

Software Paradigmen

2



I. Software Paradigmen

Inhalt

1 Prinzipien und Paradigmen des Softwareentwurfs

- Übergeordnete Architekturprinzipien
- Allgemeine Programmierparadigmen

2 Objektorientierte Programmierung

- Paradigmen der objektorientierten Programmierung
- Entwurfsmuster

3 Funktionale Programmierung

- Paradigmen der funktionalen Programmierung
- Container
- Iteratoren und Algorithmen
- Vertiefung: Funktoren
- Vertiefung: Rekursion

4 Generische / Generative Programmierung

- generische Programmierung
- generative Programmierung

3



I. Software Paradigmen

Quellen

B. Lahres, G. Rayman, S. Strich
Objektorientierte Programmierung [Kap 2,3,7]
Rheinwerk Computing
<http://openbook.rheinwerk-verlag.de/oop/>



B. Vogel, I. Arnold, A. Chughtai, E. Ihler, T. Kehrer,
U. Mehlig, U.Zdun
Software-Architektur [Kap 6]
Spektrum Akademischer Verlag



G. Pomberger, H. Dobler
Algorithmen und Datenstrukturen [Kap 4,7,9]
Pearson Studium



4



I. Software Paradigmen

Quellen

S. Dreiseitl
Mathematik für Software Engineering [Kap A]
Springer Verlag



D. W. Hoffmann
Theoretische Informatik [Kap 6]
Hanser



M. Block, A. Neumann
Haskell Intensivkurs
Springer



U. Breymann
Der C++ Programmierer
Hanser





1 Prinzipien und Paradigmen des Softwareentwurfs

- Übergeordnete Architekturprinzipien
- Allgemeine Programmierparadigmen

2 Objektorientierte Programmierung

- Paradigmen der objektorientierten Programmierung
- Entwurfsmuster

3 Funktionale Programmierung

- Paradigmen der funktionalen Programmierung
- Container
- Iteratoren und Algorithmen
- Vertiefung: Funktoren
- Vertiefung: Rekursion

4 Generische / Generative Programmierung

- generische Programmierung
- generative Programmierung



1 Prinzipien und Paradigmen des Softwareentwurfs

- Übergeordnete Architekturprinzipien
- Allgemeine Programmierparadigmen

2 Objektorientierte Programmierung

- Paradigmen der objektorientierten Programmierung
- Entwurfsmuster

3 Funktionale Programmierung

- Paradigmen der funktionalen Programmierung
- Container
- Iteratoren und Algorithmen
- Vertiefung: Funktoren
- Vertiefung: Rekursion

4 Generische / Generative Programmierung

- generische Programmierung
- generative Programmierung



Prinzipien sind Leitlinien für eine gute Softwareentwicklung, um in der Komplexität eines wachsenden Softwareprojekts den Überblick zu behalten, die Software verständlich, wartbar und erweiterbar zu machen.

Im Folgenden werden wir eine Auswahl dieser Prinzipien kennenlernen. Der Begriff **Modul** bezeichnet einen Codeabschnitt, der aus Sicht des Entwicklers als eigenständige, zusammengehörige Einheit betrachtet wird.



DON'T REPEAT YOURSELF

Eine Quelltextsequenz mit einer eigenständigen Funktionalität sollte nur an einer einzigen Stelle im Quellcode abgelegt sein.

In einer nach diesem Prinzip gestalteten Software brauchen notwendige Änderungen nur an einer Stelle vorgenommen werden.

BEISPIEL |

- wiederkehrende Zahlen im Quelltext als Konstanten vereinbaren und über ihren Namen verwenden
- wiederkehrende Kombinationen von Variablen/Werten als Funktionsaufruf bereitstellen
- generell: wiederkehrende Anweisungssequenzen als Funktionen/Methoden bereitstellen



1.1a Übergeordnete (Architektur-)Prinzipien

Entkopplung von Verantwortung

SINGLE RESPONSIBILITY PRINCIPLE (ROBERT MARTIN, 1974)

Jedes Modul trägt die Verantwortung für genau eine einzige (Teil-)Funktionalität der Software.

„[...]each software module should have one and only one reason to change“

↑ R. Martin „

- Jedes Modul ist genau einer Aufgabe zugeordnet, so dass Änderungen in den Anforderungen sehr einfach den zu ändernden Modulen zugeordnet werden können
- Charakteristisch für dieses Prinzip ist, dass alle Teil des Moduls eng miteinander zusammenhängen/wirken, es gibt keine davon unabhängigen Codebestandteile im Modul

↑ Prinzip der hohen Kohäsion „

K ist nicht für die Ausgabe verantwortlich:

```
class K {
    string s
public:
    K(): s("Hi!") {} // nicht: { cout << s; }
    string toString() { return s; }
};
```

```
class Printer {
public:
    virtual void print(const string&);
};
class HTMLPrinter: public Printer {
public:
    void print(const string& s) { /* HTML-Ausgabe */ }
};
```

10



I.1a Übergeordnete (Architektur-)Prinzipien

Entkopplung von Verantwortung

FIB-ITERATIV-PRINT

DIE FIBONACCI-FOLGE

- Es gibt ein Paar neugeborene Kaninchen
- Jedes neugeborene Paar wird im zweiten Lebensmonat geschlechtsreif
- Jedes geschlechtsreife Paar wirft pro Monat ein weiteres Paar

SRP verletzt.

```
void fib_(unsigned int n) {
    int i,g,f1,f2;
    f1=0; // reproduktionsfähige Paare
    f2=1; // Gesamtzahl Paare
    cout << f2 << endl; // SRP verletzt!
    for(i=1;i<n;++) { // nach dem i-ten Monat:
        g=f1+f2; // Berechnung neue Gesamtzahl
        f1=f2; // reproduktionsfähige Paare
        f2=g; // Gesamtzahl Paare
        cout << f2 << endl; // SRP verletzt!
    } // Ausgabe:
} // 1 1 2 3 5 8 13 21 34 55
```

Kontrollflussdiagramm zur Veranschaulichung der Verletzung des Prinzips der Entkopplung von Verantwortung



I.1a Übergeordnete (Architektur-)Prinzipien

Entkopplung von Verantwortung

FIB-ITERATIV-SRP

DIE FIBONACCI-FOLGE

- Es gibt ein Paar neugeborener Kaninchen
- Jedes neugeborene Paar wird im zweiten Lebensmonat geschlechtsreif
- Jedes geschlechtsreife Paar wirft pro Monat ein weiteres Paar

SRP erfüllt, Laufzeitnachteile



12



I.1a Übergeordnete (Architektur-)Prinzipien

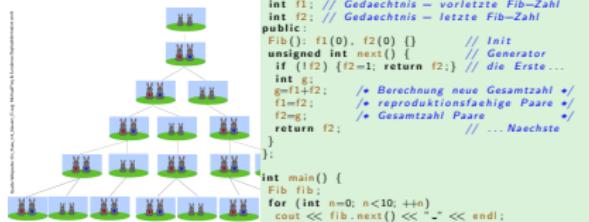
Entkopplung von Verantwortung

FIB-GENERATIV, GENERATOR

DIE FIBONACCI-FOLGE

- Es gibt ein Paar neugeborene Kaninchen
- Jedes neugeborene Paar wird im zweiten Lebensmonat geschlechtsreif
- Jedes geschlechtsreife Paar wirft pro Monat ein weiteres Paar

SRP mittels Generator erfüllt



Kontrollflussdiagramm zur Veranschaulichung der Erfüllung des Prinzips der Entkopplung von Verantwortung



I.1a Übergeordnete (Architektur-)Prinzipien

Aufgabentrennung

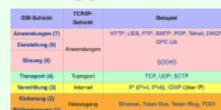
SEPARATION OF CONCERNS (EDSGER DIJKSTRA, 1974)

Aufgabenbereiche sollen in Modulen gekapselt werden und nicht über mehrere Module verstreut werden.

- Modularisierung/Kapselung (z.B. in Klassen/Pakete)
- Trennung von Anwendungslogik/Modell und Darstellung/View (Model-View-Controller, vgl. Folie 47)

BEISPIEL | Protokoll-Schichtenmodelle, z.B. TCP/IP

Jede Schicht kann unabhängig von den anderen betrachtet werden.



I.1a Übergeordnete (Architektur-)Prinzipien

Aufgabentrennung

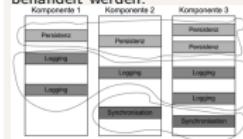
SEPARATION OF CONCERNS (EDSGER DIJKSTRA, 1974)

Aufgabenbereiche sollen in Modulen gekapselt werden und nicht über mehrere Module verstreut werden.

- Modularisierung/Kapselung (z.B. in Klassen/Pakete)
- Trennung von Anwendungslogik/Modell und Darstellung/View

WARNUNG | Übergreifende Aspekte

Aspekte, wie Persistenz, Logging sind „Crosscutting“, müssen durch besondere Programmiertechniken (Aspektorientierte Programmierung) behandelt werden.



Quelle: Nagel, Architektur Software-Architektur



I.1a Übergeordnete (Architektur-)Prinzipien

OOP-Prinzipien

OPEN-CLOSED PRINCIPLE (BERTRAND MEYER, 1988)

Module sollen offen für Erweiterungen aber geschlossen für Veränderungen sein

PROGRAM TO INTERFACES, NOT IMPLEMENTATIONS

Trennung von Schnittstelle und Implementation
Information Hiding

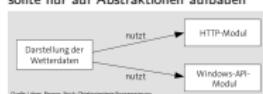
- Erweiterbarkeit in der OOP: Vererbung
- Geschlossenheit in der OOP: Die Implementation der Basisklasse muss dafür nicht angefasst werden
- Nutzung von Interfaces, ohne Annahmen über die Implementation zu kennen/zu verwenden. Ermöglicht die Wartbarkeit der Implementation



DEPENDENCY INVERSION PRINCIPLE

- Both high-level and low-level objects must depend on the same abstraction“
- Module sollen nur über (gemeinsame) Abstraktionen von anderen nutzender/genutzten Modulen abhängen

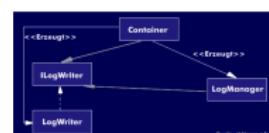
Eine betriebssystemunabhängige App sollte nur auf Abstraktionen aufbauen



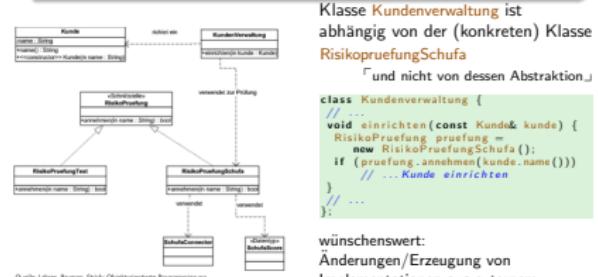
INVERSION OF CONTROL

Erzeugung/Aufruf eines Moduls für eine nutzende Klasse von einem externen Kontext aus

- Ein Modul wird extern registriert und zu einem späteren Zeitpunkt aufgerufen
- Anstelle der direkten Abhängigkeit des nutzenden Moduls von dem genutzten wird das genutzte Modul von einem externen Kontext erzeugt, der den Kontrollfluss dafür steuert
- Beobachter/Listener (vgl. Folie 45) sind einfache Beispiele
- Die vorher starke Kopplung zwischen nutzendem und genutztem Modul wird zu einer **losen Kopplung**



DEPENDENCY INJECTION IST EINE SPEZIELLE UMSETZUNG VON IoC
Erzeugung/Aufruf eines Moduls für eine nutzende Klasse von einem externen Kontext aus



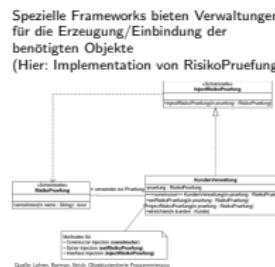
Klasse **Kundenverwaltung** ist abhängig von der (konkreten) Klasse **RisikoPruefungSchuфа**
„und nicht von dessen Abstraktion.“

```
class Kundenverwaltung {
    ...
    void einrichten(const Kunde& kunde) {
        RisikoPruefung pruefung = new RisikoPruefungSchuфа();
        if (pruefung.annehmen(kunde.name()))
            ...
        ...
    }
}
```

wünschenswert:
Änderungen/Erzeugung von Implementierungen aus externem Kontext heraus ermöglichen



DEPENDENCY INJECTION IST EINE SPEZIELLE UMSETZUNG VON IoC
Erzeugung/Aufruf eines Moduls für eine nutzende Klasse von einem externen Kontext aus



```
class KundenVerwaltung
/* gef. : public InjectRisikoPruefung */
RisikoPruefung pruefung;
...
KundenVerwaltung(
    const RisikoPruefung& pruefung) {
    this.pruefung = pruefung;
}
void setRisikoPruefung(
    const RisikoPruefung& pruefung) {
    this.pruefung = pruefung;
}
/* ergf */
void injectRisikoPruefung(
    const RisikoPruefung& pruefung) {
    this.pruefung = pruefung;
}
*/
void einrichten(const Kunde& kunde) {
    if (pruefung.annehmen(kunde.name()))
        ...
    ...
}
```

1 Prinzipien und Paradigmen des Softwareentwurfs

- Übergeordnete Architekturprinzipien
- Allgemeine Programmierparadigmen

2 Objektorientierte Programmierung

- Paradigmen der objektorientierten Programmierung
- Entwurfsmuster

3 Funktionale Programmierung

- Paradigmen der funktionalen Programmierung
- Container
- Iteratoren und Algorithmen
- Vertiefung: Funktoren
- Vertiefung: Rekursion

4 Generische / Generative Programmierung

- generische Programmierung
- generative Programmierung

Programmierparadigmen geben vor, wie Programmelemente (Funktionen, Daten), sowie der Kontrollfluss innerhalb einer Programmiersprache repräsentiert werden



- Daten repräsentieren Zustände
- Anweisungen manipulieren Daten und geben den Kontrollfluss vor

IMPERATIVE PROGRAMMIERUNG

Die Programmausführung definiert eine Input-Output-Relation zwischen den Anfangs-/Endzuständen.

Die Anweisungen bestimmen, „**WIE**“ die Transformation des Anfangs- in den Endzustand durchgeführt wird

- **prozedurale Programmierung** (um 1960):
Funktionale Dekomposition zur Vermeidung von Codeverdopplung und Erhöhung der Wartbarkeit
- **strukturierte Programmierung** (um 1970):
Kontrollfluss erlaubt nur drei Kontrollstrukturen: **Sequenz, Verzweigung, Wiederholung**
- **modulare Programmierung**
Dekomposition in (meist umfangreiche) eigenständige logische Einheiten („Module“)

- Objektvariablen repräsentieren (Objekt-)Zustände
- Methoden manipulieren Objektvariablen



OOP

Die Interaktion/Benachrichtigung von Objekten mittels Methodenaufrufe definiert den Kontrollfluss

OOP erweitert das imperative Paradigma um

- abstrakte Datentypen (Klassen) / Datenkapselung
- Vererbung
- Polymorphie



- zustandslos/nebenwirkungsfrei
 - Kontrollfluss wird durch das System vorgegeben



DEKLARATIVE PROGRAMMIERUNG

Das Problem wird durch abstrakte Zusammenhänge beschrieben
Der Programmierer beschreibt „WAS“ er erreichen will

- ## • funktionale Programmierung

Zusammenhänge werden durch Funktionen beschrieben. Verkettung von Funktionen, sowie **Rekursion** sind häufig Konstruktionsmittel.

- ## • logische Programmierung:

Zusammenhänge werden durch Fakten und Regeln beschrieben
Programmablauf mittels Anfragen



Example of GCD program

```
int gcd(int a, int b) {  
    while (a!=b) {  
        if (a>b) a = a-b;  
        else b = b-a;  
    }  
    return a;  
}
```

```
(define gcd
  (lambda (a b)
    (cond ((= a b) a)
          ((> a b) (gcd (- a b) b))
          (else (gcd (- b a) a)))))
```

```

gcd(A,B,G) :- A = B, G=A.
gcd(A,B,G) :- A > B, C is A-B, gcd(C,B,G).
gcd(A,B,G) :- B > A, C is B-A,
gcd(C,A,G).

```

%Prolo

Funktion muss für jeden

```
neu geschrieben werden
void tasche(int aa, int bb) {
    int temp;
    a=bb;
    b=temp;
}
void tasche(char aa, char bb) {
    char temp;
    a=bb;
    b=temp;
}
void tasche(double aa, double bb) {
    double tempaa;
    a=bb;
    b=tempaa;
}
```

Funktions-Template muss nur
einmal geschrieben werden:

sämtl. Klassen mit Zuweisungsoperator können als Typ zugewiesen werden.

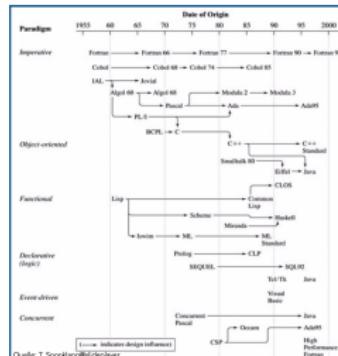
GENERIS

identischer Algorithmus für unterschiedliche Datentypen
Der Datentyp parametrisiert den Algorithmus

- **Program to Interfaces:**
Nutzung von Interfaces ist ebenfalls generischer Stil
 - **generative Entwicklung:**
Compiler erzeugt zur Compilezeit benötigten Code
möglichtriebene Entwicklung: z.B. automatische Codegenerierung aus UML-Diagramm
 - **Aspektorientierte Programmierung:**
Quelltextgenerierung für übergreifende Aspekte einer Klasse



Programmierparadigmen	
imperativ	Programm besteht aus Daten und Algorithmen gut zur (funktionalen) Dekomposition
objektorientiert	Programm besteht aus der Interaktion von Objekten gut zur Modellierung von Systemen
funktional	Programm besteht aus Verkettung von Funktionen gut für formale Beweise
logisch	Programm besteht aus Fakten und Regeln gut zum Suchen/Abfragen



Sprachen wie C++/Java sind **Multiparadigmensprachen**, d.h. sie unterstützen mehrere Paradigmen



- ➊ Prinzipien und Paradigmen des Softwareentwurfs
 - Übergeordnete Architekturprinzipien
 - Allgemeine Programmierparadigmen

- ➋ Objektorientierte Programmierung
 - Paradigmen der objektorientierten Programmierung
 - Entwurfsmuster

- ➌ Funktionale Programmierung
 - Paradigmen der funktionalen Programmierung
 - Container
 - Iteratoren und Algorithmen
 - Vertiefung: Funktoren
 - Vertiefung: Rekursion

- ➍ Generische / Generative Programmierung
 - generische Programmierung
 - generative Programmierung



- ➊ Prinzipien und Paradigmen des Softwareentwurfs
 - Übergeordnete Architekturprinzipien
 - Allgemeine Programmierparadigmen

- ➋ Objektorientierte Programmierung
 - Paradigmen der objektorientierten Programmierung
 - Entwurfsmuster

- ➌ Funktionale Programmierung
 - Paradigmen der funktionalen Programmierung
 - Container
 - Iteratoren und Algorithmen
 - Vertiefung: Funktoren
 - Vertiefung: Rekursion

- ➍ Generische / Generative Programmierung
 - generische Programmierung
 - generative Programmierung



Die drei Säulen der objektorientierten Programmierung

- ➊ Modularisierung/Kapselung
- ➋ Vererbung
- ➌ Polymorphie

OOP - Paradigmen

- ➊ Separation of Concerns (Modularisierung/Kapselung in Klassen/Pakete)
- ➋ Open-Closed Principle (durch Vererbung und Polymorphie)
- ➌ Program to Interfaces (Nutzung von Interfaces/Schnittstellen)



MODULARISIERUNG / KAPSELUNG

- ➊ Jede Klasse verbindet ihre Daten und die zugehörigen darauf operierenden Methoden zu einem eigenständigen Software-Modul
- ➋ Jede Klasse besitzt einen eigenen Namensraum, um Namenskonflikte mit anderen Modulen auszuschließen
- ➌ Nur die (Objekte der) Klasse haben direkten Zugriff auf die Daten der Klasse
- ➍ (Objekte) Andere(r) Klassen können nur auf freigegebene (öffentliche, ggf. geschützte, paketsichtbare) Daten/Methoden zugreifen.

VERERBUNG

Vererbung ermöglicht die Erweiterung bestehender Software ohne die Funktionalität der Elternklasse zu beeinträchtigen.

Damit unterstützt die objektorientierte Programmierung das Prinzip:
„offen für Erweiterungen, aber geschlossen für Veränderungen“

- Unterklassen erben die Spezifikation ihrer Elternklassen (und erweitern diese)
- Unterklassen erben die Implementierung der Methoden der Elternklassen
- Jedes Objekt einer Unterklassen kann anstelle eines Objektes der Elternklasse eingesetzt werden
„Prinzip der Ersetzbarkeit.“

POLYMORPHIE

Polymorphie ist die dynamische Bestimmung einer auszuführenden Implementation einer Methode

- Compilezeit-Polymorphie entsteht durch Überladung von Funktionen/Methoden
- Laufzeit-Polymorphie entsteht durch (virtuelles) Überschreiben einer Methode in einer abgeleiteten Klasse. Die zugehörige Implementation wird „spät gebunden“



- ① Prinzipien und Paradigmen des Softwareentwurfs
 - Übergeordnete Architekturprinzipien
 - Allgemeine Programmierparadigmen

- ② Objektorientierte Programmierung
 - Paradigmen der objektorientierten Programmierung
 - **Entwurfsmuster**

- ③ Funktionale Programmierung
 - Paradigmen der funktionalen Programmierung
 - Container
 - Iteratoren und Algorithmen
 - Vertiefung: Funktoren
 - Vertiefung: Rekursion

- ④ Generische / Generative Programmierung
 - generische Programmierung
 - generative Programmierung



Das Rad muss nicht immer neu erfunden werden...

Entwurfsmuster sind Designvorschläge für objektorientierte Software, in denen das Zusammenwirken von Klassen, Objekten und Methoden für einige spezifische Probleme beschrieben wird

Sehr gut zugängliche Beschreibung einiger Entwurfsmuster
<http://www.philippauer.de/study/se/design-pattern>



Verhaltensklassen: Entensimulation

Eigenschaften, die in einer Vererbungshierarchie unterschiedliche Ausprägungen haben, aber über eine gemeinsame Schnittstelle angesprochen werden sollen, lassen sich nicht einfach warten.



Probleme:

1. Kommen neue Enten hinzu, muss jedes Mal die Methode "flyen" neu implementiert werden (wartungsintensiv und ggf. unpassend)
2. Methode "flyen" in eigenständige Klassen kapseln, die jeweils von den Entenobjekten benutzt wird (oder auch nicht)
 -> Methode "flyen" nicht mehr in Schnittstelle Ente

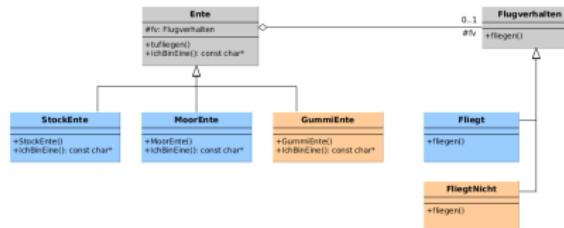
Elegante Lösung durch Delegation an eine Verhaltensklasse "Flugverhalten", die die Fähigkeiten aller Ententypen enthält

1.2b Entwurfsmuster

Verhaltensklassen: Flugverhalten als Klassenhierarchie

Idee: Interface **Flugverhalten** (insbesondere Methode `fliegen()`) wird auf verschiedene Weisen implementiert. Das Interface **Flugverhalten** dient

- der Sicherstellung der Implementation der Methode `fliegen()`
- der einheitlichen Zuordnung eines Flugverhaltens zu einer Ente mittels eines Basisklassenzeigers

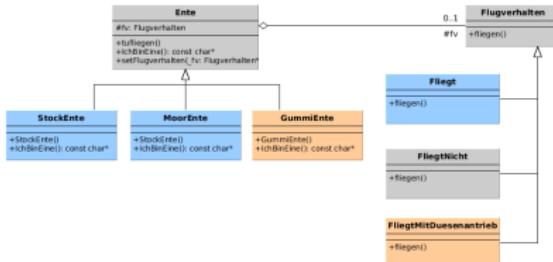


1.2b Entwurfsmuster

Verhaltensklassen: Erweiterung des Flugverhaltens LIVE_ENTENSIMULATION

Das bereits erarbeitete Konzept kann nun problemlos um weitere Flugverhalten erweitert werden (**FliegtMitDuesenAntrieb**).

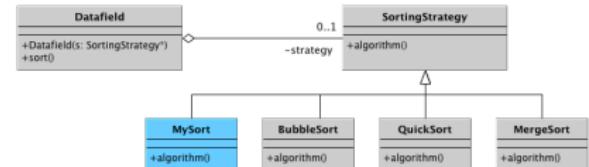
Das Flugverhalten einer Ente kann mittels der Funktion `setFlyBehavior` (**Flugverhalten***) gesetzt werden.



1.2b Entwurfsmuster

Das Strategie-Muster: Sortieralgorithmen

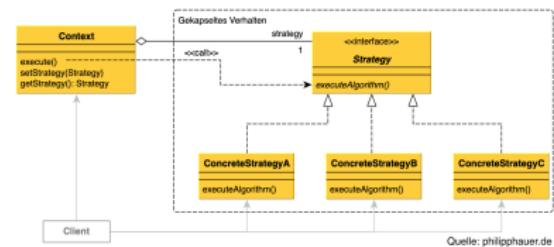
Das Flugverhalten für Enten ist ein Beispiel für ein **Entwurfsmuster**, hier das **"Strategie-Muster"**. Diese werden in der Vorlesung "Objektorientierte Analyse und Design" vertieft.



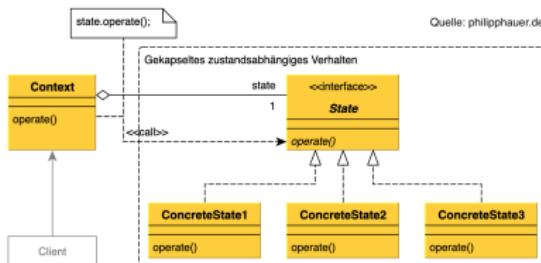
Ein typischer Anwendungsfall für ein Strategie-Muster ist die Sortierung von Datenfeldern mit Hilfe von verschiedenen Algorithmen. Auch die Implementation eines eigenen Sortierverfahrens ist hier durch Ableitung vom Interface **SortingStrategy** leicht umzusetzen.

1.2b Entwurfsmuster

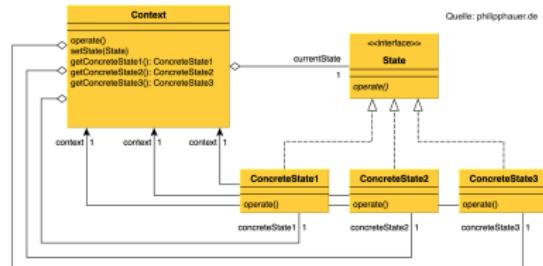
Das „Strategy“-Muster formal



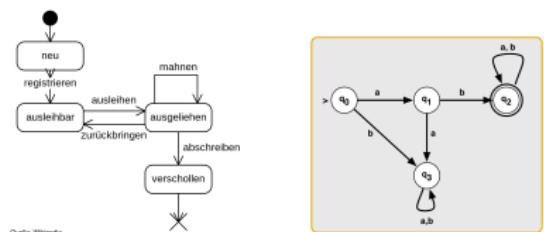
Quelle: philippauer.de



Formal ist dies dasselbe Klassendiagramm...



Für die Implementation von Zustandsübergängen halten die Zustände selbst eine Instanz des Kontextes. Dieser hält je eine Instanz aller möglichen Zustände und getter/setter dafür

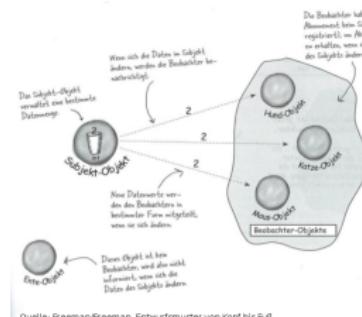


...aber jetzt hat man Zustände statt Eigenschaften und Operationen (zur Zustandsänderung) anstelle von Strategien

1.2b Entwurfsmuster

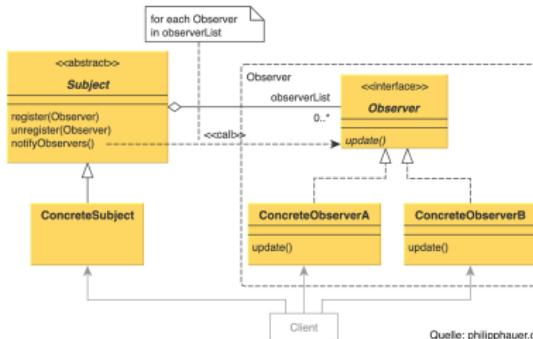
Das „Observer“-Muster

Das Beobachter-Muster soll für bei der Quelle „registrierte“ Beobachter dafür sorgen, dass jeder Beobachter informiert wird, wenn das Quellobjekt seinen Zustand ändert



1.2b Entwurfsmuster

Das „Observer“-Muster

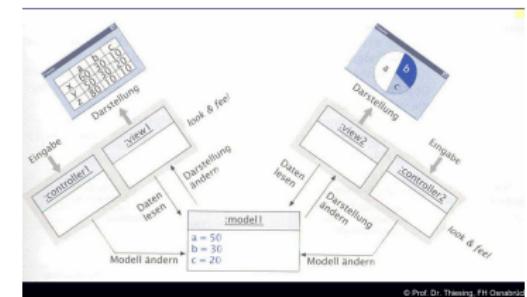


1.2b Entwurfsmuster

Das „Observer“-Muster

Bei graphischen Benutzeroberflächen ist die Darstellung (die **View**) von Daten (Zuständen/Spielständen...) ein Beobachter des **Modells** (Programmlogik).

Über einen **Controller**, der ggf. selbst die View „beobachtet“ (Mausclick/Tastatureingabe...) kann auf das Modell zugegriffen werden
Die Trennung in Model-View-Controller



1.2b Entwurfsmuster

Das „Observer“-Muster

LIVE-AMPELSIMULATION

Wie werden die Informationen zwischen Subjekt und Beobachter ausgetauscht?

- Push-Benachrichtigung — lose Kopplung

Geänderte Daten im Subjekt werden der **update()**-Methode übergeben

- Der Beobachter benötigt keine Informationen über das Subjekt
- Schlecht wartbar: Möglicherweise benötigt nicht jeder Beobachter dieselben Parameter
- Besser wartbare Variante: **Event-Delegation-Model**
Das Subjekt erzeugt ein Datenobjekt mit allen benötigten Informationen und übergibt dieses an die Beobachter



- Pull-Benachrichtigung — starke Kopplung

Die Beobachter halten eine Instanz des Subjekts und holen sich nach einem Subjekt-Update die benötigten Subjekt-Daten selbst

- Geeignet, wenn das Subjekt nichts über seine Beobachter weiß
- Beobachter müssen die Struktur des Subjekts kennen



- ⑤ Prinzipien und Paradigmen des Softwareentwurfs
 - Übergeordnete Architekturprinzipien
 - Allgemeine Programmierparadigmen
- ⑥ Objektorientierte Programmierung
 - Paradigmen der objektorientierten Programmierung
 - Entwurfsmuster
- ⑦ Funktionale Programmierung
 - Paradigmen der funktionalen Programmierung
 - Container
 - Iteratoren und Algorithmen
 - Vertiefung: Funktoren
 - Vertiefung: Rekursion
- ⑧ Generische / Generative Programmierung
 - generische Programmierung
 - generative Programmierung



- ⑤ Prinzipien und Paradigmen des Softwareentwurfs
 - Übergeordnete Architekturprinzipien
 - Allgemeine Programmierparadigmen
- ⑥ Objektorientierte Programmierung
 - Paradigmen der objektorientierten Programmierung
 - Entwurfsmuster
- ⑦ Funktionale Programmierung
 - Paradigmen der funktionalen Programmierung
 - Container
 - Iteratoren und Algorithmen
 - Vertiefung: Funktoren
 - Vertiefung: Rekursion
- ⑧ Generische / Generative Programmierung
 - generische Programmierung
 - generative Programmierung



Erinnerung: imperative Programmierung

```
void tausche(int& a, int& b) {
    int hilf=a;
    a=b;
    b=hilf;
}
```

PARADIGMA: „WIE“ (KOMME ICH ZUM ZIEL)

Programme bestehen aus Daten+Kontrollstrukturen

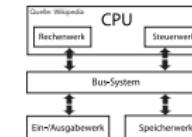
- Die Werte aller Daten definieren den gegenwärtigen Zustand
- Man muss sich den Zustand in jedem Programmschritt kennen und dem Kontrollfluss folgen, um das Ergebnis eines Algorithmus zu erhalten
- Man kann einzelne Programmzeilen/Abschnitte nur verstehen, wenn man den Zustand bzw. die vorhergehenden Zeilen kennt
- Dies gilt auch für Funktionen/Methoden in C++/Java, wenn sie Zugriff auf globale Daten haben (z.B. globale Variablen)



Die Elemente der imperativen Programmierung orientieren sich an der Von-Neumann-Architektur

```
void tausche(int& a, int& b) {
    int hilf=a;
    a=b;
    b=hilf;
}
```

- Daten/Anweisungen — Speicherwerk
- Kontrollfluss — Steuerwerk
- Durchführung von Operationen — Rechenwerk





funktionale Programmierung

PARADIGMA: „WAS“ (SOLL GETAN WERDEN)

Programme bestehen aus der Verkettung von Funktionen

- Jede Funktionsdefinition ist unabhängig von den anderen Programmteilen
- Jeder Funktionsaufruf führt bei gleichen Aufrufparametern stets zu den gleichen Ergebnissen
- Die Auswertung von Funktionen geschieht nicht (imperativ) als sequentielle Abarbeitung von Befehlen, sondern durch (oft rekursive) Verschachtelung von Funktionsaufrufen
- Variablen können nur innerhalb einer Funktion definiert werden und verlieren ihre Bedeutung nach Verlassen der Funktion
- Die Unabhängigkeit vom Zustand nennt man **zustandslos**, oder auch **nebenwirkungsfrei**
→ Analyse von Programmen formal erheblich einfacher als im imperativen Paradigma



Das λ -Kalkül — ein System grammatischer Ersetzungsregeln

DAS λ -KALKÜL ALS OPERATIVER KERN FUNKTIONALER SPRACHEN

Von Alonzo Church und Stephen C. Kleene in den Dreißigerjahren entwickelt

- Syntax unterscheidet nicht zwischen Daten und Funktionen
- λ steht für ein generisches Symbol einer Funktion
- erlaubte Ausdrücke sind „links-assoziative Aneinanderreihungen von...“
 - Variablennamen x, y, z, \dots
 - Applikationen (fx), wenn f und x λ -Ausdrücke sind
 - Abstraktionen $\lambda x.M$, wenn x eine Variable und M ein λ -Ausdruck ist
 „bindet eine Variable (Funktionsdefinition)“
- (Ersetzungs-)Regeln
 - (α -)Substitution (Umbenennung), sofern dabei keine freien Var. gebunden werden
 - (β -)Reduktion: Ersetzung des gebundenen Arguments durch das Applikat (entspricht dem Einsetzen/Auswerten einer Funktion)

$$(\lambda x.M)N \longrightarrow M[x \text{ ersetzt durch } N]$$



Das λ -Kalkül — ein System grammatischer Ersetzungsregeln

Die λ -Ausdrücke und Ersetzungsregeln ermöglichen die Definition von Zahlen und Rechenregeln, die mittels grammatischer(!) Operationen im λ -Kalkül implementiert werden können:

Zahlen:

$$\begin{aligned} 0 &\equiv \lambda s z . z \\ 1 &\equiv \lambda s z . s z \\ 2 &\equiv \lambda s z . s (s z) \\ 3 &\equiv \lambda s z . s (s (s z)) \\ 4 &\equiv \lambda s z . s (s (s (s z))) \\ &\dots \end{aligned}$$

„Anzahl der s hinter dem „.“ entspricht dem Zahlenwert._j

Zahloperationen (Anordnung natürlicher Zahlen):

Nachfolgerfunktion:
 $\text{succ} \equiv \lambda n s z . (n\ s\ z)$

Applizieren auf Zahlen:

$$\begin{aligned} \text{succ } 0 &\equiv (\lambda n s z . (n\ s\ z)) (0\ s\ z) \\ &\xrightarrow{M \rightarrow M[N \text{ für } n]} (\lambda s z . s z) (0\ s\ z) \\ &\xrightarrow{S \rightarrow s, z \rightarrow z} 0\ s\ z \\ &\longrightarrow 1 \end{aligned}$$

Analog: $\text{succ } 1 \longrightarrow 2, \text{succ } 2 \longrightarrow 3, \dots$

BEISPIEL | Auswertung von $3 + 2$

$$\begin{aligned} ((3 \text{ succ}) 2) &= (\lambda s z . s (s (s z))) (\lambda n s z . (n\ s\ z)) (0\ s\ z) \\ &\longrightarrow \text{succ} (\text{succ} (\text{succ } 2)) \longrightarrow \text{succ} (\text{succ } 3) \longrightarrow \text{succ } 4 \longrightarrow 5 \end{aligned}$$



- mathematische Berechnungen entstehen im λ -Kalkül durch Auswertung einer formalen Syntax/Grammatik
- funktionaler Aspekt: Durch formale Manipulationsregeln können Funktionen verkettet und ausgewertet werden
- funktionale Programmiersprachen definieren basierend auf dem λ -Kalkül eigene Syntax zur Manipulation von Ausdrücken

„gutes Einführungsvideo: https://youtu.be/eis11j_iGMs“



I.3a Funktionale Programmierung in Haskell

Die funktionale Programmiersprache Haskell ist 1990 von dem Mathematiker/Logiker Haskell Brooks Curry entwickelt worden.

- Haskell ist eine typisierte Programmiersprache
- Neben einfachen Datentypen sind die fundamentalen Bausteine **Listen** und **Funktionen**
- Kontrollstrukturen: Verzweigung, keine Schleife, dafür Rekursion(+Listen)
- **Lazy evaluation (Bedarfsauswertung)** erlaubt Umgang mit unendlichen Listen oder nicht auswertbaren Termen:
 $\text{length}([2+1, 3*2, 1/0, 5-4])=4$ „Zur Bestimmung der Länge der Liste ist eine Auswertung der einzelnen Listenelemente nicht notwendig.“

„Neben der angegebenen Literatur gibt es auch einen sehr informativen Foliensatz der Vorlesung Funktionale Programmierung an der Uni Dortmund von Prof. E.E. Doberkat, den man im WWW findet.“



I.3a Funktionale Programmierung in Haskell — Die Fakultät

```

1   -- Fakultät
2   -- mathematische Definition für nichtnegative ganze Zahlen lautet
3   0! = 1
4   n! = n*(n-1)!
5
6   -- In Haskell sieht das so aus:
7   fak :: Int -> Int      -- Definition einer Funktion
8   |      -- die ganze Zahlen in ganze Zahlen abbildet
9   |      -- Implementation: Parameter n wird ohne
10  |      -- runde Klammer notiert
11  | fak n
12  |      -- Implementation: Parameter n wird ohne
13  |      -- runde Klammer notiert
14  |      | (n==0) = 1      -- "|" kennzeichnet eine Alternativen
15  |      | (n>0) = n*fak (n-1) -- für die Berechnung des Funktionswertes,
16  |      |                  -- die durch "=" eingeleitet wird
17

```

“Playground”

```

fak 6
map fak [0 .. 8]
-- map wendet die Fkt
-- fak sukzessive auf
-- die ganze Liste an

```

fak 6	720	Int
map fak [0 .. 8]	[1,2,6,24,120,720,5040,40320]	[Int]



I.3a Funktionale Programmierung in Haskell — Die Fibonacci-Zahlen

```

31   -- Fibonacci-Zahlen "iterativ" bzw. linear rekursiv
32   -- Zunächst eine Hilfsfunktion fibl mit 2 Parametern:
33   -- Param1: nichtnegative ganze Zahl n
34   -- Param2: Ein Paar (a,b) nichtnegative ganzer Zahlen
35   -- return: Paar, bestehend aus aufeinanderfolgenden Fib-Zahlen
36
37   fibl 0 (f1, f2) = (f1, f2)  -- "Basisfall": Funktion macht nichts, wenn n==0
38
39   fibl n (f1, f2) =          -- Alternativen können auch durch explizites
40   |      -- Ausschreiben der Funktionsdefinition für
41   |      -- den betreffenden Fall notiert werden
42
43   | fibl (n-1) (f2, f1 + f2)  -- entspricht der iterativen C++ Sequenz:
44   |      -- int g=f1+f2; f1=f2; f2=g;
45
46   -- Funktion zum Aufrufen für den Anwender
47   -- Aufruf von fibl mit Startwerten f1=0, f2=1
48   fibl n = fst(fibl n (0, 1))  -- "fst" extrahiert das erste Element
49   -- aus einer Liste
50

```

“Playground”

fibl 5 (0, 1)	(5,8)	Num b => (b, b)
fibl 5	5	Num b => b
fst ("Hello", 5)	"Hello"	[Char]
snd ("Hello", 5)	5	Num b => b



I.3a Funktionale Programmierung in Haskell — Primzahlsieb

```

67   -- Primzahlsieb
68   -- Parameter: Liste [2,3,4,...,n]
69   -- return:  Liste von Primzahlen bis n
70   sieb :: [Int] -> [Int]  -- Umklammerung [] kennzeichnet eine Liste
71   sieb (x:xs)            -- (x:xs) kennzeichnet eine Liste mit Listenkopf x
72   | (xs==[]) = []        -- [] meint die leere Liste
73   | otherwise = x: sieb [n | n <- xs, mod n x > 0]
74   |      -- "[" innerhalb einer Liste meint: "für die gilt"
75   |      -- "<-" meint "entnimmt aus"
76   |      -- ":" meint: "diejenigen Elemente, für die gilt"
77   |      -- "mod n x>0" meint (in C++/Java): "n%>x>0"
78

```

“Playground”

sieb [2 .. 50]	[2,3,5,7,11,13,17,19,23,29,31,37]	[Int]
	[41,43]	



```
62
63 -- Quicksort
64 quickSort [] = []
65 quickSort (x:xs) =
66   quickSort [y | y <= xs, y<=x] ++
67   [x] ++
68   quickSort [y | y <= xs, y>x]
69
```

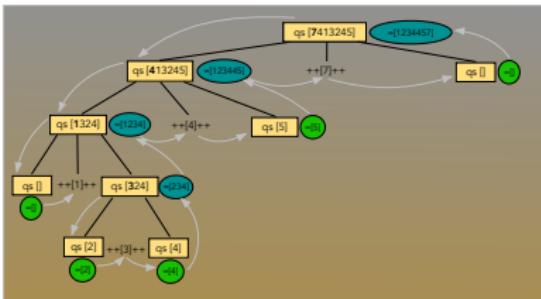
"Playground"

dieListe = [7, 4, 1, 3, 2, 4, 5]
quickSort dieListe

dieListe :: Num a => [a]
(Ord a, Num a) => [a]



```
62
63 -- Quicksort
64 quickSort [] = []
65 quickSort (x:xs) =
66   quickSort [y | y <= xs, y<=x] ++
67   [x] ++
68   quickSort [y | y <= xs, y>x]
69
```



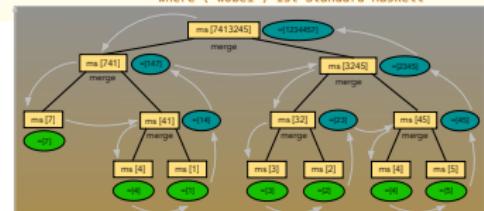
```
74
75 -- Mergesort
76 merge [] ys = ys
77 merge xs [] = xs
78 merge (x:xs) (y:ys)
79   | x <= y = x:merge xs (y:ys)
80   | otherwise = y:merge (x:xs) ys
81 medianpos liste = div (length liste) 2 -- "div" meint Ganzahldivision
82 aufteilen liste = splitAt (medianpos liste) liste
83
84 mergeSort [] = []
85 mergeSort [x] = [x]
86 mergeSort liste = merge (mergeSort links) (mergeSort rechts)
87   where (links, rechts) = aufteilen liste
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
```

"Playground"

merge [1,3,7] [2,4,6]	[1,2,3,4,6,7]	(Ord a, Num a) => [a]
medianpos dieListe	3	Int
aufteilen dieListe	([7,4],[3,2,4,5])	Num a => ([a], [a])
mergeSort dieListe	[1,2,3,4,4,5,7]	(Ord a, Num a) => [a]



```
74
75 -- Mergesort
76 merge [] ys = ys
77 merge xs [] = xs
78 merge (x:xs) (y:ys)
79   | x <= y = x:merge xs (y:ys)
80   | otherwise = y:merge (x:xs) ys
81 medianpos liste = div (length liste) 2 -- "div" meint Ganzahldivision
82 aufteilen liste = splitAt (medianpos liste) liste
83
84 mergeSort [] = []
85 mergeSort [x] = [x]
86 mergeSort liste = merge (mergeSort links) (mergeSort rechts)
87   where (links, rechts) = aufteilen liste
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
```



65 I.3a Funktionale Programmierung
in Haskell — Quick- vs. Mergesort



O. Henkel

Quicksort	Mergesort
'60 Hoare	'46 von Neumann
Partitionieren, dann Konkatenieren	Aufteilen, dann Zusammenführen
Zahlenwert Position	Zahlenwert Position
Zeit günstig: $n \log n$	$n \log n$
Zeit ungünstig: n^2	$n \log n$
Zeit Durchschnitt: $n \log n$	$n \log n$
Speicher: n	$n + \frac{n}{2}$
stabil: nein	ja

66 I.3a Funktionale Programmierung
in C++

Das (zustandslose) funktionale Programmierparadigma wird in C++
unterstützt durch

- Iteratorensequenzen auf Containern als „Liste“ und zugehörige „Algorithmen“ auf solchen Sequenzen
- λ -Funktionen, Funktionsobjekte oder Funktionszeiger
 - „im Folgenden übergreifend als „Funktoren“ bezeichnet.“
- (rekursive) Funktionen



Prinzipien und Paradigmen des Softwareentwurfs

- Übergeordnete Architekturprinzipien
- Allgemeine Programmierparadigmen

Objektorientierte Programmierung

- Paradigmen der objektorientierten Programmierung
- Entwurfsmuster

Funktionale Programmierung

- Paradigmen der funktionalen Programmierung
- Container**
- Iteratoren und Algorithmen
- Vertiefung: Funktoren
- Vertiefung: Rekursion

Generische / Generative Programmierung

- generische Programmierung
- generative Programmierung



Bibliothek	vector	deque	list
Beschreibung	dynamisches Feld	doppelseitige Schlange	Liste
logisches Speicherabbild	[front] ... [back]	[front] ... [back]	[front] ... [back]
Konstruktion	vector<T> c; vector<T> c(n); vector<T> c(n,v); vector<T> c {...}; vector<T> c(d);	deque<T> ~	list<T> ~
Anfügen/Entfernen	c.push_back(); c.pop_back();	+c.push_front(v); +c.pop_front();	~
Elementzugriff	v=c.front(); v=c.back(); v=c[i]; v=c.at(i);	~	-c[1]; -c.at(i);
Globalzugriff	n=c.size(); c.resize(n); c.clear(); b=c.empty(); =	~	+c.unique([bp]); +c.remove(v); +c.remove_if(p); +c.sort([cmp]); +c.reverse(); +c.merge(d,[cmp]);

Legende:

int n; T v; initializer.list<T> {...}; bool b; C<T> d; wobei C den jeweiligen Containertyp bezeichnet
Optionaler Parameter: Prädikate: p (unär), bp, cmp (binär) für Argumente vom Typ T

~: Dasselbe wie links; +: zusätzlich; -: ohne

↳: abgeleitet, Anforderung: Typ T muss „~, <, <=“ unterstützen; lexikographischer Vergleich gemäß logischem Speicherabbild



Adapter machen sich die Funktionalität einer gegebenen Klasse zunutze und adaptieren ihre Schnittstelle an spezielle Bedürfnisse

Bibliothek	stack	queue
Beschreibung	Stapel	Warteschlange
logisches Speicherabbild	[top] ... [bottom]	[front] ... [back]
Konstruktion	stack<T> c; stack<T> c(d);	queue<T> ~
Anfügen/Entfernen	c.push(v); c.pop();	~(FIFO***)
Elementzugriff	v=c.top(); (LIFO) v=c.back();	v=c.front(); (PRIO) v=c.top(); (PRIO**)
Globalzugriff	n=c.size(); b=c.empty(); =	~
	==, < (lex) ↳ !, <, <=, >, >=	==, <, <=, ↳ !, <, <=, >, >=

Legende:

int n; T v; bool b; C<T> d; wobei C den jeweiligen Containertyp bezeichnet

~: Dasselbe wie links; +: zusätzlich; -: ohne

↳: abgeleitet Operatoren

lexikographischer Vergleich gemäß logischem Speicherabbild; für PRIO keine Vergleichsoperatoren definiert

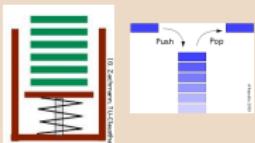
*: z.B. priority_queue <int> == priority_queue<int, vector<int>, less<int> >

**: Das top-Element im Container c wird festgelegt durch: v _c cmp(top,x)==false

***: LIFO: pop entfernt top-Element, push fügt top-Element an

FIFO: pop entfernt front-Element, push fügt back-Element an

HINTERGRUND | Stapel – Last In First Out (LIFO)



Ein Stapel-Container ist ähnlich einem Vektor, nur dass Daten nur nach dem LIFO-Prinzip abgelegt und entnommen werden können:

Umsetzung in C++ nach Einbindung der Bibliothek `stack`:

```
stack<int> stapel;
stapel.push(1); stapel.push(2); stapel.push(3); // 1 2 3
int obersterWert = stapel.top(); // 3
stapel.pop(); // 1 2
bool leer = stapel.empty(); // false
```

HINTERGRUND | Warteschlange – First In First Out (FIFO)



Typische Anwendung: Container/Pufferspeicher (z.B. Drucker-Warteschlange)

Umsetzung in C++ nach Einbindung der Bibliothek `queue`:

```
queue<int> schlange;
schlange.push(1); schlange.push(2); schlange.push(3); // 1 2 3
int vordersterWert = schlange.front(); // 1
schlange.pop(); // 2 3
bool leer = schlange.empty(); // false
```

HINTERGRUND | Prioritätswarteschlange



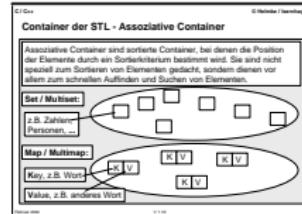
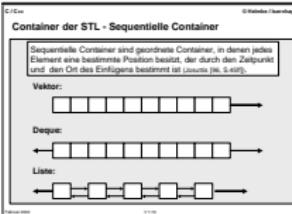
Entnahme der Daten nach Priorität
(Ordnungsrelation auf den Elementen)

Umsetzung in C++ nach Einbindung der Bibliothek `queue`:

```
priority_queue<int> pschlange;
pschlange.push(1); pschlange.push(3); pschlange.push(2); // 1 3 2
int wichtigsterWert = pschlange.top(); // 3
pschlange.pop(); // 1 2
bool leer = pschlange.empty(); // false
```

73 I.3b Funktionale Programmierung

in C++: Assoziative Container



- Bezug zum funktionalen Paradigma: Auch assoziative Container können mit Iteratoren sequentiell durchlaufen und als Iteratorsequenz `c.begin()...c.end()` bearbeitet werden
- Es gibt zwei Arten assoziativer Container:
 - sortiert, baumartig angelegt
 - unsortiert, mittels Hash-Schlüssel

74 I.3b Funktionale Programmierung

in C++: sortierte assoziative Container



Sortierte assoziative Container sind baumartig angelegte Speicher.
Die Position eines Wertes wird durch den zugehörigen Schlüssel bestimmt

Die vier (sortierten) assoziativen Container in C++ sind:

Containertyp C	Beschreibung	Hilfstypen
<code>map<K,V></code>	enthält Schlüssel-Wert-Paare als Elemente	<code>C::key_type=K, C::mapped_type=V, C::value_type=pair<const K, V></code>
<code>multimap<K,V></code>	gleiche Schlüsselwerte können mehrfach vorkommen	<code>~</code>
<code>set<K></code>	Wie Map, mit K=V	<code>C::key_type=K, C::value_type=K</code>
<code>multiset<K></code>	Wie Multimap, mit K=V	<code>~</code>

Der Hilfstyp `pair<K,V>` für Schlüssel-Wert-Paare (BIBLIOTHEK: UTILITY) koppelt zwei unterschiedliche Objekte aneinander, z.B.:

```
pair<Datum, double> messung(Datum(01, 01, 2000), 7.15);
Datum dat = messung.first; // Datum(01, 01, 2000)
cout << "Messwert:" << messung.second; // 0.75
```

75 I.3b Funktionale Programmierung

in C++: sortierte assoziative Container



LIVE_GEBURTSTAGSLISTE

Bibliothek	set/map	Multi-
Beschreibung	Menge/Zuordnung	
Konstruktion	<code>set<K,[cmp]>c;</code> <code>set<K>c(d);</code> <code>set<K,[cmp]>c(first,last);</code> <code>map<K,V,[cmp]>c;</code> <code>map<K,V>c(d);</code> <code>map<K,V,[cmp]>c(first,last);</code>	
Anfügen/Entfernen	keine seq. Operationen; Einfüge-Methoden/Iteratoren verwendbar	
Elementzugriff	Bidirektionalliteratoren für seq. Zugriff <code>* / map: v=<c[k]>; c[k]=v;</code> <code>n=c.size(); c.clear(); b=c.empty(); n=c.count(k); ***</code> <code>*, ==, <(lex)* !=, <, >, >=</code>	
Globalzugriff	<code>it=c.find(k);</code> <code>it=c.lower_bound(k);</code> <code>it=c.upper_bound(k);</code> <code>it=pair<C::key_type, C::value_type>::equal_range(k);</code>	
Iteratorzugriff	<code>it.ok=c.insert(e);</code> <code>c.insert(first,last);</code> <code>n=c.erase(k);***</code> <code>c.erase(it);c.erase(first,last);</code>	
Legende:	<code>int a, b, c; C c, d; K k;</code> Schlüssel-Typ muss <code>==</code> , <code><</code> unterstützen; cmp binäres Prädikat, <code>V v;</code> (Wert-Typ); <code>(multi)map::pair<const K, V> e;</code> (Element-Typ); <code>(multi)set::K=V</code> und Elemente sind als <code>K e;</code> deklariert <code>=</code> gemäß sortierter Schlüsselsequenz zwischen <code>c.begin(); c.end();</code> <code>*</code> Falls <code>k</code> nicht existiert wird Wert <code>V() erzeugt!</code> *** Anzahl Elemente mit Schlüssel <code>k</code>	
Iteratoren:	<code>C::iterator it, first, last; pair<C::iterator, bool> itok; pair<C::iterator, C::iterator> it_pair;</code> <code>Iteratoren auf (multi)map vereinbaren mittels <code>it->first, it->second</code> auf Schlüssel/Wert des zugehörigen Elementes</code>	



ausführlicher: [MAPCONT_ZUGRIFF](#)

```
// Standard-Abkürzungen
typedef map<string,int> Container;
typedef Container::iterator C_Iter;
typedef Container::value_type C_Elem;
typedef Container::key_type C_Key;
typedef Container::mapped_type C_Value;

Container months;
months.insert(C_Elem("Januar",31)); //Einfuegen
months["Februar"] = 28; // bei map auch so möglich!
months["April"] = 30;

// Zugriff ueber Iteratoren
C_Iter it; // *it hat den Typ pair<string, int>
cout << "alphabetische Liste:" << endl;
for (it=months.begin(); it!=months.end(); it++)
    cout << it->first << " ->->" << it->second << endl;
cout << endl; // first/second greifen auf key/value zu
```



ausführlicher: [MAPCONT_ZUGRIFF](#)

```
// Iterator-Arithmetik
C_Iter cur = months.find("Februar");
C_Iter prev = --cur; // -> April
C_Iter next = ++cur; // -> Januar!

// Einfuegen / Doppelte
pair< C_Iter, bool > it_OK;
it_OK=months.insert(C_Elem("Maerz",31));
// it.OK.first: eingefügtes Element, it.OK.second==true
it.OK=months.insert(C_Elem("Februar",29));
// it.OK.first==months.end(), it.OK.second==false
```



ausführlicher: [MULTIMAPCONT_ZUGRIFF](#)

```
// Standard-Abkürzungen
typedef multimap<string,int> Container;
typedef Container::iterator C_Iter;
typedef Container::value_type C_Elem;
typedef Container::key_type C_Key;
typedef Container::mapped_type C_Value;

// Einfügen nur noch in der allgemeinen Form möglich
months.insert(C_Elem("Januar",31));
// Rueckgabe: eingefügte It-Pos.
// months["Januar"]==31; // geht nicht
months.insert(C_Elem("Februar",28));
months.insert(C_Elem("Februar",29)); // jetzt möglich!
```

// Zugriff ueber Iteratoren

```
C_Iter it; // Lexikographische Ordnung:
for (it=months.begin(); it!=months.end(); it++)
    cout << it->first << " ->->" << it->second << endl;
```

// Finden

```
months.find("Januar")->second; // 31
it=months.find("Gibt_nicht"); // it==months.end()
```



ausführlicher: [MULTIMAPCONT_ZUGRIFF](#)

```
// Bereiche: equal_range liefert Bereichsiterator
pair< C_Iter, C_Iter > it_pair=months.equal_range("Februar");
// (it_pair.first)->first==Februar, (it_pair.first)->second==28
// (it_pair.second)->first==Januar, (it_pair.second)->second==31

// Bereiche ueber lower_bound/upper_bound
C_Iter it1,it2;
// erstes El. mit Schluessel <= bzw. > "Februar"
it1=months.lower_bound("Februar");
it2=months.upper_bound("Februar");

// binaeres Praedikat für C_Key-Werte...
class MyCompareStrings {
public:
    bool operator()(const C_Key& s, const C_Key& t) const
    { return (s.length())<(t.length()); }
}; // ~ Anforderung an Typ C_Key!
// ... definiert eine andere Sortierung
multimap<C_Key, C_Value, MyCompareStrings> someMonths;
someMonths.insert(months.begin(),months.end());
for (it=someMonths.begin(); it!=someMonths.end(); it++)
    cout << it->first << " ->->" << it->second << endl;
// Sequenz jetzt nach Wortlaenge der Schluessel sortiert
```

I.3b Funktionale Programmierung in C++: unsortierte assoziative Container

SORTIERTE ASSOZIATIVE CONTAINER

Aufgrund der geordneten Baumstruktur der Elementsschlüssel lassen sich Containerelemente (bzw. deren Schlüssel) in einem Container mit n Elementen in der Größenordnung von $\log_2 n$ einfügen oder finden

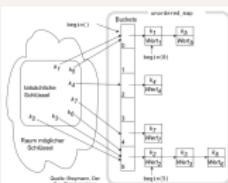


UNSORTIERTE ASSOZIATIVE CONTAINER

Die Elemente sind nicht angeordnet gespeichert und lassen sich durch eine **Hashfunktion** in konstanter Zeit einfügen oder finden

Hashfunktionen berechnen aus (jedem) Schlüssel direkt eine Zahl („Bucket“) und stehen für die Standarddatentypen zur Verfügung.

Landen mehrere Elemente in demselben Bucket, wird eine Liste gebildet, durch die von `begin(n)` bis `end(n)` für den Bucket Nr. n iteriert werden kann



I.3b Funktionale Programmierung in C++: unsortierte assoziative Container

- Hash-Container:

`unordered_map`, `unordered_set`, `unordered_multimap`, `unordered_multiset`
Bei Ausgabe von Iteratorsequenzen sind die Elemente nicht über die Schlüssel sortiert

- Methoden:

Dieselben, wie für die sortierten assoziativen Container, bis auf

- Iteratoren sind nicht mehr bidirektional, sondern unterstützen nur den Vorwärtsinkrement
- Hash-Container können nicht mit `<`, `<=`, `>`, `>=` verglichen werden
- kein `lower_bound`, `upper_bound` (aber noch `equal_range`)

- Verwendungsbeispiel:

 UNORDERED_MAPCONT_ZUGRIFF,
UNORDERED_MULTIMAPCONT_ZUGRIFF

In den Beispielen sind zusätzlich die Änderungen bezüglich der sortierten assoziativen Container markiert

I.3b Funktionale Programmierung in C++: unsortierte assoziative Container

- Methoden nur für Hash-Container

- Konstruktion

```
// Definition einer eigenen Hash-Funktion
struct meinHash {
    size_t operator()(string key) const {
        return key.length();
    } // Berechnung einer Zahl aus key
};
unordered_multimap<string, int, meinHash> someMonths;
// key      value   Hash
```

- Buckets

```
// Einige Methoden für Buckets
someMonths.bucket_count() // Anzahl Buckets
someMonths.bucket("Mai") // Nummer des Buckets zum Schlüssel "Mai"
someMonths.bucket.size(someMonths.bucket("Mai")) // Groesse des Buckets zum Schlüssel "Mai"
```



- ① Prinzipien und Paradigmen des Softwareentwurfs
 - Übergeordnete Architekturprinzipien
 - Allgemeine Programmierparadigmen
- ② Objektorientierte Programmierung
 - Paradigmen der objektorientierten Programmierung
 - Entwurfsmuster
- ③ Funktionale Programmierung
 - Paradigmen der funktionalen Programmierung
 - Container
 - Iteratoren und Algorithmen
 - Vertiefung: Funktoren
 - Vertiefung: Rekursion
- ④ Generische / Generative Programmierung
 - generische Programmierung
 - generative Programmierung



I.3c Funktionale Programmierung

in C++: Iteratoren (Erinnerung)

DIE LISTENABSTRAKTION DER C++ CONTAINER

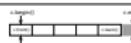
Jeder(!) Container (außer ~~MAP~~ ADAPTER) stellt in seinem Namensbereich eine Iteratorklasse bereit, um Containerelemente zu selektieren.

Damit sind Iteratorsequenzen Abstraktionen („Program to Interfaces, Dependency Inversion Principle“), die die Elemente jedes Containers als Liste bereitstellen, auf der andere Algorithmen operieren können.

Container	alle	vector, deque, string
Kategorie	bidirektional c.begin(); c.end(); it++; it--;	Random-Access
Positionierung	advance(it, n); n=distance(first, last);	+it+=n; it-=n; +it=it+n; it=it-n;
Elementzugriff	*it; it->"	+it[a];" +<, <=, >, >=
Vergleich	==, !=	

Legende:

```
int n; C c; C::ContainerTyp (kein Adapter)
C::iterator it, first, last;
Jeder Container stellt die Iteratorklasse C::iterator bereit, wobei C der zugehörige ContainerTyp ist
+: zusätzlich; <: ohne
*: it-> == *(it). selektiert Klassen- oder Strukturkomponente im Elementtyp, falls zutreffend
**: it[n] == *(it+n);
```





Einfüge-Iteratoren fügen an ihrer Position ein neues Element ein, ohne den Containerinhalt zu überschreiben
 Sie verwenden dabei die Container-Methoden
 insert, push_front und push_back (sofern vorhanden)

Funktion	Anwendung	Alternative
einfügen	insert_iterator<C> ins_it(c, pos); *ins_it=e;	ins_it= inserter(c, pos); *inserter(c, pos)=e;
vorne	front_insert_iterator<C> frontIns_it(c); *frontIns_it=e;	frontIns_it=front.inserter(c); *front.inserter(c)=e;
hinten	back_insert_iterator<C> backIns_it(c); *backIns_it=e;	backIns_it=back.inserter(c); *back.inserter(c)=e;
anfügen		

Legende:
 C (nur Adapter); C::iterator pos; C::value_type e;
 Jeder Container C stellt durch C::value_type den Typ seiner Elemente bereit
 Die aufgeführten Alternativen verwenden die dort genannten Funktionsobjekte

86 I.3c Funktionale Programmierung

in C++: Weitere Iteratoren — Einfüge-Iteratoren

EP ITERATOREN_ZUM_EINFÜGEN

```
// Standard-Abkürzungen
typedef list<int> Container;
typedef Container::iterator C_it;
typedef Container::value_type C_Elem;

// Info-Ausgabe: Implementation sei vorgegeben
// Die Nutzung von template wird im Abschnitt
// generische/generative Programmierung behandelt
template<typename Container> void info(const Container& c);

Container c; // Container und daran gebundene Iteratoren
front_insert_iterator<Container> frontIns_it(c);
back_insert_iterator<Container> backIns_it(c);

for (int i=0; i<5; i++) *frontIns_it=(-i*i); // vorne anfügen
for (int i=0; i<5; i++) *backIns_it=(i*i); // hinten anfügen
info(c); // Ausgabe: 10| -16 -9 -4 -1 0 0 1 4 9 16 |
```

87 I.3c Funktionale Programmierung

in C++: Weitere Iteratoren — Einfüge-Iteratoren

EP ITERATOREN_ZUM_EINFÜGEN

```
C_it pos=c.begin();
while (*pos!=0) pos++; // Position des Wertes 0 aufsuchen
// und einen Einfüge-Iterator darauf platzieren:
insert_iterator<Container> ins_it(c, pos); // so
//insert_iterator<Container> ins_it=inserter(c, pos); // oder so

*ins_it=100; // Dasselbe wie c.insert(pos, 100);
*ins_it=101; // Dasselbe wie c.insert(pos, 101);
*ins_it=102; // Dasselbe wie c.insert(pos, 102);
info(c); // 13| -16 -9 -4 -1 100 101 102 0 0 1 4 9 16 |

*inserter(c, ++pos)=104; // Dasselbe wie c.insert(++pos, 104);
*inserter(c, pos)=105; // Dasselbe wie c.insert(pos, 105);
info(c); // 15| -16 -9 -4 -1 100 101 102 0 104 105 0 1 4 9 16 |
```

Strom-Iteratoren erlauben ein (einfügendes bzw. auslesendes)
Durchlaufen von Strömen (`cin/cout/fstream`)

Anwendung	Beschreibung
<code>istream<T> it_in(istream); T t = *it_in++;</code>	Eingabe-Iterator für T-Objekte, der an <code>istrom</code> gebunden wird
<code>istream<T> it_eof;</code>	generischer End-Iterator
<code>ostream<T> it_out(ostream, text); *it_out++ = *it_cin++;</code>	Ausgabe-Iterator für T-Objekte, der an <code>ostrom</code> gebunden wird und nach jeder Ausgabe noch <code>text</code> anfügt

Legende:
C=ContainerTyp, T=C::value_type

```
// Standard-Akzuerungen
typedef list<int> Container; //oder list<double>, vector<float>...
typedef Container::iterator C_Iter;
typedef Container::value_type C_Elem;

// Einlesen
// Eingabestrom-Iterator an einen Eingabestrom binden
istream<C_Elem> it_cin(cin);

// generischen End-Iterator definieren
istream<C_Elem> eof;

// Container c initialisieren mit dem
// noch einzugebenen(!) Iteratorbereich
// Bei einem ifstream-Objekt hiesse das:
// Dateiinhalt bis zum Dateiende oder
// Einlesefehler in den Container schreiben
Container c(it_cin, eof);
```

```
// Auslesen
// Ausgabestrom (hier: fuer Datei) anlegen
ofstream out(dateiname.c_str());

// Ausgabestrom-Iterator daran binden
// Der zweite Parameter fügt jeder
// (später folgenden) Elementausgabe
// ein ":" an
ostream<C_Elem> it_out(out, ":");

// Wie gehabt:
// Eingabestrom-Iterator an einen Eingabestrom binden
// und generischen End-Iterator definieren
istream<C_Elem> it2_cin(cin), eof2;

// Kopierprogramm cin --> out!
while (it2_cin != eof2)
    *it_out++ = *it2_cin++;
```

Unabhängig von Container/Datentyp enthält die Bibliothek `<algorithm>`
eine Sammlung von Algorithmen, die auf Operatorsequenzen nach
folgendem Schema operieren

```
algo(it.first ,it.last ,[Schlüssel/Wert/Praed. p],[it.ziel])
while(it.first != it.last) {
    if(p(*it.first)) { /* Aktion */
        ++it.first;
    }
}
```

`algo` ist durch den Namen der entsprechenden Bibliotheksfunktion zu ersetzen;
verlangt der Algorithmus ein Prädikat `p`, kann dies durch einen beliebigen
`#P FUNKTOR`, also (λ)-Funktion/Funktionsobjekt/-Zeiger realisiert werden)

Einteilung in:

- ➊ lesende/nicht verändernde Algorithmen
- ➋ schreibende/verändernde Algorithmen
- ➌ sortieren/spezielle Aufgaben

Online-Referenz z.B. www.cplusplus.com/reference/algorithm

92  I.3c Funktionale Programmierung
in C++: Algorithmen für Iteratorsequenzen — lesend/nichtverändernd

Algorithmus	Header	siehe Abschnitt	Seite
<code>adjacent_find</code>	<code><algorithm></code>	23.3.5	728
<code>all_of</code>	<code><algorithm></code>	23.3.17	707
<code>any_of</code>	<code><algorithm></code>	23.3.17	707
<code>count</code>	<code><algorithm></code>	23.3.16	706
<code>count_if</code>	<code><algorithm></code>	23.3.16	706
<code>equal</code>	<code><algorithm></code>	23.8.2	741
<code>find</code>	<code><algorithm></code>	23.3.1	721
<code>find_end</code>	<code><algorithm></code>	23.3.4	735
<code>find_first_of</code>	<code><algorithm></code>	23.3.3	723
<code>for_each</code>	<code><algorithm></code>	23.11.3	737
<code>ismatch</code>	<code><algorithm></code>	23.8.1	738
<code>none_of</code>	<code><algorithm></code>	23.3.17	707
<code>sample (+)</code>	<code><algorithm></code>	23.3.15	705
<code>split (+)</code>	<code><algorithm></code>	23.3.1	672

(*) kennzeichnet einen Algorithmus, zu dem es keine parallel ausführbare Variante gibt.

Quelle: Breynaar, Der C++ Programmierer

Algorithmus	Header	siehe Abschnitt	Seite
<code>accumulate (+)</code>	<code><numeric></code>	23.3.5	695
<code>adjacent_difference</code>	<code><numeric></code>	23.3.9	698
<code>choose (+)</code>	<code><numeric></code>	23.3.12	748
<code>gcd (+)</code>	<code><numeric></code>	23.3.13	769
<code>inner_product (+)</code>	<code><numeric></code>	23.3.7	696
<code> iota (+)</code>	<code><numeric></code>	23.3.4	694
<code>lcm (+)</code>	<code><numeric></code>	23.3.13	768
<code>max (+)</code>	<code><algorithm></code>	23.3.11	767
<code>min (+)</code>	<code><algorithm></code>	23.3.11	767
<code>minmax (+)</code>	<code><algorithm></code>	23.3.11	767
<code>partial_sum</code>	<code><numeric></code>	23.3.8	697

(*) kennzeichnet einen Algorithmus, zu dem es keine parallel ausführbare Variante gibt.

93  I.3c Funktionale Programmierung
in C++: Algorithmen für Iteratorsequenzen — lesend/nichtverändernd

```
typedef list<int> Container;
typedef Container::iterator C_Iter;
typedef Container::value_type C_Elem;

// Element-Funktionen
void printValues(C_Elem x) { cout << x << " " ; }
class IsMultipleOf {
public:
    unsigned int divisor;
    IsMultipleOf(unsigned int n): divisor(n) {}
    bool operator()(int x) const { return !(x%divisor); }
};

Container c = { 0, 1, 4, 9, 16, 25, 36, 49, 64, 81 };
for_each(c.begin(), c.end(), printValues); // for_each
// count_if
long n = count_if(c.begin(), c.end(), IsMultipleOf(3));

C_Iter it=c.begin();
for (int i=0; i<n; ++i) {
    it = find_if(it, c.end(), IsMultipleOf(3));
    cout << *it++ << " " ;
} // 9 36 81
```



1.3c Funktionale Programmierung in C++: Algorithmen für Iteratorsequenzen — schreibend

Tabelle 29.8: Verlinkende Algorithmen auf Folgen

Algorithmus	Header	seite Abschnitt	Seite
copy	<algorithm>	23.1.5	758
copy_n	<algorithm>	23.1.5	760
copy_if	<algorithm>	23.1.5	760
copy_backward (>)	<algorithm>	23.1.5	758
fill	<algorithm>	23.3.2	693
fill_n	<algorithm>	23.3.2	693
generate	<algorithm>	23.3.3	694
generate_n	<algorithm>	23.3.3	694
iter_swap (l, r)	<algorithm>	23.1.6	761
move	<algorithm>	23.2.1	791
move_backward (>)	<algorithm>	23.2.1	791
remove	<algorithm>	23.1.9	764
remove_if	<algorithm>	23.1.9	764
remove_copy	<algorithm>	23.1.9	764
remove_copy_if	<algorithm>	23.1.9	764
replace	<algorithm>	23.1.8	763
replace_if	<algorithm>	23.1.8	763
replace_copy	<algorithm>	23.1.8	763
replace_copy_if	<algorithm>	23.1.8	763
reverse	<algorithm>	23.3.14	704
reverse_copy	<algorithm>	23.3.14	704
rotate	<algorithm>	23.3.11	701
rotate_copy	<algorithm>	23.3.11	701
shuffle (l, r)	<algorithm>	23.3.12	702
stable_partition	<algorithm>	23.4.1	712
swap	<utility>	23.1.6	761
swap_ranges	<algorithm>	23.1.6	761
transform	<algorithm>	23.1.7	761
unique	<algorithm>	23.3.13	702
unique_copy	<algorithm>	23.3.13	702

(*) kennzeichnen einen Algorithmus, zu dem es keine parallel ausführbare Variante gibt.



1.3c Funktionale Programmierung in C++: Algorithmen für Iteratorsequenzen — schreibend

ALGO_SCHREIBEND

```

typedef list<int> Container;
typedef Container::iterator C_Iter;
typedef Container::value_type C_Elem;

// generische Ausgabe von Containerinhalten
// Die Nutzung von template wird im Abschnitt
// generische/generative Programmierung behandelt
template<typename Container>
void info(const Container& c) {
    cout << c.size() << " | ";
    copy(c.begin(), c.end(), ostream_iterator<C_Elem>(cout, " ") );
    cout << " | " << endl; }

Container c = {1, 4, 9, 16, 25, 36, 49, 64, 81, 100};
// ... 4 durch 9 ersetzen:
replace(c.begin(), c.end(), 4, 9); info(c);
// ... aufeinanderfolgende Duplicata ersetzen:
C_Iter it=unique(c.begin(), c.end()); info(c);
// ... Teil mit ungültigen Werten löschen
c.erase(it, c.end()); info(c);
// ... Wert 16 löschen
it=remove(c.begin(), c.end(), 16); info(c);
// ... it jetzt neuer Enditerator

```



1.3c Funktionale Programmierung in C++: Algorithmen für Iteratorsequenzen — schreibend

Der Algorithmus `it=generate_n(first, n, gen);` verwendet ein Funktionsobjekt `gen`, um damit `n` generierte Werte an die Position `first` im Zielcontainer einzusetzen. `it` liefert die Position hinter der eingefügten Sequenz.

Übung: Fügen Sie die ersten 10 Fibonacci-Zahlen in den Container `c` ein

```

class Fib {
    int f1 {0}; // Gedaechtnis – vorletzte Fibonacci-Zahl
    int f2 {1}; // Gedaechtnis – letzte Fibonacci-Zahl
public:
    /* Berechnung der naechsten Fibonacci-Zahl */
    unsigned int operator ()(); // --> fib-generativ.cpp
};

// Standard-Akzuerungen
typedef list<int> Container;
typedef Container::iterator C_Iter;
typedef Container::value_type C_Elem;

Container c; // noch leer!
// Hier Code anfuegen ...

```

UEBUNG_CONTAINERFUELLEN



1.3c Funktionale Programmierung in C++: Algorithmen für Iteratorsequenzen — suchen/sortieren/etc



1.3c Funktionale Programmierung

in C++: Algorithmen für Iteratorsequenzen — suchen/sortieren/etc

Algorithmus	Header	Code Beispiel	Seite
binary_search (l, r)	<algorithm>	23.3.7	718
equal_range (l, r)	<algorithm>	23.3.7	718
greater	<algorithm>	23.4.1	719
greater_equal	<algorithm>	23.4.4	719
is_heap	<algorithm>	23.3.3	718
is_heap_if	<algorithm>	23.3.3	718
is_permutation (l, r)	<algorithm>	23.3.8	718
is_sorted	<algorithm>	23.4.2	718
is_sorted_until	<algorithm>	23.4.2	718
is_lexicographical_compare	<algorithm>	23.3.9	718
lower_bound (l, r)	<algorithm>	23.3.7	718
max (l, r)	<algorithm>	23.1.12	717
max (l, r, comp)	<algorithm>	23.1.10	717
max_element	<algorithm>	23.3.8	718
max_heap	<algorithm>	23.4.4	718
max_heap_if	<algorithm>	23.4.4	718
min (l, r)	<algorithm>	23.3.8	718
min_element	<algorithm>	23.3.8	718
minmax (l, r)	<algorithm>	23.1.10	717
next_permutation (l, r)	<algorithm>	23.3.8	718
nth_element	<algorithm>	23.4.5	717
partial_sort	<algorithm>	23.4.4	718
partial_sort_copy	<algorithm>	23.4.3	718
partition	<algorithm>	23.4.3	718
partition_copy	<algorithm>	23.4.3	718
partition_point (l, r)	<algorithm>	23.4.1	718
pop_heap (l, r)	<algorithm>	23.3.2	718
prev_permutation (l, r)	<algorithm>	23.3.8	718
random_shuffle (l, r)	<algorithm>	23.4.12	717
search (l, r, s)	<algorithm>	23.5.3	723
search (l, r, s, nops, Algorithm)	<algorithm>	23.5.4	723
set_difference (l, r, s)	<algorithm>	23.6.4	723
set_intersection (l, r, s)	<algorithm>	23.6.3	723
set_symmetric_difference (l, r, s)	<algorithm>	23.6.3	723
set_union (l, r, s)	<algorithm>	23.6.2	723
sort (l, r)	<algorithm>	23.4.2	718
stable_partition (l, r)	<algorithm>	23.4.1	718
stable_sort (l, r)	<algorithm>	23.4.3	718
upper_bound (l, r)	<algorithm>	23.3.7	718

(*) kennzeichnen einen Algorithmus, zu dem es keine parallel ausführbare Variante gibt.



CONTAINERBEZOGENE UNZULÄNGLICHKEITEN

- Nicht jeder Container besitzt dieselben Methoden/Methodennamen
- Beispiel (Folie 86): Einige Einfügeiteratoren nutzen `push_front`

UNZULÄNGLICHKEITEN BEZÜGLICH `CONTAINER::VALUE_TYPE`

- `Container::value_type`-Daten müssen neben der Standardfunktionalität Ein-/Ausgabe, Überprüfung auf Gleichheit, Überprüfung der Anordnung oft zusätzliche Methoden unterstützen
- Beispiel (Folie 93): Im Prädikat `IsMultipleOf` den %-Operator, bei numerischen Algorithmen oft auch +, ++ ...; denkbar auch: `length()` (z.B. bei Strings)

Themenfeld: GENERISCHE/GENERATIVE PROGRAMMIERUNG



LIVE_QUICKSORT

Erinnerung: funktionaler Quicksort in Haskell:

```
cc
83 -- Quicksort
84 quickSort [] = []
85 quickSort (x:xs) =
86   quickSort [y | y <- xs, y<=x] ++
87   [x] ++
88   quickSort [y | y <- xs, y>x]
89
```

Die C++ Implementation soll "rein" funktional sein, im Sinne von

- übergeben wird eine zu sortierende Iteratorsequenz numerischer Werte (bzw. `Container::value_type` muss `operator<` unterstützen)
- der zugehörige Container unterstützt möglicherweise keine Random-Access-Iteratoren, sondern nur bidirektionale
- die Verwendung von Schleifenkonstruktionen ist nicht erlaubt (stattdessen: Rekursion)
- die Verwendung geeigneter Algorithmen aus der Bibliothek `algorithm` ist im Rahmen der obigen Vorgaben zugelassen



- ① Prinzipien und Paradigmen des Softwareentwurfs
 - Übergeordnete Architekturprinzipien
 - Allgemeine Programmierparadigmen
- ② Objektorientierte Programmierung
 - Paradigmen der objektorientierten Programmierung
 - Entwurfsmuster
- ③ Funktionale Programmierung
 - Paradigmen der funktionalen Programmierung
 - Container
 - Iteratoren und Algorithmen
 - Vertiefung: Funktoren
 - Vertiefung: Rekursion
- ④ Generische / Experimental Programmierung
 - generische Programmierung
 - generative Programmierung

1.3d Funktionale Programmierung in C++: λ -Funktionen

O. Henkel

FUNKTOREN IN CPP

λ -Funktionen in C++ sind anonyme Funktionsobjekte:

```
[l] ( Parameterliste ) -> Rueckgabety { Funktionsdefinition; return ...; }
```

Sie sind nebenwirkungsfrei, da sie keinerlei Zustandsinformationen aus der Umgebung verwenden oder verändern können und werden direkt an der Stelle ihrer Verwendung definiert:

```
list<int> l = { 1, -5, -7, 3, 4, 3, 2, 2, -4};
list<int>::iterator it;
for (it=l.begin(); it!=l.end(); it++) cout << *it << " ";
cout << endl; // 1 -5 -7 3 4 3 2 2 -4

l.sort([](int i, int j) { return (abs(i)<abs(j)); });
// ^-- Rueckgabetyp vom Compiler ermittelt
for (it=l.begin(); it!=l.end(); it++) cout << *it << " ";
cout << endl; // 1 2 2 3 3 4 -4 -5 -7
```

Es sind u.a. noch diverse (nebenwirkungsbefreite(!)) Varianten möglich, um auf Variablen des aktuellen Sichtbarkeitsbereiches (Scope) zuzugreifen:

- | | |
|-------------|--|
| [] | Standard |
| [&x] | alle sichtbaren Variablen per Referenz zugänglich |
| [&x] | nur x ist per Referenz zugänglich |
| [=] | alle sichtbaren Variablen werden per Wert kopiert |
| [=, &x, &y] | x und y sind per Referenz zugänglich alle anderen per Wert |
| [&x, x, y] | x und y sind per Wert zugänglich alle anderen per Referenz |
| [this] | Referenzzugriff auf alle Objektvariablen |
| [*this] | Wertzugriff auf alle Objektvariablen |

1.3d Funktionale Programmierung in C++: λ -Funktionen

O. Henkel

EINSCHUB: FUNKTIONSABSCHLUSS (FUNCTION CLOSURE)

...bezeichnet das Konzept, einer (λ -)Funktion Zugriff auf die sichtbaren Variablen der Umgebung zu gewähren, dies selbst aber noch in einer Funktion zu kapseln, so dass der Zustand von außen nicht sichtbar ist.

```
int verdopple() {
    int anf=5;
    int erg = [anf]()>>int { return anf+anf; }();
    return erg;
}

int main() {
    verdopple();
    return 0;
}
```

ÜBUNG |

Ändern Sie die Implementation des Quicksort so ab, dass bei der Partitionierung anstelle eines Prädikats eine λ -Funktion verwendet wird.



Funktionsobjekte sind Klassen mit einer öffentlichen Überladung von `operator()`

```
/*
 * binares Prädikat zum Vergleich von Listenelementen
 */
class BetragKleiner {
public:
    bool operator()(int i, int j) { return (abs(i)<abs(j)); }
};

int main() {
    list<int> l = { 1, -5, -7, 3, 4, 3, 2, 2, -4};
    list<int>::iterator it;
    for (it=l.begin(); it!=l.end(); it++) cout << *it << " ";
    cout << endl; // 1 -5 -7 3 4 3 2 2 -4

    l.sort( BetragKleiner() );

    for (it=l.begin(); it!=l.end(); it++) cout << *it << " ";
    cout << endl; // 1 2 2 3 3 4 -4 -5 -7

    return 0;
}
```

Im Gegensatz zu λ -Funktionen und Funktionsobjekten bieten Funktionszeiger die Möglichkeit, erst zur Laufzeit die zu verwendende (λ -)Funktion/Funktionsobjekt auszuwählen.

BEISPIEL |

Ein eigener Algorithmus (z.B. `mySort`) benötigt ein Prädikat / Funktionsobjekt (z.B. `istKleiner`), das erst zur Laufzeit ausgewählt wird

Für die Umsetzung unterscheidet man zwei Möglichkeiten:

- **Funktionszeiger**
durchläuft z.B. ein Feld von Funktionsadressen, sodass zur Laufzeit (ggf. durch Benutzeraktion) ein Funktionsaufruf stattfindet.
- **Methodenzeiger**
Eine Methode soll für viele, erst zur Laufzeit bekannte Objekte aufgerufen werden. Objekte rufen erst zur Laufzeit festgelegte Methoden auf.



Voraussetzung: Verschiedene Funktionen mit gleicher Parameterliste stehen zur Verfügung:

- `void f(Param param);`
- `void g(Param param);`

Ein Funktionszeiger wählt die Funktion zur Laufzeit aus:

<code>void (*pFkt) (Param) = &f;</code>	<i>// Zeiger auf eine Funktion mit Parameter vom Typ Param und Rückgabetyp void</i>
<i>// Alle Klammern notwendig!</i>	
<code>(*pFkt) (param);</code>	<i>// entspricht Aufruf f(param)</i>
<code>pFkt=&g;</code>	<i>// Zeigerzuweisung zu g</i>
<code>(*pFkt) (param);</code>	<i>// entspricht Aufruf g(param)</i>

```
/*
 * zwei Vergleichsfunktionen mit gleicher Signatur
 */
bool betragKleiner(int i, int j) { return ( abs(i)<abs(j) ); }
bool betragGroesser(int i, int j) { return ( abs(i)>abs(j) ); }

int main() {
    list<int> l = { 1, -5, -7, 3, 4, 3, 2, 2, -4};
    list<int>::iterator it;
    for (it=l.begin(); it!=l.end(); it++) cout << *it << " ";
    cout << endl; // 1 -5 -7 3 4 3 2 2 -4

    // Auswahl zur Laufzeit
    int i; cin >> i;
    bool (*pVergleich)(int, int);
    if (i%2==0) pVergleich = betragKleiner;
    else pVergleich = betragGroesser;

    l.sort(pVergleich);

    for (it=l.begin(); it!=l.end(); it++) cout << *it << " ";
    cout << endl; // 1 2 2 3 3 4 -4 -5 -7
    // oder: -7 -5 4 -4 3 3 2 2 1
    return 0; // ^--^ (nicht stabil)
}
```

Voraussetzung: Eine Klasse mit verschiedenen Methoden gleicher Parameterliste steht zur Verfügung:

```
class K {
public:
    void f(Param);
    void g(Param);
};
```

- Methodenzeiger wählt Methode der Klasse:

```
void (K::*pMethod)(Param) = &K::f
```

- Aufruf durch Bindung an ein Objekt:

```
K eins, zwei;
(eins.*pMethod)(param); // entspricht: eins.f(param)
K *pzwei = &zwei;
(pzwei->pMethod)(param); // entspricht zwei.f(param)
```

- Wahl einer anderen Methode:

```
pMethod=&K::g;
```

```
/*
 * binares "Praedikat" zum Vergleich ganzer Zahlen
 * nach der Groesse ihres Restes.
 * Die Klasse stellt Methoden fuer verschiedene
 * Vergleichskriterien bereit
 */
class Rest {
    int n; // "Zustand" des Funktionsobjektes
public:
    Rest(int _n=1) : n(_n) {}
    int gibN() { return n; }
    bool kleiner(int i, int j) { return ((i%n) <= (j%n)); }
    bool groesser(int i, int j) { return ((i%n) >= (j%n)); }
};
```

```
/*
 * SimpselSort—Algorithmus auf einem Container von int—Zeigern,
 * der ein Rest—Objekt (praedikat)
 * mit einer Rest—Methode (pCmp) verbindet,
 * um den Vergleich zweier int—Werte vorzunehmen
 */
void mySort.pmethod(vector<int*>& vec,
                     Rest& praedikat,
                     bool (Rest::*pCmp)(int, int)) {
    // Einfacher SimpselSort—Algorithmus
    for (size_t i=0; i<vec.size(); i++)
        for (size_t j=i+1; j<vec.size(); j++)
            if (!(*praedikat.*pCmp)(*vec[i], *vec[j])) {
                // — Dereferenzierte Methode,
                // — an Objekt praedikat gebunden
                int* hilf=vec[i];
                vec[i]=vec[j];
                vec[j]=hilf;
            }
}
```

```
Original: 10| 1 2 3 4 5 6 7 8 9 10
sortiere 'kleiner';
modulo 2| 10| 2 4 6 8 10 3 7 1 9 5
modulo 3| 10| 2 5 8 10 3 6 9 4 7 1
modulo 10| 10| 3 5 2 8 4 6 9 7 8 9 10

sortiere 'grosserer';
modulo 2| 10| 1 3 5 7 9 6 4 8 2 10
modulo 3| 10| 5 8 2 4 1 7 10 9 6 3
modulo 10| 10| 9 6 7 5 4 3 2 1

Einen Vektor mit Zeigern darauf erzeugen
// Nur dieser wird durch die nachfolgenden Algorithmen veraendert
vector<int*> pWerte(nWerte);
for (int i=0; i<nWerte; i++) pWerte[i]=&werte[i];

Vektor sortieren mittels Methodenzeiger

zunachst werden Vergleichs—Objekte parametrisiert
Rest praedikat[3] = {Rest(2), Rest(3), Rest(100)};

Jetzt die Vergleichsmethode der Klasse auswaehlen
bool (Rest::*vergleich)(int,int) = &Rest::kleiner;

Start: Durchlaufe alle Objekte mit der gewaehlten Methode
sortiere 'kleiner'.
for (int i=0; i<3; i++) mySort.pmethod(pWerte, praedikate[i], vergleich);

Wechselt das Vergleichskriteriums
vergleich = &Rest::grosserer;
sortiere 'grosserer'.
for (int i=0; i<3; i++) mySort.pmethod(pWerte, praedikate[i], vergleich);
```



Übung

```
int Antwort(const char * text) {
    cout << text;
    return 42;
}
```

Schreiben Sie ein Programm, das diese Funktion über einen Funktionszeiger aufruft und den Rückgabewert auf dem Bildschirm ausgibt

[UEBUNG „ANTWORT“ 42](#)



Uebung: OK oder nicht OK?

```
void (*pf)(char *);

void f1(char *);
char *f21(char *);
char* f22(char *);
void f3(double *);

int main() {
    pf=f1;
    pf=&f1;
    pf=&f21;
    pf=f3;
    *pf(42);
    *pf("Die Antwort ist ");

    return 0;
}
```

ALGORITHMUS ZUR ENTSCHEIDUNG VON ZEIGERDEKLARATIONEN

- ➊ Beginn ist der Variablenname
- ➋ Interpretation von links nach rechts
- ➌ Bei Erreichen einer schließenden runden Klammer wird von rechts nach links interpretiert
- ➍ Bei Erreichen einer öffnenden runden Klammer „von rechts“ wird wieder von links nach rechts interpretiert, beginnend nach der zugehörigen geschlossenen runden Klammer
- ➎ Ist das Ende des Ausdrucks erreicht, werden die noch verbliebenen Teile von rechts nach links interpretiert

```
double (**)[100]; /* Zeiger auf ein double-Feld */
double *x[100]; /* Feld von double-Zeigern */
/* Feld von Fktszeigern, die char zurueckliefern
   und als 1. Argument integer erwarten und als
   2. Argument einen Fktszeiger auf eine Fkt.
   mit Rueckgabetyp int und zwei double-Arumenten */
char (*f[10])(int, int (*g)(double, double));
```

ALGORITHMUS ZUR ENTSCHEIDUNG VON ZEIGERDEKLARATIONEN

- ➊ Beginn ist der Variablenname
- ➋ Interpretation von links nach rechts
- ➌ Bei Erreichen einer schließenden runden Klammer wird von rechts nach links interpretiert
- ➍ Bei Erreichen einer öffnenden runden Klammer „von rechts“ wird wieder von links nach rechts interpretiert, beginnend nach der zugehörigen geschlossenen runden Klammer
- ➎ Ist das Ende des Ausdrucks erreicht, werden die noch verbliebenen Teile von rechts nach links interpretiert

```
/* f: Fktszeiger, Rueckgabetyp: int,
   Argument: Fktszeiger ohne Param.,
   der int zurueckliefert */
int (*f)( int (*g)() );
/* f: Funktion mit Rueckgabetyp: int (*())
   Argument: Wie eben */
int (*f( int (*g)() ))();
```

ALGORITHMUS ZUR ENTSCHEIDUNG VON ZEIGERDEKLARATIONEN

- ➊ Beginn ist der Variablenname
- ➋ Interpretation von links nach rechts
- ➌ Bei Erreichen einer schließenden runden Klammer wird von rechts nach links interpretiert
- ➍ Bei Erreichen einer öffnenden runden Klammer „von rechts“ wird wieder von links nach rechts interpretiert, beginnend nach der zugehörigen geschlossenen runden Klammer
- ➎ Ist das Ende des Ausdrucks erreicht, werden die noch verbliebenen Teile von rechts nach links interpretiert

```
/* einfacher mit typedef */
typedef int (*Fkt)();

/* f: Fktszeiger, Rueckgabetyp: int,
   Argument: Fktszeiger ohne Param., der int zurueckliefert */
int (*f)(Fkt);

/* f: Funktion mit Rueckgabetyp: int (*())
   Argument: Wie eben */
Fkt f(Fkt);
```

int a

int a	integer-Variable
int a[n]	Feld von integer-Werten
int a[n][m]	2-dimensionales Feld von integer-Werten
int *a	Zeiger auf integer
int **a	Zeiger auf Zeiger auf integer
int *a[n]	Feld von integer-Zeigern
int (*a)[n]	Zeiger auf ein Feld von integer-Werten
int a()	Funktion ohne Parameter mit Rückgabetyp int
int *a()	Funktion ohne Parameter mit Rückgabetyp int *
int (*a)()	Zeiger auf eine Funktion ohne Parameter mit Rückgabetyp int
int ***a()	Funktion ohne Parameter mit Rückgabetyp int **
int (*(*a))()	Zeiger auf eine Funktion ohne Parameter mit Rückgabetyp int *
int (**a)()	Zeiger auf Zeiger auf eine Funktion ohne Parameter mit Rückgabetyp int
int **(**a)()	Zeiger auf Zeiger auf eine Funktion ohne Parameter mit Rückgabetyp int **

ÜBUNG | Welchen Typ hat g?

`int *(*(*g)[10])(char *,char *);`



- ➊ Prinzipien und Paradigmen des Softwareentwurfs
 - ➎ Übergeordnete Architekturprinzipien
 - ➎ Allgemeine Programmierparadigmen
- ➋ Objektorientierte Programmierung
 - ➎ Paradigmen der objektorientierten Programmierung
 - ➎ Entwurfsmuster
- ➌ Funktionale Programmierung
 - ➎ Paradigmen der funktionalen Programmierung
 - ➎ Container
 - ➎ Iteratoren und Algorithmen
 - ➎ Vertiefung: Funktoren
 - ➎ Vertiefung: Rekursion
- ➍ Generische / Generative Programmierung
 - ➎ generische Programmierung
 - ➎ generative Programmierung



Eine Funktion heißt **rekursiv**,

wenn sie sich direkt oder indirekt selbst aufruft.

Rekursive Funktionen sind immer dann nützlich, wenn sich der Algorithmus in kleinere, aber gleichbleibende Abläufe zerlegen lässt.

BEISPIEL |

Die Fakultät $n! = n * (n - 1) * (n - 2) * \dots * 2 * 1$ einer nicht negativen ganzen Zahl hat eine äquivalente rekursive Beschreibung durch zwei Regeln: 1. $0! = 1$, 2. $n! = n * (n - 1)!$

Fakultät iterativ

```
int fak(unsigned int n){
    int i, ergebnis=1;
    for (i=2; i<=n; ++i)
        ergebnis *= i;
    return ergebnis;
}
```

Iterative Definition für nicht negative ganze Argumente n .



Eine Funktion heißt **rekursiv**,

wenn sie sich direkt oder indirekt selbst aufruft.

Rekursive Funktionen sind immer dann nützlich, wenn sich der Algorithmus in kleinere, aber gleichbleibende Abläufe zerlegen lässt.

BEISPIEL |

Die Fakultät $n! = n * (n - 1) * (n - 2) * \dots * 2 * 1$ einer nicht negativen ganzen Zahl hat eine äquivalente rekursive Beschreibung durch zwei Regeln: 1. $0! = 1$, 2. $n! = n * (n - 1)!$

Fakultät rekursiv

```
int fak(unsigned int n){
    int ergebnis;
    if (n==0)
        ergebnis=1;
    else
        ergebnis=n*fak(n-1);
    return ergebnis;
}
```

Rekursive Definition besteht aus zwei Teilen

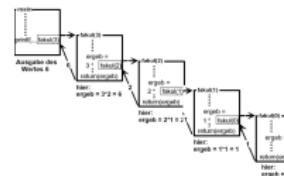
➊ **Basisfall** (hier $n == 0$): direkt lösbar, beendet die Rekursion

➋ **Rekursionsschritt**:
Algorithmus mit Rückgriff auf "einfachere" Teilprobleme, die auf den Basisfall führen



Fakultät rekursiv

```
int fak(unsigned int n){
    int ergebnis;
    if (n==0)
        ergebnis=1;
    else
        ergebnis=n*fak(n-1);
    return ergebnis;
}
```



Quelle: H. Heindl, B. Lutz, J. Wohlbach: Grundlagen der Informatik

STACK UND REKURSIVE FUNKTIONEN

- Bei jedem Aufruf einer Funktion wird auf dem Stack Speicher für die Parameter und die lokalen Variablen belegt
- Bei rekursiven Funktionen erhält jede aufgerufene **Instanz** der Funktion Speicherplatz auf dem Stack!
- Das gilt für alle Parameter und lokalen Variablen der rekursiven Funktion



Was passiert beim Aufruf `fak(-1)`?

```
int fak(unsigned int n){
    int ergebnis;
    if (n==0)
        ergebnis=1;
    else
        ergebnis=n*fak(n-1);
    return ergebnis;
}
```

```
int fak (int n) {
    int ergebnis;
    if (n<0) return -1;
    if (n==0)
        ergebnis=1;
    else
        ergebnis=n*fak(n-1);
    return ergebnis;
}
```

Parameter vom Typ `unsigned int`, also wird die Fakultät von $2^{32} - 1$ berechnet

Dazu sind $2^{32} - 1$ Funktionsaufrufe mit jeweils Speicherplatz für lokale Variablen und Parameter nötig!

Ist dieses Verhalten unerwünscht können negative Werte durch Verwendung des Datentyps `int` abgefangen und ein Fehlercode (hier `-1`) zurückgegeben werden.

Ohne die neu eingefügte Abfrage ergäbe sich eine Endlosrekursion!
Der Rekursionsschritt muss schließlich stets in einen der Basisfälle münden



```
#include <iostream>
#include <string>
using namespace std;

string reverse(const string& s) {
    // HIER: rekursive Implementation einfügen
}

int main() {
    string s {"Hallo"};
    string t = reverse(s);

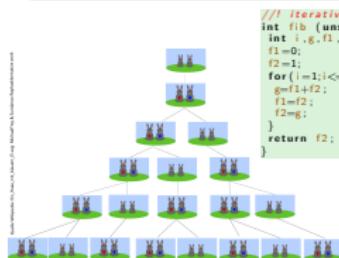
    cout << s << endl; // "Hallo"
    cout << t << endl; // "ollaH"

    return 0;
}
```



DIE FIBONACCI-FOLGE

1. Es gibt ein Paar neugeborener Kaninchen
2. Jedes neugeborene Paar wird im zweiten Lebensmonat geschlechtsreif
3. Jedes geschlechtsreife Paar wirft pro Monat ein weiteres Paar

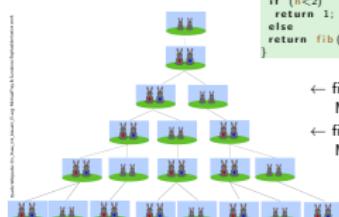


```
/// iterative Formulierung
int fib(unsigned int n) {
    int i, g, f1, f2;
    f1=0; // reproduktionsfaehige Paare
    f2=1; // Gesamtzahl Paare
    for(i=1;i<n;i++) { /* nach dem i-ten Monat: */
        g=f1+f2; // Berechnung neue Gesamtzahl
        f1=f2; // reproduktionsfaehige Paare
        f2=g; // Gesamtzahl Paare
    }
    return f2;
}
```



DIE FIBONACCI-FOLGE

1. Es gibt ein Paar neugeborener Kaninchen
2. Jedes neugeborene Paar wird im zweiten Lebensmonat geschlechtsreif
3. Jedes geschlechtsreife Paar wirft pro Monat ein weiteres Paar



```
/// rekursive Formulierung
int fib(unsigned int n){
    if (n<2) {
        return 1; // Basisfall
    } else
        return fib(n-1)+fib(n-2); // Rekursionsschritt
}
```

← fib(n-2) Paare sind im nächsten Monat reproduktionsfähig
 ← fib(n-1) Paare sind auch im nächsten Monat noch da



Ökonomiegewinn

Rekursive Algorithmen lassen sich oft kompakter und übersichtlicher als iterative Algorithmen formulieren



ÜBUNG |

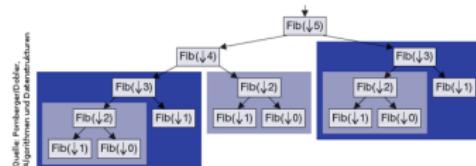
Schreiben Sie eine rekursive Funktion binomi die zu zwei Zahlen n und k die Binomialkoeffizienten berechnet gemäß

$$\bullet \quad \binom{n}{0} = \binom{n}{n} = 1$$

$$\bullet \quad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$



Aufrubaum für den Aufruf von fib(5)



Die Grafik zeigt, dass die Anzahl der Funktionsaufrufe rasant zunimmt, da Teilergebnisse (im Bild mit gleichfarbiger Hintergrundfarbe hinterlegt) nicht zwischengespeichert, sondern jeweils neu berechnet werden.

Die Anzahl der rekursiven Funktionsaufrufe für fib(n) wächst exponentiell mit n



Ökonomieverlust

- Die Anzahl der rekursiven Aufrufe kann unmäßig schnell zunehmen, insbesondere wenn mehrere rekursive Aufrufe im Funktionsrumpf stattfinden.
- Werden Zwischenergebnisse nicht wiederverwendet, so ist das Laufzeitverhalten ebenfalls ineffizient

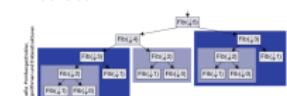


Verschiedene Rekursionstypen

Fakultät



Fibonacci



lineare Rekursion

Jeder Ausführungspfad der Funktion enthält höchstens einen rekursiven Aufruf

Rekursionsbaum

Mindestens ein Ausführungspfad enthält mindestens zwei rekursive Aufrufe

VON DER BAUMREKURSION ZUR LINEAREN REKURSION

... gelangt man, indem der Rekursionsbaum zunächst von der Wurzel bis zum untersten Blatt (linker Zweig) durchlaufen und beim Rückweg die Zwischenergebnisse nach oben durchgereicht werden!

„Die übrigen Zweige müssen nicht durchlaufen werden.“

(while)-Schleifen lassen sich stets auch rekursiv implementieren

iterativ

```
while (Bedingung) {
    Anweisungen
}
```

```
int fib (unsigned int n) {
    int g, f1, f2;
    f1=0; f2=1; // Basisfall speichern
    while (n>0) { // Anzahl Schritte
        g=f1+f2; // nächste Fib-Zahl
        f1=f2;
        f2=g; // Zwischenergebnisse
        n--;
        // Noch (n-1) Schritte
    }
    return f2;
}
```

rekursiv

```
void f(Parameter) {
    if (Bedingung) {
        Anweisungen
        f(Parameter)
    }
}
```

```
typedef unsigned int uint;
pair<uint, uint> // Bibliothek utility
fib1 (uint n, uint f1, uint f2) {
    if (n==0) return pair<uint, uint>(f1, f2);
    else return fib1(n-1, f2, f1+f2);
}

uint fib(uint n) {
    return fib1(n, 1, 1).first;
}
```

rekursive Funktionen = "while + Stapel"

ENTREKURSIVIERUNG REKURSIVER ALGORITHMEN

- Der rekursive Funktionsaufruf wird durch Verwendung eines Stapelspeichers simuliert, in dem diejenigen Daten abgelegt werden, die bei einem echten (rekursiven) Funktionsaufruf auf den Call-Stack abgelegt worden wären. Es wird also ein Stapel-Container zur Aufnahme der Funktionsparameter eingerichtet.
- Die ursprünglich rekursive Funktion wird dann so abgeändert, dass der Stapel abgearbeitet wird, solange er nicht leer ist.
- Jedes Element des Stapels enthält alle Daten, die die ursprüngliche Funktion als Parameter hatte, d.h. es kann jeweils entweder der Basisfall abgearbeitet oder die Anweisungsfolge des Rekursionsschrittes durchgeführt werden, wobei rekursive Funktionsaufrufe wieder durch Ablage der Funktionsparameter auf dem Stapel simuliert werden.

Entrekursivierung durch Nachbildung des Call-Stacks

```
Push(anfangsZustand); // Call-Stack sichern
while (Stapel nicht leer) {
    Pop(aktuellerZustand); // aktuellen Call-Stack holen
    if (Problem direkt lösbar)
        löse Problem; // Operationen des an den Blättern des
                        // rekursiven Aufrufbaums nachbilden
    else
        Push(Teilprobleme); // (Parameter der) Teilprobleme auf den
                            // Call-Stack (in umgekehrter Reihenfolge)
                            // ablegen
}
```

BEISPIEL |

- Fibonacci-Folge



```
///! rekursive Formulierung
int fib(unsigned int n){
    if (n<2)                                // Basisfall
        return 1;
    else
        return fib(n-1)+fib(n-2); // Rekursionsschritt
}
```



```
///! entrekursiviert durch Stapel
stack<int> callStack;
int fib(unsigned int n){
    // _____ Inhalt des Call-Stacks
    unsigned int ergebnis=0;
    unsigned int anfangsZustand=n;
    callStack.push(anfangsZustand); // Call-Stack sichern

    while (!callStack.empty()) {
        int aktuellerZustand = callStack.top();
        callStack.pop();           // aktuellen Call-Stack holen
        aktuellerZustand;

        if (n<2) {                // Problem direkt lösbar
            // Parameter auf dem Stack werden addiert
            // Jedes "Blatt" im rekursiven Aufrufbaum zählt 1
            ergebnis+=1;
        }
        else {
            // ---> Teilprobleme auf Stack
            callStack.push(n-1);
            callStack.push(n-2);
        }
    }
    return ergebnis;
}
```



Zusammenfassende Bemerkungen

- rekursive Algorithmen müssen immer einen Basisfall haben, der auf jeden Fall erreicht wird!
- Es gibt Problemstellungen, für die eine rekursive Formulierung am einfachsten und daher angemessen ist.
- Rekursive Algorithmen lassen sich oft sehr kompakt und übersichtlich formulieren.
- Nachteil ist oft die schlechte Performanz hinsichtlich Speicherbedarf und Laufzeit.
- Grundsätzlich sind rekursive Programmstrukturen zu Schleifen+Verwendung eines Stapel äquivalent.



- ① Prinzipien und Paradigmen des Softwareentwurfs
 - Übergeordnete Architekturprinzipien
 - Allgemeine Programmierparadigmen
- ② Objektorientierte Programmierung
 - Paradigmen der objektorientierten Programmierung
 - Entwurfsmuster
- ③ Funktionale Programmierung
 - Paradigmen der funktionalen Programmierung
 - Container
 - Iteratoren und Algorithmen
 - Vertiefung: Funktoren
 - Vertiefung: Rekursion
- ④ Generische / Generative Programmierung
 - generische Programmierung
 - generative Programmierung



- ① Prinzipien und Paradigmen des Softwareentwurfs
 - Übergeordnete Architekturprinzipien
 - Allgemeine Programmierparadigmen
- ② Objektorientierte Programmierung
 - Paradigmen der objektorientierten Programmierung
 - Entwurfsmuster
- ③ Funktionale Programmierung
 - Paradigmen der funktionalen Programmierung
 - Container
 - Iteratoren und Algorithmen
 - Vertiefung: Funktoren
 - Vertiefung: Rekursion
- ④ Generische / Generative Programmierung
 - generische Programmierung
 - generative Programmierung



Ziel: Entwicklung von allgemein einsetzbaren Algorithmen

Umsetzung

- Einführung einer Abstraktionsschicht (z.B. Iteratoren in der STL)
- generische Programmierung:
Algorithmen durch Typen parametrisieren



DIE STANDARD-TEMPLATE-LIBRARY (STL)

Die STL nutzt generische Definitionsmuster für ihre Container, Iteratoren und Algorithmen.

Generizität wird durch diesen Ansatz bereits zur Compilezeit hergestellt.

```
// ----- generische Ausgabe von Containerinhalten -----
template<typename Container>
void info(const Container& c) {
    cout << c.size() << endl;
    copy(c.begin(), c.end(), ostream_iterator<Container::value_type>(cout, " "));
    cout << endl;
```

**SCHABLOENEN**

Eine Schablone stellt ein Definitionsmuster zur Verfügung, das nach Bedarf zur Compilezeit Definitionen erzeugt.

`template<typename T> Definitionsmuster //haengt von T ab`

`template<int N> Definitionsmuster //haengt von N ab`

template → Schlüsselwort für die Erstellung eines Templates

typename → Schlüsselwort (alternativ: **class**)

T → Generischer Typbezeichner, der in **Definitionsmuster** anstelle eines konkreten Datentyps verwendet wird

Paradebeispiel

```
template<typename T>
void tausche(T& links, T& rechts) {
    T hilf {links};
    links = rechts;
    rechts = hilf;
}
```



// ----- typgerechte Konstanten -----

```
template<typename T>
constexpr T pi = T(3.141592653589793238462643383);
// ~ Compiler soll auswerten
cout << pi<int> << endl; // 3
cout << pi<float> << endl; // 3.14159
```

// ----- Felder vorgegebener Groesse -----

```
template<typename T, int N>
class MyArray {
    T wert[N];
};
MyArray<char,10> text;
```

// ----- identischer Algorithmus fuer Zahltypen -----

```
template<typename T>
T produkt(const T& a, const T& b) {
    return a*b;
}
cout << produkt(5.,3.) << endl; // 15
cout << produkt(5.,3) << endl; // geht nicht
```

**ANFORDERUNGEN AN DEN SCHABLONENPARAMETER**

Typische Anforderungen sind

- Vergleich (wo sinnvoll)
- Kopien anlegen, Zuweisung
- Umwandlung von/in Zahlenwerte
- Verknüpfungen wie +, -, <<
- semantische Anforderungen z.B. GENERATIVE PROGRAMMIERUNG

Es liegt in der Verantwortung des Programmierers, darauf zu achten, dass ein konkret eingesetzter Datentyp diese Anforderungen erfüllt



```
// allgemeine Schablone
template<typename T> void f(T a, T b) { /* ... */ }
// Schablonentyp wird beim Aufruf vom Aktualparameter abgelesen

// spezielle Schablone fuer int-Parameter
// (z.B. effizientere Implementation moeglich)
template<> void f<int>(int a, int b) { /* ... */ }
// Wird automatisch gewaehlt, wenn Aktualparametertyp=int

// Verwendung
f(3.5, 5.); // allgemeine Schablone, T=double
f(3, 5); // spezielle Schablone
f(3.5, 5); // geht nicht
f<double>(3.5, 5); // T=double, Aufruf allgemeine Schablone
f<int>(3.5, 5); // T=int, spezielle Schablone
```

Auswahlmechanismus beim Aufruf

spezialisierte Schablone kommt vor allgemeine Schablone
 ("speziell" akzeptiert weniger Parameter als "allgemein")
 Danach wird die am besten passende Schablone gewählt



```
// allgemeine Schablone
template<typename T> void f(T a, T b) { /* ... */ }

// spezielle Schablone fuer int-Parameter
template<> void f<int>(int a, int b) { /* ... */ }

// keine Schablone
void f(int a, int b) { /* ... */ }

// Verwendung
f(3.5, 5.); // allgemeine Schablone
f(3, 5); // keine Schablone
f<>(3, 5); // erzwingt Schablone: spezielle Schablone
f(3.5, 5); // keine Schablone, Typkonversion 3.5 --> int
f<>(3.5, 5); // geht NICHT: Ein Schablonenparameter erlaubt
// nur Typumwandlungen in EINEN Typ
```

Auswahlmechanismus beim Aufruf

keine Schablone	<u>kommt vor</u>	spezialisierte Schablone	<u>kommt vor</u>	allgemeinere Schablone
-----------------	------------------	--------------------------	------------------	------------------------



```
// allgemeinst Schablone
template<typename S, typename T>
void f(S a, T b) { /* ... */ }

// allgemeine Schablone
template<typename T>
void f(T a, T b) { /* ... */ }

// spezielle Schablone fuer int-Parameter
template<> void f<int>(int a, int b) { /* ... */ }

// Verwendung
f(3.5, 5.); // allgemeine Schablone
f(3, 5); // spezielle Schablone
f(3.5, 5); // allgemeinst Schablone
f<int>(3.5, 5); // spezielle Schablone
f<double>(3.5, 5); // S=double, T=int, allgemeinst Schablone
f<int>(3.5, 5.); // S=int, T=double, allgemeinst Schablone
f<double>(3.5, 5.); // T=double, allgemeine Schablone
f<double, int>(3.5, 5); // S=double, T=int; allgemeinst Schablone
```

Auswahlmechanismus beim Aufruf

speziellere Schablone kommt vor allgemeinere Schablone



Die Funktionsschablonen können überladen werden und damit weitere Varianten bereitstellen

```
template<typename T>
T produkt(const T& a, const T& b) {
    return a*b;
}

// ueberladene Schablone
template<typename T>
T produkt(vector<T>& v, vector<T>& w) {
    T erg=0;
    if (v.size()!=w.size()) return erg;
    for (size_t i=0; i<v.size(); i++)
        erg += produkt(v[i],w[i]);
    return erg;
}

// Verwendung
vector<int> v = {1,2,3,4,5}, w={3,2,5,2,1};
produkt(v,w); // 35
```



- ① Prinzipien und Paradigmen des Softwareentwurfs
 - Übergeordnete Architekturprinzipien
 - Allgemeine Programmierparadigmen
- ② Objektorientierte Programmierung
 - Paradigmen der objektorientierten Programmierung
 - Entwurfsmuster
- ③ Funktionale Programmierung
 - Paradigmen der funktionalen Programmierung
 - Container
 - Iteratoren und Algorithmen
 - Vertiefung: Funktoren
 - Vertiefung: Rekursion
- ④ Generische / Generative Programmierung
 - generische Programmierung
 - generative Programmierung



Ziel: Compiler führt Teile des Codes aus

Umsetzung

- rekursiv erzeugte Template-Typen
- Traits zur Abfrage von Typinformationen zur Compilezeit

Γ trait: Merkmal_...

Anwendung: ressourceneffiziente Programmierung

- Metaprogrammierung / bedingte Kompilierung
- Move-Semantik

Γ HTTP://WWW.CPLUSPLUS.COM/REFERENCE/

Γ HTTPS://EN.CPPREFERENCE.COM/W/

Γ Wird voraussichtlich benötigt:

using NeuerName = Typ ist eine Variante zu

typedef Typ NeuerName, die auch für Templates verwendbar ist ..



```

1 // ----- Fakultaet -----
2 template <int N>
3 struct FAK {
4     static const unsigned long wert { N*FAK<N-1>::wert };
5 };
6
7 template<>
8 struct FAK<> { // "Basisfall" der Rekursion
9     static const unsigned long wert { 1 };
10};
11 cout << FAK<15>::wert << endl; // 1307674368000

```

Der Compiler erzeugt nacheinander die Typen **FAK<N>** für **N** absteigend von 15 bis 0 und erzeugt in Zeile 11 den Wert $15*14*13*...*2*1*1$

Übung: Fibonaccizahlen

Γ UEBUNG_FIBTEMPLATE



```
1 // ----- groesster gemeinsamer Teiler -----
2 template <int N, int M>
3 struct GGT
4 {
5     static const int wert { GGT<M, N%M>::wert };
6 };
7
8 template <int M>
9 struct GGT<M, 0>
10 {
11     static const int wert { M };
12 };
13 cout << GGT<288,168>::wert << endl; // 24
```

Der Compiler erzeugt nacheinander die Typen `GGT<288,168>`,
`GGT<168,120>`, `GGT<120,48>`, `GGT<48,24>`, `GGT<24,0>` und ersetzt in Zeile 13 `wert` durch 24.



Erstellung eines Typs `Bruch` für rationale Zahlen inklusive generativer Implementation der Operationen Addition/Subtraktion/Kürzen

153 I.4b generative Programmierung

O. Henkel variable Parameteranzahl

TEMPLATE „ELLIPSE“

Mittels der Ellipse ... ist es seit C++11 möglich, eine variable Anzahl von Typparametern in einer Schablonendefinition zu halten:

```
1 template <typename ...Args>
2 int summieren(Args... args) {
3     return (args+...);
4 }
```

Die runden Klammern in Zeile 3 sind notwendig und kennzeichnen die Anwendung des binären Operators `operator+` als einen „Fold-Expression“ (C++17), der nacheinander abgearbeitet wird.

Eine listenartige Bearbeitung einer variablen Anzahl von Schablonenparametern kann wie folgt erreicht werden:
 ↑ denn der Kommaoperator in Zeile 3 ist ebenfalls ein binärer Operator..

```
1 template <typename Funktor, typename ...Args>
2 void anwenden(Funktor f, Args... args) {
3     (f(args), ...);
4 }
5
6 int main() {
7     int erg=0;
8     anwenden([&erg](int x){ erg+=x*x; }, 2, 4, 7, 4, 3, 1, 7);
9     cout << erg << endl; // 144
```

154 I.4b generative Programmierung

O. Henkel Kontrollstrukturen

TEMPLATE „KONTROLLSTRUKTUREN“

```
// Verzweigung / bedingte Kompilierung
template<bool b>
struct IFELSE {};
```

```
template<>
struct IFELSE<true> { static void f() { cout << "true"; } };
template<>
struct IFELSE<false> { static void f() { cout << "false"; } };

const int n=1;
IFELSE<(n>0)>::f(); cout << endl; // "true"

// Schleifen
template<unsigned int N>
struct DOWHILE {
    static void f() {
        cout << "!";
        DOWHILE<((N-1)>0) ? (N-1) : 0 >::f();
    }
};

template<>
struct DOWHILE<0> { static void f() {} };

DOWHILE<5>::f(); cout << endl; // "!!!!!"
```

155 I.4b generative Programmierung

O. Henkel Compilezeit-Polymerphismus

TEMPLATE „POLYMORPHIE“

```
// dynamische Polymerphie
class Abstrakt {
public:
    virtual void f() = 0;
    virtual void g() = 0;

    void h() { f(); g(); }
};

class U: public Abstrakt {
public:
    void f() { cout << "U::f()"; }
    void g() { cout << "U::g()"; }
};

class V: public Abstrakt {
public:
    void f() { cout << "V::f()"; }
    void g() { cout << "V::g()"; }
};

Abstrakt* pA = new U;
pA->h();
pA->f();
// U::f() U::g() U::f()
```

156 I.4b generative Programmierung

O. Henkel Compilezeit-Polymerphismus

TEMPLATE „POLYMORPHIE“

```
// Compilezeit-Polymerphie
template<class X>
class Abs: public X { // Abs erbt vom Schablonenparameter!
public:
    void h() { Abs<X>::f(); Abs<X>::g(); }

    class Y {
public:
    void f() { cout << "Y::f()"; }
    void g() { cout << "Y::g()"; }
};

    class Z {
public:
    void f() { cout << "Z::f()"; }
    void g() { cout << "Z::g()"; }
};

    Abs<Y> A;
    A.h();
    Y().f();
    // Y::f() Y::g() Y::f()
```

```
template<typename T>
T summe(const list<T>& c) {
    T ergebnis {};
    for (auto wert: c) ergebnis += wert;
    return ergebnis;
}
```

- `summe` kann nicht mit z.B. `vector<int>`-Objekten aufgerufen werden
- Aufruf von `summe` mit einem Container des Typs `list<string>` funktioniert (Konkatenation), ist aber nicht in jedem Fall wünschenswert

Ziel: gewünschte Anforderungen an den Container spezifizieren

Die Bibliothek `type_traits` definiert Strukturdatentypen, mit denen Eigenschaften der STL-Typsablonen abgefragt werden können.

// Einfache Typ-Kategorie Quelle: Breymann, Der C++ Programmierer

```
template <class T> struct is_void;
template <class T> struct is_null_pointer;
template <class T> struct is_integral;
template <class T> struct is_floating_point;
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_lvalue_reference;
template <class T> struct is_rvalue_reference;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;
```

Die Bibliothek `type_traits` definiert Strukturdatentypen, mit denen Eigenschaften der STL-Typsablonen abgefragt werden können.

// zusammengesetzte Typ-Kategorie Quelle: Breymann, Der C++ Programmierer

```
template <class T> struct is_reference;
template <class T> struct is_arithmetic;
template <class T> struct is_fundamental;
template <class T> struct is_object;
template <class T> struct is_scalar;
template <class T> struct is_compound;
template <class T> struct is_member_pointer;
```

Die Bibliothek `type_traits` definiert Strukturdatentypen, mit denen Eigenschaften der STL-Typsablonen abgefragt werden können.

// Typeigenschaften Quelle: Breymann, Der C++ Programmierer

```
template <class T> struct is_const;
template <class T> struct is_literal_type;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct is_final;
template <class T> struct is_signed;
template <class T> struct is_unsigned;
template <class T, class... Args> struct is_constructible;
template <class T> struct is_default_constructible;
template <class T> struct is_copy_constructible;
template <class T> struct is_move_constructible;
template <class T, class U> struct is_assignable;
template <class T> struct is_copyAssignable;
template <class T> struct is_moveAssignable;
template <class T> struct is_destructible;
template <class T> struct has_virtual_destructor;
```



Wie funktioniert das?

Grundlage ist die folgende Typschablone

```
// Bremann, Der C++ Programmierer
template <class T, T v>
struct integral_constant {
    static constexpr T value = v;
    using value_type = T;
    using type = integral_constant<T, v>;
    ...
};
```

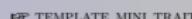
Diese Schablone stellt in ihrem Namensbereich für eine gegebene Instanz die Namen der übergebenen Klasse und den übergebenen Wert bereit.

„Namensgebung: der zweite Template-Parameter `v` muss ein ganzzahliger Typ sein.“

Beispiel:

`integral_constant < int, 5>::value.type` ist gleich `int` und
`integral_constant < int, 5>::value` ist gleich `5`

Da jeder Containertyp `C` der STL mittels `C::value.type` den Datentyp der Elemente bereitstellt, kann man z.B. prüfen, ob der Datentyp der Elemente den geforderten Eigenschaften genügt.



```
1 // eine eigene Mini-Traits-Bibliothek
2 // _integral_constant anstelle von integral_constant
3 template <bool B>
4 using _bool_constant = _integral_constant<bool, B>;
5 using true_t = _bool_constant<true>;
6 using false_t = _bool_constant<false>;
7
8 template<class T, class U>
9 struct _is_same : false_t {};
10 template<class T>
11 struct _is_same<T, T> : true_t {};
12
13 template <typename Container>
14 struct is_int_long
15 : _integral_constant<bool,
16   _is_same<int, typename Container::value_type>::value ||
17   _is_same<long, typename Container::value_type>::value
18 > {};
19
20 int main() {
21     cout << boolalpha;
22     cout << is_int_long<vector<long>>::value << endl; // true
23     cout << is_int_long<vector<char>>::value << endl; // false
24 }
```



`enable_if` stellt einen Typnamen (z.B. für Rückgabewert einer Funktion) bedingt zur Verfügung

std::enable_if

Quelle: en.cppreference.com

```
Defined in header <cassert>
template<bool B, class T = void>
struct enable_if;
```

If `B` is `true`, `std::enable_if` has a public member `typeof` type, equal to `T`; otherwise, there is no member `typeof`. This definition is a convenient way to leverage `std::if` to conditionally remove functions from overload resolution based on type traits and to provide separate function overloads and specializations for different type return types. `std::enable_if` can be used as an additional function argument (not applicable to operator overloads), as a return type (not applicable to constructors and destructors), or as a class template or function template parameter.

Member types

Type Definition
type either `T` or no such member, depending on the value of `B`

Helper types

```
template<bool B, class T = void>
using enable_if_t = typename enable_if<B,T>::type;
```

Possible implementation

```
template<bool B, class T = void>
struct enable_if;
```

```
template<class T>
struct enable_if<true, T> { typedef T type; };
```



```
1 #include <iostream>
2 #include <type_traits>
3 #include <vector>
4 #include <list>
5 #include <string>
6 using namespace std;
7
8 // aus Bremann, Der C++ Programmierer
9 template <typename Container>
10 typename std::enable_if<
11   is_arithmetic<typename Container::value_type>::value ,
12   typename Container::value_type>::type
13 summe(const Container& c) {
14     typename Container::value_type ergebnis {};
15     for (auto wert : c) ergebnis += wert;
16     return ergebnis;
17 }
18
19 int main() {
20     vector<int> v { 4, 2, -1, 3 };
21     cout << summe(v) << endl; // 8
22
23     list<double> l { 4.5, -3.25, 1.25, 2.5 };
24     cout << summe(l) << endl; // 5
25
26     list<string> s { "Pullover", "Hose" };
27     cout << summe(s) << endl;
28 }
29
30 return 0;
31 }
```

No matching function for call to 'summe'

Ziel

Funktion `ausgabe` soll Objekte der Klasse `X` oder `Y` (und ggf. weitere) übergeben bekommen und als `string`-Objekte ausgeben

```
1 class X { // besitzt Methode toString
2     string s;
3 public:
4     X(string _s): s(_s) {}
5     string toString() const { return s; }
6 };
7
8 class Y { // besitzt stattdessen operator string
9     string s;
10 public:
11     Y(string _s): s(_s) {}
12     operator string() const { return s; }
13 };
```

ToDo

Funktionsschablone `void ausgabe(const T&)` erstellen, die zur Compilationszeit überprüft, welche geeignete Methode zur Verfügung steht und für jeden Typ die passende Funktionsinstanz bereitstellt



```
1 template <class T, typename = string>
2 struct hatToString : false_type {};
3 template <class T>
4 struct hatToString<T, decltype(declval<T>().toString())>
5 : true_type {};
6
7 template <class T>
8 void ausgabe(const T& t) {
9     if constexpr(hatToString<T>::value)
10         cout << t.toString() << endl;
11     else
12         cout << static_cast<string>(t) << endl;
13 }
14
15 int main() {
16     X x("Pullover");
17     Y y("Hose");
18
19     cout << boolalpha;
20     cout << hatToString<X>::value << endl; // true
21     cout << hatToString<Y>::value << endl; // false
22
23     ausgabe(x); // Pullover
24     ausgabe(y); // Hose
25 }
```



die kommentierten Anforderungen sind zu implementieren

```
1 typedef list<unsigned int> Container; // OK
2 //typedef list<double> Container; // Compiler warnt :
3 //typedef list<char> Container; //! nicht OK :
4 //typedef list<bool> Container; //! nicht OK :
5 typedef Container::iterator C_Iter;
6 typedef Container::value_type C_Elem;
7
8 // unaeres Praedikat nur sinnvoll fuer numerische Werte!
9 class IsMultipleOf {
10 public:
11     unsigned int divisor;
12     IsMultipleOf(unsigned int n): divisor(n) {}
13     bool operator()(const int& x) const { return !(x%divisor); }
14 };
15
16 Container c {1, 4, 9, 16, 25, 36, 49, 64, 81, 100};
17 C_Iter it;
18
19 cout << "Anzahl der Elemente, die ein Vielfaches von 3 sind:";
20 long n = count_if(c.begin(), c.end(), IsMultipleOf(3));
21 cout << n << endl;
```