaufruf.

Wir führen die Tests aus
--- und hoffen --- !!!
Nein, es ist ein systematisches Vorgehen, das zwar auch nicht fehlerfrei ist, aber ziemlich sicher die Fehler aufdeckt.

## Refactoring

Von Refactoring spricht man allgemein, wenn man Änderungen am Code vornimmt, ohne seine Funktionalität zu verändern. Refactoring dient somit lediglich dazu, Code übersichtlicher, wartungsfreundlicher und damit weniger fehleranfällig zu machen.

Refactoring wird aber meist einer Erweiterung der Funktionalität vorgeschaltet.Es gibt neben der Aufspaltung einer großen Funktion in kleinere Funktionen weitere Refactoring-Techniken.

Wir werden im Folgenden noch einige kennen lernen. Details können aber dem Lehrbuch von Fowler zum Refactoring entnommen werden.

Martin Fowler: Refactoring -- Improving the Design of Existing Code, Addison Wesley, 2002

# Lösung (2)

```
/** Es wird ein Vektor erzeugt und mit dem Wert 81
gefüllt. Anschließend wird geprüft, ob auch jedes
Element den Wert 81 hat.
*/
bool testInit() {
   Vektor v;
   v.laenge=4;
   const int wert=81;
   Init(v, wert);
   for (int i=0; i<v.laenge; ++i)   {
      if (v.daten[i] != wert) {
         return false;
      }
   }
   return true;
}
```

```
/** Es werden zwei Vektoren erzeugt und mit
den Werten 81 und -10 belegt. Nach Aufruf
von Add sollte der ergebnis-Vektor den
Wert 71 enthalten.
*/
bool testPlus() {
   Vektor v1;
   v1.laenge=4;
   const int wert1=81;
   Init(v1, wert1);

   Vektor v2;
   v2.laenge=4;
   const int wert2= -10;
   Init(v2, wert2);

   Vektor result;
   result.laenge=4;
   Plus(v1,  v2, result);

   for (int i=0; i<result.laenge; ++i)   {
      if (result.daten[i] != (wert1+wert2)) {
         return false;
      }
   }
   return true;
```



überspringen

# Lösung

```
struct Vektor{
   int daten[10];
   int laenge;
};

void Init(Vektor& v1, int wert) {
   for (int i=0; i<v1.laenge; ++i){
      v1.daten[i] = wert;
   }
}

void Plus(const Vektor& v1, const Vektor& v2, Vektor& erg) {
   assert(v1.laenge == v2.laenge);
   assert(erg.laenge == v2.laenge);
   for (int i=0; i<v1.laenge; ++i){
      erg.daten[i] = v1.daten[i] + v2.daten[i];
   }
}
```



überspringen

| Schritt 0 | Schritt 2a | Schritt 2b | Schritt 3 |
| --- | --- | --- | --- |

**Schritt 0**

funk()

xxxxx
xxx

yyyyyy
yyy

bbbb
bbbbb

zzzz
zzzzzz

xxxxx
xxx

yyyyyy
yyy

bbbb
bbbbb

zzzz
zzzzzz

**Schritt 2a**

f1()

xxxxx
xxx

f2()

yyyyyy
yyy

f7()

bbbb
bbbbb

f8()

zzzz
zzzzzz

funk()

f1();f2();f3();f4();
f5();f6();f7();f8();

**Schritt 2b**

f()

xxxxx
xxx

yyyyyy
yyy

bbbb
bbbbb

zzzz
zzzzzz

funk()

f();
f();

**Schritt 3**

f1()

xxxxx
xxx

f2()

yyyyyy
yyy

f3()

bbbb
bbbbb

f4()

zzzz
zzzzzz

f()

f1();f2();
f3();f4();

funk()

f();
f();

## Refactoring -- Nachtrag

Refactoring dient somit „*lediglich*“ dazu, übersichtlichen, wartungsfreundlichen und damit weniger fehleranfälligen Code zu erhalten. Genau das haben wir hier durchgeführt.

Eine Art des Refactoring ist die einfache Aufspaltung einer großen Funktion in mehrere kleinere Funktionen – im Original wird jede dieser Funktionen einmal aufgerufen.

.

## Refactoring – Nachtrag (2)

Eine zweite Art des Refactoring ist die Zusammenfassung gleichartiger Funktionalität zu einer Funktion, die dann an mehreren Stellen aufgerufen wird.

Das haben wir bspw. bei der Funktion plane praktiziert, die nun in jedem Test aufgerufen wird.

Wenn man beide Möglichkeiten zur Auswahl hat, welche soll man dann zuerst anwenden?

Die Antwort ist schwierig und einfach zugleich. Man sollte es gar nicht erst so weit kommen lassen, sondern viel früher eine Aufspaltung in Funktionen durchführen, und nicht erst, wenn es gar nicht mehr anders geht und die Funktion Hunderte von Codezeilen enthält.

Wenn man dann aber doch beide Möglichkeiten hat, ist es meist gleichgültig, womit man beginnt.

Man sollte die Schritte allerdings nacheinander durchführen, da sonst die Gefahr des versehentlichen Einbaus von Fehlern zu groß ist.

# Instanziierung einer Klasse / einer Struktur auf dem Stack

C++

```
double d1;
int j;
 // ganz entsprechend auch
Vorgang v1;
Vorgang v2;
```

Java

nicht vorgesehen

Entsprechend der Erzeugung von Standardtypen wie int und double wird hier der Wert (keine Verweis) direkt auf dem Stack abgelegt.

```
struct Vorgang {
   double dauer;
   double fruehanf;
   double spaetend;
};
```

| | | |
|---|---|---|
| 5000 | v1.dauer | 5 |
| | v1.fruehanf | 0 |
| | v1.spaetend | 7 |
| 5024 | v2.dauer | 2 |
| | v2.fruehanf | 2 |
| | v2.spaetend | 5 |

# Weiteres zu Strukturen

Arrays und Records dürfen geschachtelt werden, d.h. Folgendes ist erlaubt:

```
struct Complex { double re, im; };
Complex  feld[12];

struct Person {
   char Name[15];
   char VorName[15];
   };

struct Firma {
         Person Chef;
         Person Putzfrau;
         Person Mitarbeiter[10];
         };
Firma Konzern[10];
```

# Weiteres zu Strukturen (2)

```
struct Person {
    char Name[15];      char VorName[15];
    };
struct Firma {
          Person Chef;
          Person Putzfrau;
          Person Mitarbeiter[10];
          };

Person p1;    Firma fhwf;
strcpy(p1.Name, "Boss");
fhwf.Chef = p1;
fhwf.Mitarbeiter[4] = p1;
```

# Weiteres zu Strukturen (3)

```
struct Complex {
        double re, im;
        };

struct Meter{double re; };
struct Fuss{double re; };
struct Kilo{double gewicht; };
```

```
Complex c1, c2:;
Meter m;
Fuss f;
Kilo k;
```

```
c1 = c2;      (* erlaubt *)
c1 = m;       (* verboten *)
f = k;        (* verboten *)
m = f;        (* verboten *)
f = m;        (* verboten *)
```

# Zuweisung von Strukturen als Ganzes, Typgleichheit

Strukturen können initialisiert und als Ganzes zugewiesen werden:

```
typedef struct { double x, y; } Complex;
Complex a = { 12.34, 56.78 };
Complex b = a;
Complex c;
c = b;
```

Die beiden Typen sind gleich:

```
typedef double Meter;
typedef double Kilogramm;
Meter       x = 12.34;
Kilogramm y;
y = x;                    /* syntaktisch richtig */
```

# Zuweisung von Strukturen als Ganzes, Typgleichheit

Strukturen können initialisiert und als Ganzes zugewiesen werden:
Die beiden Typen sind gleich:

```
struct Meter {double x;};
struct Kilogramm {double wert;};
Meter        x.x = 12.34;
Kilogramm y.wert = 14.0;
y = x;                      /* syntaktisch falsch */
```

## Clicker-“Abstimmung“

```
struct Student {
   int mat_nr;
   double gewicht;
};
```

```
void init(Student & s) {
  s.mat_nr = 44;
  s.gewicht= 21;
}
void testInit() {
  Student s1;
  s1.mat_nr = 11;
  init(s1);
  cout << s1.mat_nr;
}
```

Wie ist die Bildschirmausgabe?

1. 44
2. 21
3. 11
4. 4

_-_ 44 _ 21 _ 11 __ 4

## Clicker-“Abstimmung“

```
struct Student {
   int mat_nr;
   double gewicht;
};
```

```
void init(Student s) {
  s.mat_nr = 44;
  s.gewicht= 21;
}
void testInit() {
  Student s1;
  s1.mat_nr = 11;
  init(s1);
  cout << s1.mat_nr;
}
```

Wie ist die Bildschirmausgabe?

1. 44
2. 21
3. 11
4. 4

__ 44   __ 21   _-_ 11   __ 4

## Gemeinsame Aufgabe

Teilen Sie den Code zur Implementierung des struct Vektor auf verschiedene Dateien Vektor.h, Vektor.cxx; Test.h; Test.cxx und main.cxx auf.

Für die Schnellen: Entsprechendes für die „struct Matrix“