

Inhaltsverzeichnis

1	Rechenoperationen	3
2	Maschinensprache	5
3	funktionale Programmierung	5
4	funktionale Programmierung in C++	6
4.1	Overloading der arithmetischen Operationen	7
4.2	Integer-Division in <i>C++</i>	8
4.3	Anwendung	8
4.4	Bedingungen	9
4.5	Rekursion	10
4.6	Von der funktionalen zur prozeduralen Programmierung	10
5	Prozedurale Programmierung	11
5.1	Schleifen	12
5.2	Anwendung: Wurzelberechnung	13
6	Datentypen	15
6.1	Zeichenketten - String	16
6.2	Umgebungsmodell	17
7	Umgebungen	18
7.1	Namensräume	18
7.2	Referenzen	19
8	Container-Datentypen	21
8.1	Datentyp: <i>std::vector</i>	21
9	Iteratoren	25
9.1	Die Funktion <i>std::transform()</i>	26
9.2	Insertion Sort	27
9.3	Insertion Sort	29
10	Templates	31
11	Grundlagen der generischen Programmierung	31
11.1	Funktionen-Templates	32
12	Bestimmung der Effizienz von Algorithmen und Datenstrukturen	34
12.1	technisches Effizienzmaß	35
12.2	\mathcal{O} -Notation/ Ω -Notation	36

13 Zahlendarstellung	38
13.1 natürliche Zahlen \mathbb{N}	38
13.2 ganze Zahlen \mathbb{Z}	40
13.3 reelle Zahlen \mathbb{R}	41
14 Buchstabenzeichen	43
14.1 eigene Datentypen	44
15 Objektorientierte Programmierung	44
15.1 eigene Datentypen mit Kapselung	44
15.2 running example	46
15.3 Member-Funktionen	49
15.4 Vorteile der Kapselung	51
15.5 Operatoren	52
15.6 Objekte nachträglich verändern	53
16 Klasse: Image	55
16.1 Anwendung der Zeiger	62

1 Rechenoperationen

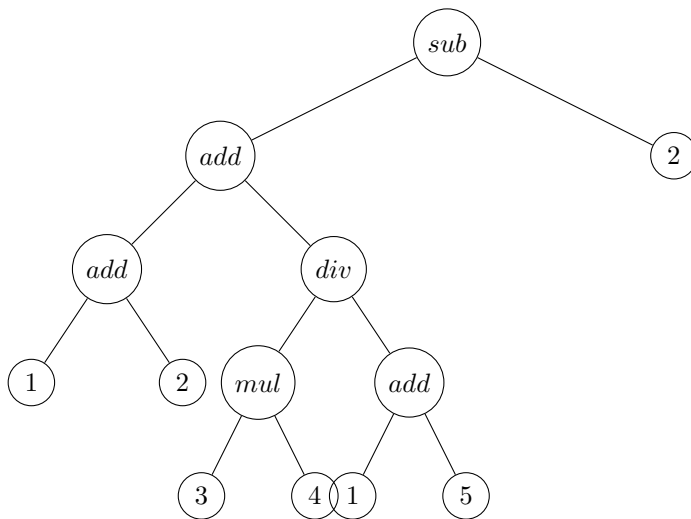
1. Baum besteht aus Knoten (Kreise) und Kanten (Pfeile)
2. Kanten verbinden Knoten mit ihren Kind-Knoten
3. jeder Knoten (außer der Wurzel) hat genau ein Elternteil
4. Knoten ohne Kinder heißen “Blätter“ (leaf-nodes)
5. Teilbaum
 - (a) wähle beliebigen Knoten
 - (b) entferne temporär dessen Eltern-Kante
 - i. der Knoten wird temporär zu einer Wurzel
 - ii. dieser Knoten mit allen seinen Nachkommen bildet wieder seinen Baum - “ Teilbaum des Originalbaums“
 - (c) Tiefe: Abstand des Knotens zur Wurzel
 - (d)

Infix-Notation:

$$1 + 2 + 3 * 4 / (1 + 5) - 2$$

Präfix-Notation:

sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2)



Präfix Notation aus dem Baum rekonstruieren

1. Wenn die Wurzel ein Blatt ist, dann “Drucke die Zahl“
2. sonst (Operator):
 - (a) Drucke Funktionsnamen
 - (b) Drucke “(“
 - (c) wiederhole ab 1) für das linke Kind
 - (d) Drucke “,”
 - (e) wiederhole den Algorithmus ab 1) für das rechte Kind
 - (f) Drucke “)”

Beachte Reihenfolge: Wurzel - Links - Rechts (Pre-Order Traversal) Ergebnis:
 $sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2)$

Definition: Rekursion Rekursion meint Algorithmus für Teilproblem von vorn

Infix Notation

1. wie bei Präfix
2. sonst
 - (a) entfällt
 - (b) wie bei Präfix
 - (c) wie bei Präfix
 - (d) Drucke Operatorsymbol
 - (e) wie bei Präfix
 - (f) wie bei Präfix
 - (g) wie bei Präfix

Beachte Reihenfolge: Links - Wurzel - Rechts (In-Order Traversal)

Ergebnis:
 $((1 + 2) + ((3 * 4) / (1 + 5))) + 2$

Berechne den Wert mit Substitutionsmethode

1. Wenn Wurzel ein Blatt hat, gib die Zahl zurück
2. sonst
 - (a) entfällt
 - (b) entfällt
 - (c) wiederhole ab 1) für linken Teilbaum und speichere Ergebnis als “left-result“

- (d) entfällt
- (e) wiederhole ab 1) für rechten Teilbaum, speichere Ergebnis als “right-result“
- (f) berechne $fkt_name(left - result, right - result)$ und gib Ergebnis zurück

Beachte Reihenfolge: Links - Rechts - Wurzel (Post-Order Traversal)

$$\begin{aligned}
 & sub(add(add(1, 2), div(mul(3, 4), add(1, 5))), 2) \\
 &= sub(add(add(1, 2), div(12, 6)), 2) \\
 &= sub(add(3, 2), 2) \\
 &= sub(5, 2) \\
 &= 3
 \end{aligned}$$

2 Maschinensprache

- optimiert für die Hardware (viele verschiedene)
- Gegensatz: höhere Programmiersprache ($C++$) ist optimiert für Programmierer
- Compiler oder Interpreter übersetzen Hoch- in Maschinensprache

Vorgang des Übersetzens

1. Eingaben (und Zwischenergebnisse) werden in Speicherzellen abgelegt \Rightarrow jeder Knoten im Baum bekommt eine Speicherzelle (Maschinensprache: durchnummeriert ; Hochsprache: sprechende Namen)
2. Speicherzellen für die Eingaben *initialisieren* ; Notation: $SpZ \leftarrow Wert$
3. Rechenoperationen in der Reihenfolge des Substitutionsmodells ausführen und in der jeweiligen Speicherzelle speichern ; Notation: $SpZ_Ergebnis \leftarrow fkt_name\ SpZ_Arg1\ SpZ_Arg2$
4. alles in Zahlencode umwandeln
 - Funktionsname \Rightarrow Opcodes
 - Speicherzellen: nur die Nummer
 - Werte sind schon Zahlen
 - Notation: Opcode Ziel SpZ SpZ_Arg1 SpZ_Arg2 oder Opcode Ziel SpZ Initialwert

3 funktionale Programmierung

(alles durch Funktionsaufrufe ausführen)

1. bei Maschinensprache wurden Zwischenergebnisse in Speicherzellen abgelegt

2. das ist auch in der funktionalen Programm. eine gute Idee

(a) Speicherzellen werden durch Namen (vom Programmierer vergeben) unterschieden

(b) Beispiel: Lösen einer quadratischen Gleichung: $ax^2+bx+c=0$, finde $x_{1/2} \Rightarrow x^2-px+q=$

$$0 \quad \text{mit} \quad p = -\frac{b}{2a}, q = \frac{c}{a} \Rightarrow x_1 = -\frac{b}{2a} + \sqrt{\left(-\frac{b}{2a}\right)^2 - \frac{c}{a}}$$
$$\Leftarrow \text{allgemein} : x_{1/2} = p \pm \sqrt{p^2 - q}$$

(c) Präfix:

$x_1 \leftarrow \text{add}(\text{div}(\text{div}(b, a), -2), \text{sqr}(\text{sub}(\text{mul}(\text{div}(\text{div}(b, a), -2), \text{div}(\text{div}(b, a), -2)), \text{div}(c, a))))$

mit Zwischenergebnissen und Infix-Notation: $p \leftarrow b/c/-2$ oder $p \leftarrow -0,5*b/a$ $q \leftarrow c/a$

$\text{discriminant} \leftarrow \text{sqr}(p * P - q)$

$x_{1/2} \leftarrow p \pm \text{discriminant}$

3. zwei Vorteile:

(a) lesbar

(b) redundante Berechnung verschieden

Beachte: In der funktionalen Programmierung können die Speicherzellen nach der Initialisierung nicht mehr verändert werden

(c) Speicherzellen mit Namen sind nützlich, um Argumente an Funktionen zu übergeben \Rightarrow Definition eigener Funktionen

Bsp: `function sq(x){ return x*x }`

4 funktionale Programmierung in C++

1. in C++ hat jede Speicherzelle einen Typ (legt Größe und Bedeutung der Speicherzelle fest)
wichtigste Typen: "int" für ganze Zahlen, "double" für reelle Zahlen, "std::string" für Text
zugehörige Literale (Konstanten): 12, -3 (int) -1.02 , 1.2e-4 (double) "text text" (string)

2. Die Initialisierung wird geschrieben als

```
type_name spz_name = initialwert
```

Bsp:

```
double a = 10
std::cout << "x_1" << x_1 << "\n" ;
```

3. eigene Funktion in C++:

```
type_ergebnis funktionsname (typ_arg1 name_arg1, typ_arg2 name_arg2)
{
    <code>
    return ergebnis;
}
```

4. zwei Funktionen mit gleichem Namen, aber unterschiedlichen Typen dürfen in C++ gleichzeitig definiert sein (“overloading”)
 - ⇒ C++ wählt automatisch die richtige Variante anhand des Argumenttypes (“overload resolution“)
5. jedes C++-Programm muss genau eine Funktion names “main“ haben: Dort beginnt die Programm-Ausführung

Bsp:

```
int main() { <code>    return 0 (erfolgreich abgearbeitet) }
```
6. Regel von C++ für erlaubte Namen (Speicherzelle & Funktion):
 - (a) erste Zeichen: Klein- oder Großbuchstaben des englischen Alphabets oder _
 - (b) optional: weitere Zeichen: wie erstes Zeichen oder Ziffern 0 ... 9
7. vordefinierte Funktionen in C++
 - (a) eingebaute Funktionen (immer vorhanden) z.B. Infix Operatoren
 - (b) Funktionen der Standardbibliothek (Programmierer muss sie explizit auffordern)
 - i. z.B. algebraische Funktionen beginnend mit std::...
 - ii. sind in Module geordnet, z.B. `cmath` $\hat{=}$ algebraische Funktionen, `iostream` $\hat{=}$ Ausgabe, z.B. `std::cout`
 - iii. Um ein Modul zu benutzen, muss man zuerst (am Anfang des Programms) sein Inhaltsverzeichnis importieren


```
#include <module_name>
```

 sprich “Header inkludieren“

```
# include <iostream>
# include <string>

int main() {

std::cout <<  "Hello" <<  "\n";
std::string >>  ausgabe = "mein erstes Programm"
std::cout <<  ausgabe;

return 0
}
```

4.1 Overloading der arithmetischen Operationen

```
int a = 3;
int b = 4;
int c = a * b;
double x = 3.0;
double y = 4.0;
double z = x * y;
```

$3.0 * 4 \Rightarrow$ automatische Umwandlung in höheren Typ, hier: "double" \Rightarrow wird als $3.0 * 4.0$ ausgeführt

4.2 Integer-Division in C++

Konsequenzen:

1. Division unterscheidet sich nach dem Datentypen: $(-12)/5 \Rightarrow -2 \neq -2.4 \Leftarrow (-12.0/5.0)$
2. negative Ereignisse werden aufgerundet, positive abgerundet (truncating division)
d.h. Nachkommastellen abschneiden, d.h. Richtung Null runden
3. Gegensatz (z.B. zu Python): floor division $\hat{=}$ wird immer abgerundet
4. Divisionsrest:

```
int a = ...;
int b = ...;
int q = a/b;
(a/b)*b = q * b
```

ist im allgemeinen ungleich $a \Rightarrow$

```
int rest = a - q*b;
```

1. wenn Division aufgeht \Rightarrow rest = 0 , sonst $\neq 0$
2. Invariante:

```
(a/b) * b + rest = a

int rest1 = a % b; // equivalent: a - (b/a)*b
```

4.3 Anwendung

Wochentag für beliebiges Datum bestimmen: gegeben: d, m, y , gesucht: $w \in \{0, \dots, b\}$

`int weekday(int d, int w, int y) ; weekday(10,11,2016) \Rightarrow 3 (Donnerstag)`

Teilprobleme

1. finde den Wochentag vom 1. Januar y
2. finde den Abstand vom (d,m,y) zum (1,1,y)
3. setze beides zusammen

Schaltjahresregel: y ist Schaltjahr, wenn:

1. y durch 4 teilbar, aber nicht durch 100 \Rightarrow 2004, 2006, nicht 2100

2. y durch 400 teilbar $\Rightarrow 2000$

\Rightarrow 400-Jahres-Zyklus der Regeln: nach 400 Jahren beginnt die Schaltjahresregel von vorn

- Beobachtung: der 1.1.2001 ist der erste Tag eines neuen Zyklus und war Montag
- die Anzahl der Tage vom 1.1. y zum 1.1.2001 ist:
 $z = y - 2001 \quad \triangle = 365 * z + z/4 - z/100 + z/400$
- floor division ist wichtig, wenn $z < 0$, z.B. $y = 2000, z = -1$

zu②: d.m. ist der x -te Tag im Jahr mit:

- kein Schaltjahr

1. $m = 1 \Rightarrow d$
2. $m = 2 \Rightarrow d + 31$
3. $m = 3 \Rightarrow d + 59$
4. $m = 4 \Rightarrow d + 90$
5. $m = 5 \Rightarrow d + 120$
6. $m > 2 \Rightarrow d + 59 + (153 * m - 457)/5$

- Schaltjahr

1. $m = 1 \Rightarrow d$
2. $m = 2 \Rightarrow d + 31$
3. $m = 3 \Rightarrow d + 60$
4. $m = 4 \Rightarrow d + 91$
5. $m = 5 \Rightarrow d + 121$
6. $m > 2 \Rightarrow d + 60 + (153 * m - 457)/5$

zu③: Wochentag von d, m, y :

```
w = (w_11y + x - 1) mod 7
```

4.4 Bedingungen

- Bei den meisten Algorithmen ist die Reihenfolge der Schritte nicht fix, sondern hängt von den Eingabedaten ab
- Beispiel: Auswahl der Offset $d \rightarrow x$ hängt von m ab
dafür die Funktion:

```
cond ( bedingung , resultat_wenn_wahr , resultat_wenn_falsch )
```

- kanonische Beispiele: Absolutbetrag, Vorzeichenfunktion

Bedingungen programmieren:

- relationale Operatoren: Vergleich von zwei Argumenten
 $<, >, \leq, \geq, !=$
- logische Operatoren: Verknüpfen von mehreren Bedingungen
 $\&\&(und), ||(oder), !=(nicht)$
- in $C++$ gibt es keine Prefix-Variante für die $cond()$ -Funktion, aber eine Infix-Variante:

```
(bedingung) ? erg_wenn_wahr : erg_wenn_falsch

int abs (int x) {
    return (x >= 0) ? x : -x;
}
double abs (double x) {
    return (x >= 0.0) ? x : -x;
}
int sign (int x) {
    return (x == 0) ? 0 : ((x > 0) ? 1 : -1);
}
```

4.5 Rekursion

bedeutet: eine Funktion ruft sich selbst auf (evtl. indirekt)

- kanonisches Beispiel: Fakultätsfunktion $k! = 1 \cdot 2 \cdot \dots \cdot (k-1) \cdot k$
- in $C++$ (rekursive Definition)

```
int fakultaet (int k) {
    return (k == 0) ? 1 : k * fakultaet(k-1) ;
}
```

- wichtige Eigenschaften:
 - jede rekursive Funktion muss mindestens einen nicht-rekursiven Zweig enthalten, der nach endlich vielen rekursiven Aufrufen erreicht wird “Rekursionsabschluss“- sonst: Endlosrekursion (Absturz)
 - bei jedem Aufruf werden dem Namen der Dateenelemente (Argumente & Zwischenergebnisse) neue Speicherzellen zugeordnet
 $fakultaet(3) \rightarrow fakultaet(2) \rightarrow fakultaet(1) \rightarrow fakultaet(0) \Rightarrow$
 $return\ 3*fakultaet(2) \leftarrow return\ 2*fakultaet(1) \leftarrow return\ 1*fakultaet(0) \leftarrow return\ 1$

4.6 Von der funktionalen zur prozeduralen Programmierung

- Eigenschaften der FP:
 - alle Berechnungen durch Funktionsaufrufe, Ergebnis ist Rückgabe

- Ergebnis hängt nur von den Werten der Funktions-Argumente ab, nicht von externen Faktoren (*referentielle Integrität*)
- Speicherzellen für Zwischenergebnisse/Argumente können nach der Initialisierung nicht geändert werden (*write once*)
- Möglichkeit der rekursiven Funktionsaufrufe (jeder Aufruf bekommt eigene Speicherzellen)
- Vorteile:
 - natürliche Ausdrucksweise für arithmetische und algebraische Funktionalität (*Taschenrechner*)
 - einfache Auswertung durch Substitutionsmodell - Auswertungsreihenfolge nach Post-Order
 - mathematisch gut formalisierbar \Rightarrow Korrektheitsbeweise (besonders bei Parallelverarbeitung)
 - Rekursion ist mächtig und natürlich für bestimmte Probleme (z.B. Fakultät)
- Nachteile:
 - viele Probleme lassen sich anders natürlicher ausdrücken (z.B. Rekursion vs. Iteration)
 - setzt unendlich viel Speicher voraus (\Rightarrow Memory management notwendig \Rightarrow später)
 - Entitäten, die sich zeitlich verändern schwer modellierbar, teilweise unnatürlich
- Korrolar: Man kann keine externen Ressourcen (z.B. die Console/Drucker, Bildschirm) ansprechen (weil zeitlich veränderlich)
“keine Seiteneffekte“
- Lösung: Einführung einer Multi-Paradigmen-sprachen, z.B. Kombination von funktionaler mit prozeduraler Programmierung

5 Prozedurale Programmierung

- Kennzeichen:
 - Prozeduren - Funktionen, die nichts zurückgeben, haben nur Seiteneffekte
Bsp: auf Konsole ausgeben
- ```
std::cout << "Hello World \n"; // Infix
operator << (std::cout, "Hello \nLeftarrow"); // Praefix notation
```
- Prozeduren in C++:
    1. Funktion, die *void* zurückgibt (Pseudotyp nur “nichts“)
    2. Returnwert ignorieren
  - Anweisen zur Steuerung des Programmablaufs (z.B. *if* / *else*)

```

// funktional:
int abs (int x) {
 return (x>=0) ? x : -x ;
}

// prozedural
int abs (int x) {
 if (x >= 0) {
 return x;
 } else {
 return -x;
 }
}

```

- Zuweisung:

- Speicherzellen können nachträglich verändert werden “*read-work*“

```

// prozedural
int foo (int x) {
 int y =2;
 int z1 = x * y; // z1 = 6
 y = 5;
 int z2 = x * y; // z2 = 15
 return z1 + z2;
}

// write once
typ const name = wert

// funktional
int foo (int x) {
 int y = 2;
 int z1 = x * y; // z1 = 6
 int y2 = 5;
 int z2 = x * y2; // z2 = 15
 return z1 + z2;
}

```

- ⇒ Folgen:

- mächtiger, aber ermöglicht völlig neue Bugs ⇒ Erhöhte Aufmerksamkeit beim Programmieren
- die Reihenfolge der Ausführung ist viel kritischer als beim Substitutionsmodell
- der Programmierer muss immer ein mentales Bild des aktuellen Systemzustands haben

## 5.1 Schleifen

der gleiche Code soll oft wiederholt werden

```

while (bedingung) {
 ... // code wird ausgefuehrt, solange bedingung "true" ist
}

```

Bsp: Zahlen von 0-2 ausgeben)

```
int counter = 0;
while (counter < 3) {
 std::cout << counter << "\n";
 counter = counter + 1;
}
```

| counter | Bedingung | Ausgabe |
|---------|-----------|---------|
| 0       | true      | 0       |
| 1       | true      | 1       |
| 2       | true      | 2       |
| 3       | false     | ∅       |

- $C++$  beginnt mit der Zählung meist bei 0 “zero-based“
- vergisst man Inkrementierung  $\text{counter} = \text{counter} + 1 \Rightarrow$  Bedingung immer true  $\Rightarrow$  Endlosschleife  $\Rightarrow$  Bug
- drei äquivalente Schreibweisen für Implementierung:

```
counter = counter + 1; // assignment
counter += 1; // add-assignment
++ counter; // pre-increment
```

## 5.2 Anwendung: Wurzelberechnung

Ziel: `double sqrt (double y)` Methode: iterative Verbesserung mittels Newtonverfahren

```
initial guess x(0) bei t=0 geraten
while not_good_enough(x(t)) {
 update x(t+1) from x(t)
 t = t+1
}
```

Newtonverfahren: finde Nullstelle einer gegebenen Funktion  $f(x)$ , d.h. suche  $x^*$ , sodass  $f(x^*) = 0$  oder  $|f(x^*)| < \epsilon$

1. Taylorreihe von  $f(x)$ :  $f(x + \Delta) \approx f(x) + f'(x) \cdot \Delta + \dots$
2.  $0 = f(x^*) \approx f(x) + f'(x) \cdot \Delta = 0 \Rightarrow \Delta = -\frac{f(x)}{f'(x)}$
3. Iterationsvorschrift:  $x^{(t+1)} = x^{(t)} - \frac{f(x^{(t)})}{f'(x^{(t)})}$
4. Anwendung auf Wurzel: setze  $f(x) = x^2 - y \Rightarrow \text{mit } f(x^*) = 0 \text{ gilt } (x^*)^2 - y = 0$
5. Iterationsvorschrift:  $x^{(t+1)} = x^{(t)} - \frac{(x^{(t)})^2 - y}{2x^{(t)}} = \frac{(x^{(t)})^2 + y}{2x^{(t)}}$   
 $x^{(t+1)} = \frac{x^{(t)} + \frac{y}{x^{(t)}}}{2}$  mit  $x^* = \sqrt{y} \Rightarrow x^{(t+1)} = \sqrt{y}$

```
double sqrt (double y) {
 if (y<0.0) {
 std::cout << "Wurzel aus negativer Zahl \n";
 return -1.0;
 }
 if (y == 0.0) {
 return 0.0;
 }
 double x = y; // initial guess
 double epsilon = 1e-15 * y; // double Genauigkeit

 while (abs(x*x-y) > epsilon) {
 x = (x + y/x) / 2.0 ;
 }
 return x;
}
```

**for - Schleife** Zum Vergleich mit der while-Schleife:

```
int c = 0;
while (c < 3) {
 ... // unser code
 c += 1; //sonst funktionsunfaehig
}
```

die *for* - Schleife ist dagegen "idiotensicher"

```
for (int c =0; // Initialisierung
 c < 3; // Bedingung (oder: c!=3)
 c+=1) { // Incrementierungsanweisung
 ... // unser code
}
```

- Befehle, um Schleifen vorzeitig abubrechen:

- *continue* (bricht aktuelle Iteration ab und springt zum Schleifenkopf)
- *break* (bricht die ganze Schleife ab und springt hinter die schließende Klammer)
- *return* (beendet die Funktion und damit auch die Schleife)

- 3 gleichbedeutende Beispiele:

```
for (int c =0; c<10; ++c) {
 if (c%2 ==0) { // gerade Zahl?
 std::cout << c << "\n";
 }
}

/* Sobald in der if-Anweisung nur eine Zeile steht, kann sie weggelassen
werden. Das ist gefaehrlich und die Klammern sollten eher trotzdem
gesetzt werden */

for (int c =0; c<10; ++c) {
 if (c %2 !=0) { // nicht gerade?
 continue;
 }
}
```

```

 }
 std::cout << c << "\n" ;
}

for (int c =0; c<10; c+=2) {
 std::cout << c << "\n" ;
}

```

- mit den wichtigsten Schleifen ist bereits ein guter Grundstein für die vielseitige Programmierung gelegt

## 6 Datentypen

- Basistypen:  
Bestandteil der Sprachsyntax und normalerweise direkt von der Hardware(CPU) unterstützt
  - int (ganze Zahlen)
  - double (Fließkommazahlen)
  - bool (*true* oder *false*)
  - später mehr
- zusammengesetzte Typen:  
mithilfe von *struct* oder *class* aus einfacheren Typen zusammengebaut
  - Standardtypen: in der C++ Standardbibliothek definiert (*#include ..*)
  - Bsp: `std::string` mit *#include <string>*
  - externe Typen: aus anderer Bibliothek, die man zuvor herunterladen und installieren muss
  - eigene Typen: vom Programmierer selbst implementiert
- durch “objekt-orientierte Programmierung“ erreicht man, dass zusammengesetzte Typen genauso einfach, bequem und effizient sind, wie Basistypen
- “Kapselung“: die interne Struktur und Implementation ist für den Benutzer unsichtbar
- Benutzer manipuliert Speicher über Funktionen (“member functions“)  $\approx$  Schnittstelle des Typs Interface

```

zusammenges_typ_name var_name = initial-wert; // init
var_name.foo(a1, a2); // oder: foo(var_name, a1, a2)

```

## 6.1 Zeichenketten - String

- zwei Datentypen in C++
- klassischer C-String: `char[]` ("character array")
- C++-String: `std::string` - gekapselt und bequem
- String-Literale: "Zeichenkette"
- einzelnes Zeichen: 'z'  
Vorsicht: die String-Literale sind C-Strings (gibt keine C++ String-Literale)
- Initialisierung:

```
std::string s1 = "abcde"; // Zuweisung
std::string s2 = s1;
std::string leer = "";
s1.size() // Laenge (Anzahl der Zeichen)
s1.empty() // Test: s1.size() == 0
```

- Addition: Strings aneinanderreihen ("concatenate")

```
std::string s3 = s + "i,k"; // "xyi,k"
std::string s3 = s + s; // "xyxy"
std::string s3 = "abc" + "def"; // Bug - Literale unterstützen + mit
// ganz anderer Bedeutung
```

- Add-Assignment: Abkürzung für Addition gefolgt von Zuweisung

```
s += "nmk"; // ist gleich zu:
s = s + "nmk"; // "xynmk"
s3 = (s + "abc") + "def"; // ok
```

- die Zeichen werden intern in einem C-Array gespeichert  
Array: zusammenhängende Folge von Speicherzellen des gleichen Types, hier: `char`

|   |   |   |   |   |
|---|---|---|---|---|
| a | b | c | d | e |
|---|---|---|---|---|

 Länge: 5;  $s[\text{index}] \in \{0, 1, 2, 3, 4\}$ 

```
std::string s = "abcde";
for (int k = 0; k < s.size(); ++k) {
 std::cout << s[k] << "\n";
}
```

Variante①: 'in-place' (den alten String überschreiben, selbe Speicherzelle)

```
int i = 0;
int k = s.size() - 1;
while (i < k) {
 char tmp = s[i] // i-tes Zeichen merken
 s[i] = s[k];
 s[k] = tmp;
 --k; // k = k - 1
 ++i;
}
```



Variante②: neuen String erzeugen

```
std::string s = "abcde";
std::string r = "";
for (int k = s.size() - 1; k >= 0; --k)
```

## 6.2 Umgebungsmodell

- in prozeduraler Programmierung: Gegenstück zum Substitutionsmodell für funktionale Programmierung
- Zwecke:
  - Regeln für Auswertung von Ausdrücken
  - Regeln für automatische Speicherverwaltung: Freigeben nicht mehr benötigter Speicherzellen (nützlich bei in der Praxis immer endlichem Speicher)  
⇒ bessere Approximation von “unendlich viel Speicher“

- Umgebung beginnt normalerweise bei “{“ und endet bei “}“  
Ausnahme: for-Schleife, Funktionsdefinitionen, globale Umgebung

```
for (int k=0; k<10; ++k) { // Laufvariable Teil der Umgebung
 ... // code
}

bool is_email (std::string s) { // Speicherzellen fuer Argumente
 // und Ergebnis gehoeren zur Umgebung
 ... // code
}
```

- automatische Speicherverwaltung:
  - Speicherzellen, die in einer Umgebung angelegt werden, werden am Ende der Umgebung in umgekehrter Reihenfolge freigegeben
  - Compiler fügt vor “{“ automatisch die notwendigen Befehle ein
  - Speicherzellen in der globalen Umgebung werden dem Programmierenden freigegeben

```
int global = 1;

int main() {
 int l = 2;
 {
 int m = 3;
 } // m wird freigegeben
} // l wird freigegeben
// global wird freigegeben
```

- Umgebungen können beliebig tief geschachtelt werden  
⇒ alle Umgebungen bilden einen Baum, mit der globalen Umgebung als Wurzel

- Funktionen sind in der globalen Umgebung definiert  
 $\Rightarrow$  Umgebung jeder Funktion sind “Kinder“ der globalen Umgebung (Ausnahme: Namensräume)  
 $\Rightarrow$  Funktionsumgebung ist nicht Kind der Umgebung, in der sie aufgerufen wird
- Jede Umgebung besitzt eine Zuordnungstabelle für alle Speicherzellen, die in der Umgebung definiert werden
 

| Name | Typ | aktueller Wert |
|------|-----|----------------|
| 1    | int | 2              |
- jeder Name kann pro Umgebung nur  $1 \times$  vorkommen (gleichzeitig in anderen Umgebungen)  
 Ausnahme: Funktionsnamen können mehrmals vorkommen bei “function overloading“ (C++)
- Alle Befehle werden relativ zur aktuellen Umgebung ausgeführt  
 aktuell: Zuordnungstabelle der gleichen Umgebung & aktueller Wert zum Zeitpunkt des Aufrufs (Zeitpunkt wichtig im Substitutionsmodell)

Beispiel:  $c = a * B$  ;

Regeln:

- wird der Name (a,b,c) in der aktuellen Zuordnungstabelle gefunden:
    - ① Typisierung  $\Rightarrow$  Fehlermeldung, wenn Typ und Operation zusammenpassen
    - ② andernfalls, setze aktuellen Wert aus Tabelle in Ausdruck ein
  - wird der Name nicht gefunden, suche in der Elternumgebung weiter mit ① oder ②
  - wird der Name bis zur Wurzel nicht gefunden  $\Rightarrow$  Fehlermeldung
  - ist der Name in mehreren Umgebungen vorhanden, gilt das zuerst gefundene (Typ, Wert)
- $\Rightarrow$  Programmierer muss selbst darauf achten, dass:
1. bei der Suche die gewünschte Speicherzelle gefunden wird  $\Rightarrow$  benutze “sprechende“ Namen
  2. der aktuelle Wert der richtige ist  $\Rightarrow$  beachte Reihenfolge der Befehle!

## 7 Umgebungen

### 7.1 Namensräume

spezielle Umgebungen in der globalen Umgebung (auch geschachtelt) mit einem Namen

- Ziele:
  - Gruppieren von Funktionalität in Module (zusätzlich zu Headern)
  - Verhindern von Namenskollisionen
  - Beispiel: C++ Standardbibliothek

```
namespace std {
 double sqrt (double x);
 namespace chrono {
 class system_clock;
 }
}
```

⇒ `std::sqrt(x)` wird zu `sqrt(x)`

Besonderheit: mehrere Blöcke mit selbem Namensraum werden verschmolzen

- Funktionen befinden sich in der globalen Umgebung
- ⇒ Umgebung der Funktion ist Kind der globalen Umgebung

```
int p = 2;
int q = 3;

int foo (int p) { // lokales p verdeckt das globale, aber globales q
 sichtbar
 return p * q;
}

int main() {
 int k = p * q; // beides ist global; =6
 int p = 4; // lokales p, was das globale verdeckt
 int r = p * q; // lokales p. globales q; =12
 int s = foo(p); // lokales p wird zum lokalen p von foo(); =12
 int t = foo(q); // globales q wird zum lokalen p von foo(); =9
}
```

Beispiel: `my_sin` (Übung 3.3)

```
double taylor_sin (double x) {
 return x - std::pow(x,3)/6.0;
}

double pump_sin (double sin_third) {
 return 3.0*sin_third - 4.0 * std::pow(sin_third,3)
}

double pi_2 = 2.0*M_PI;

double normalize (double x) {
 double k = std::floor(x/pi_2); // wie vielte Periode
 double y = x-pi_2*k; // 0 <= y < pi_2
 return (y <= M_PI) ? y : y-pi_2; // -pi < result <= pi
}

double my_sin (double x) {
 double y = normalize(x) ;
 return (std::abs(y)<=0.15) ? taylor_sin(y) : pump_sin(y/3.0);
}

int main() {
 double r = my_sin(0.78) ;
}
```

**global**  
 $\pi_2 = 6.28$

**main**  
 $r = 2$

## 7.2 Referenzen

- sind neue (zusätzliche) Namen für vorhandene Speicherzellen

```

int x = 3; // neue Variable x mit neuer Speicherzelle
int & y = x; // Referenz: y ist neuer Name fuer x, beide haben dieselbe
 Speicherzelle

y = 4; // Zuweisung an y, aber x aendert sich auch, d.h. x == 4
x = 5; // jetzt y == 5

int const & z = x; // read-only Referenz, d.h. z = 6 ist verboten
x = 6; // jetzt auch z == 6

```

- Hauptanwendung:

- Umgebung, wo eine Funktion aufgerufen wird und die Umgebung der Implementation sind unabhängig, d.h. Variablen der einen Umgebung sind in der anderen nicht sichtbar

Beispiel:

```

int foo (int x) { // pass-by-value (Uebergabe des echten Werts)
 x += 3;
 return x;
}

int bar (int & x) { // pass-by-reference (Uebergabe der Adresse der
 // Speicherzelle)
 x += 3;
 return x;
}

void baz (int & z) { // pass-by-reference
 z += 3; // kein return Wert
}

int main() {
 int a = 2;
 std::cout << foo(a) << "\n"; // Ausgabe: 5
 std::cout << a << "\n"; // Ausgabe 2

 std::cout << bar(a) << "\n"; // Ausgabe: 5
 std::cout << a << "\n"; // Ausgabe: 5

 baz(a);
 std::cout << a << "\n";
}

```

- Funktionen die Werte nur über eine Referenz ändern heißen Seiteneffekt der Funktion (Haupteffekt ist immer der return Wert) [in der funktionalen Programmierung sind Seiteneffekte verboten mit Ausnahme von Ein-/Ausgabe]

- Ziele

1. häufig möchte man Speicherzellen in beiden Umgebungen teilen  $\Rightarrow$  verwende Referenzen
2. häufig will man vermeiden, dass eine Variable kopiert wird (pass-by-value)  
 $\Rightarrow$  durch *pass-by-value* braucht man keine Kopie  $\Rightarrow$  typisch *const &*  $\cong$  read-only, keine Seiteneffekte

```
void print_string(std::string const & s) {
 std::cout << s;
}
```

## 8 Container-Datentypen

dienen dazu, andere Datentypen aufzubewahren

- Art der Elemente
  - homogene Container: alle Elemente haben den gleichen Typ (typisch für  $C++$ )
  - heterogene Container: Elemente können verschiedene Typen haben (z.B. Python)
- Art der Größe
  - statische Container: feste Größe, zur Compilezeit bekannt
  - dynamische Container: Größe zur Laufzeit veränderbar
- Arrays sind die wichtigsten Container, weil effizient auf Hardware abgebildet und einfach zu benutzen
  - klassisch: Arrays sind statisch, z.B. C-Arrays (hat  $C++$  geerbt)
  - modern: dynamische Arrays:
    - \* Entdeckung einer effizienten Implementierung
    - \* Kapselung durch Objekt-Orientierte-Programmierung (sonst zu kompliziert)
- ein dynamisches Array:  $std::string$  ist Abbildung  $int \mapsto char$   $Index \rightarrow Zeichen$
- wir wollen das selbe Verhalten für beliebige Elementtypen:  $std::vector$

### 8.1 Datentyp: $std::vector$

```
#include <vector>

std::vector<double> v(20, 0.0); // initialisiert mit Groesse,
 Initialwert

// analog: std::vector<int>, std::vector<std::string>
```

- Abbildung:  $int \mapsto double$
- weitere Verallgemeinerung: Indextyp beliebig (man sagt dann “Schlüssel-Typ§“ typische Fallen:
  - Index ist nicht im Bereich  $0 \leq Index < size$ , z.B. Matrikelnummer
  - Index ist String, z.B. Name eines Studenten

- `std::map`, `std::unordered_map` (Binärer Suchbaum)

Beispiel:

```
std::map<int, double> noten; // noten[3 1 2 4 5 2 3 1 3] = 1.0
std::map<string, double> noten; // noten["krause"] = 1.0
```

dabei: <Schlüsseltyp, Elementtyp>

- Erzeugen:

```
std::vector<double> v(20, 1.0);
std::vector<double> v; // leeres Array (erst ab C++ 11)

std::vector<double> v = {1.0, -3.0, 2.2}; // "initializer list"
```

- Größe:

```
v.size()
v.empty() (=v.size() ==0)
```

- Größe ändern:

```
v.resize(neue_groesse, initialwert)
1.: neue_gr < size() // Elemente ab neue_gr gelöscht, andere bleiben
2.: neue_gr > size() // neue Elemente mit Initialwert am Ende
 angehängt

v.push_back(neues_element) // ein neues Element am Ende anhängen

v.insert(v.begin()+index, neues_element); // neues Element an Position
 // index einfügen folgende Werte werden
 // um eine Position verschoben
 v.begin() ist Iterator

v.pop_back() // letztes Element löschen (effizient)

v.erase(v.begin()+index) // Element aus Position löschen,
 // hintere Werte verschieben

v.clear() // alles löschen
```

- Zugriff:

```
v[k] // Element bei Index k
v.front() // erstes Element
v.back() // letztes Element

v.at(k) // wie v[k], aber Fehlermeldung, wenn nicht 0 <= k < size()
```

- Funktionen für Container: benutzen in C++ Iteration, damit sie für verschiedenste Container funktionieren
- Iteration-Range:

```

v.begin()
v.end() // hinter dem letzten Element

im Header <algorithm>

```

- alle Elemente kopieren:

```

std::vector<double> source = {1.0, 2.0, 3.0, 4.0, 5.0};
std::vector<double> target(source.size(), 0.0);

std::copy(source.begin(), source.end(), target.begin());
std::copy(source.begin()+2, source.end(), target.begin());
// unbenutzte Initialwerte bleiben erhalten

```

- Elemente sortieren:

```

std::sort(v.begin(), v.end()); // "in-place"
std::random_shuffle(v.begin(), v.end()) // "in-place"

```

### Warum ist `push_back()` effizient?

- veraltete Lehrmeinung: Arrays sind nur effizient, wenn statisch (d.h. Größe zur Compilezeit, spätestens bei Initialisierung bekannt)
- modern: bei vielen Anwendungen genügt, wenn Array (meist) nur am Ende vergrößert wird (z.B. `push_back`)  
dies kann sehr effizient unterstützt werden  $\Rightarrow$  dynamisches Array
- `std::vector` verwaltet intern ein statisches Array der Größe "`v.capacity() >= v.size()`"
  - wird das interne Array zu klein  $\Rightarrow$  wird automatisch auf ein doppelt so großes umgeschaltet
  - ist das interne Array zu groß, bleiben unbenutzte Speicherzellen als Reserve
- Verhalten bei `push_back()`
  - noch Reserve vorhanden: lege neues Element in eine unbenutzte Speicherzelle  $\Rightarrow$  billig & chillig
  - keine Reserve:
    1. alloziere neues statisches Array mit doppelter Kapazität
    2. kopiere die Daten aus allem ins neue Array
    3. gebe das alte Array frei
    4. gehe zu ①, jetzt wieder Reserve vorhanden  
Umkopieren ist nicht teuer, da es nur selten nötig ist
  - Beispiel:

```

std::vector<int> v;
for (int k = 0; k < 32; ++k) {
 v.push_back(k);
}

```

| k        | <i>cap_vor_p_b()</i> | <i>cap_nach_p_b()</i> | <i>size()</i> | Reserve | Umkopierung |
|----------|----------------------|-----------------------|---------------|---------|-------------|
| 0        | 0                    | 1                     | 1             | 0       | 0           |
| 1        | 1                    | 2                     | 2             | 0       | 1           |
| 2        | 2                    | 4                     | 3             | 1       | 2           |
| 3        | 4                    | 4                     | 4             | 0       | 0           |
| 4        | 4                    | 8                     | 5             | 3       | 4           |
| 5 ... 7  | 8                    | 8                     | 8             | 0       | 0           |
| 8        | 8                    | 16                    | 9             | 7       | 8           |
| 9 ... 15 | 16                   | 16                    | 16            | 0       | 0           |
| 16       | 16                   | 32                    | 17            | 15      | 16          |

...

- Kosten:
  - 32 Elemente einfügen = 32 Kopien extern  $\Rightarrow$  intern
  - aus altem Array ins neue kopieren = 31 Kopien intern  $\Rightarrow$  intern  
 $\Rightarrow$  im Durchschnitt sind pro Einführung 2 Kopien nötig  
 $\Rightarrow$  dynamisches Array ist doppelt so teuer, wie das statische  
 $\Rightarrow$  immer noch sehr effizient
- relevante Funktionen von *std::vector*
  - *v.size()*: aktuelle Zahl der Elemente
  - *v.capacity() - v.size()*: Reserve ( $\geq 0$ )
  - *v.resize(new\_size)*: ändert immer *v.size()*, aber *v.capacity()* nur wenn  $< new\_size$
  - *v.reserve(new\_capacity)*: ändert *v.size()* nicht, aber *v.capacity()* falls *new\_capacity*  $\geq size$
  - *v.shrink\_to\_fit()*: *v.reserve(v.size())* (Reserve ist danach 0), wenn Endgröße erreicht
- wenn Reserve  $> size$ : *capacity* kann auch halbiert werden

## wichtige Container der C++ Standardbibliothek

- dynamisches Arrays: *std::string*, *std::vector*
- assoziative Arrays: *std::map*, *std::unordered\_map*
- Mengen: *std::set*, *std::unordered\_set* (jedes Element ist höchstens einmal enthalten)
- Stapel: *std::stack* (Funktion: "last-in-first-out") z.B. gestapelte Bierkästen.
- Warteschlange: *std::queue* (Funktion: "first-in-first-out")
- Kartendeck: *std::deque* gleichzeitig Stapel und Warteschlange
- Stapel mit Priorität: *std::priority\_queue* (Priorität vom Nutzer definiert)



## 9 Iteratoren

- für Arrays lautet kanonische Schleife:

```
for (int k = 0; k != v.size(); ++k) {
 int current = v[k]; // aktuelles Element lesen
 v[k] = new_value; // aktuelles Element schreiben
}
```

- wir wollen eine so einfache Schleife für beliebige Container
  - der Index-Zugriff  $v[k]$  ist bei den meisten Containern nicht effizient
  - Iteratoren sind immer effizient  $\Rightarrow$  es gibt sie in allen modernen Programmiersprachen, aber die Details sind sehr unterschiedlich
  - Analogie: Zeiger einer Uhr, Cursor in Textverarbeitung  
 $\Rightarrow$  ein Iterator zeigt immer auf ein Element des Containers oder auf Spezialwert “ungültiges Element”
  - in C++ unterstützt jeder Iterator 5 Grundoperationen

1. Iterator auf erstes Element erzeugen:

```
auto iter = v.begin(); // auto ist Universaltyp, wird
 // vom Compiler automatisch
 // mit richtigen Typen ersetzt
```

2. Iterator auf “ungültiges Element” erzeugen:

```
auto end = v.end() // typischerweise v[v.size()]
```

3. Vergleich:

```
iter1 == iter2;
iter != end; (= !(iter == end)) // iter zeigt nicht auf
 // ungültiges Element
```

4. zum nächsten weitergehen: ++iter, Ergebnis ist v.end(), wenn man vorher beim letzten Element war
5. auf Daten des aktuellen Elements zugreifen: \*iter (“Dereferenzierung“)

- $\Rightarrow$  kanonische Schleife:

```
for (auto iter = v.begin(); iter != v.end(); ++iter) {
 int current = *iter; // lesender Zugriff;
 *iter = new_value; // schreibender Zugriff

 // Abkürzung in C++: rang-based for-loop
 for (int & element : v) {
 int current = element; // lesen
 element = new_value; // schreiben
 }
}
```

- wenn die zugrunde liegenden Speicherzellen geändert werden, also die Containergröße sich ändert, werden die Iteratoren ungültig
- Iteratoren mit den 5 Grundoperationen heißen “forward iterators“ (wegen `++iter`)
- “bidirectional iterators“ unterstützen auch `--iter` (alle Iteratoren aus Standardbibliothek)
- “random access iterators“ können beliebige Sprünge machen (`iter+ = 5`)  
unterstützt von `std::string` und `std::vector`
- Besonderheit für assoziative Arrays (`std::map`):
  - Schlüssel und Werte können beliebig gewählt werden  
⇒ das aktuelle Element ist immer ein Schlüssel/Wert-Paar  
(`*iter`).`first` ⇒ Schlüssel  
(`*iter`).`second` ⇒ Wert

```
v[(*iter).first] == (*iter).second;
```

- Bei `std::map` liefern die Iteratoren die Elemente in aufsteigender Reihenfolge der Schlüssel (Unterschied zu `std::unordered_map`)

## 9.1 Die Funktion `std::transform()`

Die Funktion `std::transform()`

- `std::transform()` erlaubt, die Daten “on-the-fly“ zu ändern  
z.B. nach Kleinbuchstaben konvertieren:

```
std::string source = "aAbCdE"; std::string = target = source; // Target
 muss gleiche Laenge haben
std::transform(source.begin(), source.end(), target.begin(), std::tolower);
//Name einer Funktion, die ein einzelnes Element transformiert
```

- z.B. die Daten quadrieren:

```
double sq (double x) {
 return x*x;
}

std::transform(source.begin(), source.end(), target.begin(), sq);
```

- das ist eine Abkürzung für eine Schleife: (zwei Schleifen auf einmal)

```
auto src_begin = source.begin();
auto src_end = source.end();
auto tgt_begin = target.begin();

for (; src_begin != src_end; ++src_begin, ++tgt_begin) {
 // mehrere Inkrementierungen durch , getrennt
 *tgt_begin = sq(*src_begin); // mit * Bezug zu originalen Daten
}
```

- der Argumenttyp der Funktion muss mit dem *source*-Elementtyp kompatibel sein
- der Argumenttyp der Funktion muss mit dem *target*-Elementtyp kompatibel sein
- Das letzte Argument von `std::transform()` muss ein Funktor sein ( $\cong$  verhält sich wie eine Funktion)

Dazu gibt es drei Varianten:

1. normale Funktion, z.B. `sq` Aber wenn die Funktion für mehrere Argumenttypen überladen ist, muss der Programmierer dem Compiler sagen, welche Version gemeint ist  
 $\Rightarrow$  ("function pointer cast")
2. Funktorobjekte  $\Rightarrow$  objekt-orientierte Programmierung
3. definiere eine namenlose Funktion  $\cong$  "Lambda-Funktionen"  $\lambda$   
 statt  $\lambda$  wird in `C++` `[]` geschrieben

```
std::transform(source.begin(), source.end(), target.begin(),
[] (double x) { // statt Funktionsname sq, wie bei 1.
 // steht hier die ganze Funktionsimplementierung
 return x*x
}); // der Returntyp wird automatisch eingesetzt, wenn es
 // nur einen Returntyp gibt
```

- Lambda-Funktionen können noch viel mehr  $\Rightarrow$  für Fortgeschrittene
- `std::transform` kann "in-place" arbeiten (d.h. source-Container überschreiben), wenn source und target gleich
- die Funktion `std::sort()` wird zum "in-place" sortieren eines Arrays

```
std::vector<double> v = {4.0, 2.0, 3.0, 5.0, 1.0};
std::sort(v.begin(), v.end()); // -> v = {1.0, 2.0, 3.0, 4.0, 5.0}
```

- `std::sort()` ruft intern den " $<$ " Operator des Elementtyps auf, um die Reihenfolge zu bestimmen

Def: "totale Ordnung"

- \*  $a < b$  muss  $\forall a, b$  gelten
- \* transitiv:  $(a < b) \wedge (b < c) \Rightarrow (a < c)$
- \* anti-symmetrisch:  $!(a < b) \wedge !(b < a) \Rightarrow a == b$

## 9.2 Insertion Sort

schnellste Sortieralgor. für kleine Arrays ( $n \leq 30$ , hängt vom Compiler & CPU ab)

- für große Arrays sind Merge Sort, Heap Sort, Quick Sort schneller
- `std::sort()` wählt automatisch einen schnellen Algor.

Idee von Insertion Sort: wie beim Aufnehmen und Ordnen eines Kartenblatts

- gegeben: bereits sortiertes Teilarray bis zur Position  $k - 1$

- füge das  $k$ -te Element an der richtigen Stelle ein. Erzeuge Lücke an der richtigen Position durch Verschieben von Elementen nach rechts
- wiederhole für  $k = 1, \dots, N$  (siehe Übung 5.1 “Einsortieren“)

```

4 2 3 5 1
4 3 5 1 (current = 2)
 4 3 5 1
2 4 3 5 1
2 4 5 1 (current = 3)
 4 5 1
2 3 4 5 1
2 3 4 5 1
2 3 4 1 (current = 5)
2 3 4 5 1
2 3 4 5 1 (current = 1)
1 2 3 4 5

```

```

void insertion_sort(std::vector<double> &v) {
 for (int k = 1; k < v.size(); ++k) {
 double current = v[k];
 int j = k; // Anfangsposition der Luecke
 while (j > 0) {
 if (v[j-1] < current) {
 break; // j ist richtige Position der Luecke
 }
 v[j] = v[j-1];
 --j;
 }
 v[j] = current; // current in die Luecke kopieren
 }
}

```

- andere Sortierung: definiere Funktor  $cmp(a, b)$ , der das gewünschte “kleiner“ realisiert  $\cong$  gibt genau dann *true* zurück, wenn  $a$  “kleiner  $b$  nach neuer Sortierung
- neue Sortierungen am besten per Lambda-Funktion an `std::sort` übergeben

```

std::sort(v.begin(), v.end()); // Standardsortierung aufsteigend

std::sort(v.begin(), v.end(), // Standardsortierung aufsteigend
 [](double a, double b) {
 return a < b;
 })

std::sort(v.begin(), v.end(), // absteigende Sortierung
 [](double a, double b) {
 return b < a;
 })

std::sort(v.begin(), v.end(), // normale Sortierung nach Betrag
 [](double a, double b) {

```

```

 return std::abs(a) < std::abs(b);
 }
)

// Stringvergleich
std::vector<std::string> v = {"Ac", "ab", "De", "cf"};
std::vector<std::string> v = {"Ac", "De", "ab", "cf"} // case
insensitive
std::vector<std::string> v = {"ab", "Ac", "cf", "De"} // case sensitive

```

- Das letzte Argument von `std::transform()` muss ein Funktor sein ( $\cong$  verhält sich wie eine Funktion)

Dazu gibt es drei Varianten:

1. normale Funktion, z.B. `sq` Aber wenn die Funktion für mehrere Argumenttypen überladen ist, muss der Programmierer dem Compiler sagen, welche Version gemeint ist  
 $\Rightarrow$  ("function pointer cast")
2. Funktorobjekte  $\Rightarrow$  objekt-orientierte Programmierung
3. definiere eine namenlose Funktion  $\cong$  "Lambda-Funktionen"  $\lambda$   
 statt  $\lambda$  wird in `C++` geschrieben

```

std::transform(source.begin(), source.end(), target.begin(),
[] (double x) { // statt Funktionsname sq, wie bei 1.
 // steht hier die ganze Funktionsimplementierung
 return x*x
}); // der Returntyp wird automatisch eingesetzt, wenn es
 // nur einen Returntyp gibt

```

- Lambda-Funktionen können noch viel mehr  $\Rightarrow$  für Fortgeschrittene
- `std::transform` kann "in-place" arbeiten (d.h. source-Container überschreiben), wenn source und target gleich
- die Funktion `std::sort()` wird zum "in-place" sortieren eines Arrays

```

std::vector<double> v = {4.0, 2.0, 3.0, 5.0, 1.0};
std::sort(v.begin(), v.end()); // -> v = {1.0, 2.0, 3.0, 4.0, 5.0}

```

- `std::sort()` ruft intern den " $<$ " Operator des Elementtyps auf, um die Reihenfolge zu bestimmen

Def: "totale Ordnung"

- \*  $a < b$  muss  $\forall a, b$  gelten
- \* transitiv:  $(a < b) \wedge (b < c) \Rightarrow (a < c)$
- \* anti-symmetrisch:  $!(a < b) \wedge !(b < a) \Rightarrow a == b$

## 9.3 Insertion Sort

schnellste Sortialgor. für kleine Arrays ( $n \leq 30$ , hängt vom Compiler & CPU ab)

- für große Arrays sind Merge Sort, Heap Sort, Quick Sort schneller
- `std::sort()` wählt automatisch einen schnellen Algor.

Idee von Insertion Sort: wie beim Aufnehmen und Ordnen eines Kartenblatts

- gegeben: bereits sortiertes Teilarray bis zur Position  $k - 1$
- füge das  $k$ -te Element an der richtigen Stelle ein. Erzeuge Lücke an der richtigen Position durch Verschieben von Elementen nach rechts
- wiederhole für  $k = 1, \dots, N$  (siehe Übung 5.1 “Einsortieren“)

```

4 2 3 5 1
4 3 5 1 (current = 2)
 4 3 5 1
2 4 3 5 1
2 4 5 1 (current = 3)
 4 5 1
2 3 4 5 1
2 3 4 5 1
2 3 4 1 (current = 5)
2 3 4 5 1
2 3 4 5 1 (current = 1)
1 2 3 4 5

```

```

void insertion_sort(std::vector<double> &v) {
 for (int K = 1; K < v.size(); ++K) {
 double current = v[K];
 int j = K; // Anfangsposition der Luecke
 while (j > 0) {
 if (v[j-1] < current) {
 break; // j ist richtige Position der Luecke
 }
 v[j] = v[j-1];
 --j;
 }
 v[j] = current; // current in die Luecke kopieren
 }
}

```

- andere Sortierung: definiere Funktor  $cmp(a, b)$ , der das gewünschte “kleiner“ realisiert  $\cong$  gibt genau dann `true` zurück, wenn  $a$  “kleiner  $b$  nach neuer Sortierung
- neue Sortierungen am besten per Lambda-Funktion an `std::sort` übergeben

```

std::sort(v.begin(), v.end()); // Standardsortierung aufsteigend

std::sort(v.begin(), v.end(), // Standardsortierung aufsteigend
 [](double a, double b) {
 return a < b;
 })

```

```

std::sort(v.begin(), v.end(), // absteigende Sortierung
 [](double a, double b) {
 return b < a;
 }
)

std::sort(v.begin(), v.end(), // normale Sortierung nach Betrag
 [](double a, double b) {
 return std::abs(a) < std::abs(b);
 }
)

// Stringvergleich
std::vector<std::string> v = {"Ac", "ab", "De", "cf"};
std::vector<std::string> v = {"Ac", "De", "ab", "cf"} // case
 insensitive
std::vector<std::string> v = {"ab", "Ac", "cf", "De"} // case sensitive

std::sort(v.begin(), v.end(),
 [](std::string a, std::string b) {
 std::transform(a.begin(), a.end(), a.begin(), std::tolower);
 std::transform(b.begin(), b.end(), b.begin(), std::tolower);

 return a < b;
 }
);

```

## 10 Templates

*insertion\_sort* soll für beliebige Elementtypen funktionieren:

```

template <typename ElementType>

void insertion_sort(std::vector<ElementType> & v) {
 for (int k = 1; k < v.size(); ++k) {
 ElementType current = v[k];
 ... // Rest unverändert
 }
}

```

“ElementType“ ist Platzhalter für den tatsächlichen Elementtyp und wird vom Compiler automatisch ersetzt.

## 11 Grundlagen der generischen Programmierung

- Ziel: benutze *template*-Mechanismus, damit eine Implementation für viele verschiedene Typen verwendbar ist  
erweitert funktionale und prozedurale und objekt-orientierte Programmierung
- zwei Arten von Templates (“Schablonen“)
  1. Klassen-Templates für Datenstrukturen, z.B. Containersollen beliebige Elementtypen unterstützen

- Implementation  $\Rightarrow$  später
- Benutzung: Datenstrukturname gefolgt vom Elementtyp in spitzen Klammern  
`std::vector<double>`

## 2. Funktionen-Templates: es gab schon “function overloading“

Beispiel:

```
int sq (int x) {
 return x * x;
}

double sq (double x) {
 return x * x;
}
... usw fuer komplexe Zahlen
```

$\Rightarrow$  Nachteile:

- wenn die Implementationen gleich sind nutzlose Arbeit
- Redundanz ist gefährlich, korrigiert man ein Bug, wird leicht eine Variante vergessen

## 11.1 Funktionen-Templates

mit Templates reicht eine Implementation:

```
template <typename T> // T ist Platzhalter fuer beliebigen Typ
// wird spaeter durch einen tatsaechlichen Typ ersetzt
T sq (T x) {
 return x * x; // implizierte Anforderung an den Typ:
 // er muss Multiplikation unterstuetzen
}
```

- Benutzung:
  - Typen für die Platzhalter hinter dem Funktionsnamen in spitzen Klammern
  - meist kann man die Typangabe `<type>` weglassen, weil der Compiler sie anhand des Argumenttyps automatisch einsetzt
- kombiniert man Templates mit Overloading, wird die ausprogrammierte Variante vom Compiler bevorzugt
- Funktion, die ein Array aus Konsole ausgibt:

```
std::vector<double> v = {1.0, 2.0, 3.0};
print_vector(v); // {1.0, 2.0, 3.0}
```

für beliebige Elementtypen:

```
template <typename ElementType>
void print_vector(std::vector<ElementType> const & v) {
 // const: read-only, &: nur Kopie verwenden
 std::cout << "{";
 if (v.size() > 0) {
```



```

 std::cout << " " << v[0];
 for (int k = 1; k < v.size(); ++k) {
 std::cout << " " << v[k];
 }
 }
 std::cout << " }";
}

```

- Verallgemeinerung für beliebige Container mittels Iteratoren:

```

std::list<int> l = {1,2,3};
print_container (l.begin(), l.end()) // {1,2,3}

```

- es genügen forward iterators

```

Iterator iter2 = iter; // Kopie erzeugen
++iter1; // zum naechsten Element
iter1 == iter2, iter1 != end // zeigen sie auf das selbe Element
+ iter1 // Zugriff auf aktuelles Element

template <typename Iterator>

void print_container(Iterator begin, Iterator end) {
 std::cout << "{";
 if (begin != end) { // teste, ob Container leer
 std::cout << " " << *begin;
 ++begin;
 for (; begin != end; ++begin) {
 std::cout << ", " << *begin;
 }
 }
 std::cout << "}";
}

```

- Beispiel 3: checken, ob Container sortiert ist

```

Version 1: hard-coded

bool check_sorted (std::vector<double> const & v) {
 for (int k = 1; k < v.size(); ++k) {
 if (v[k] < v[k-1]) { // Sortierfehler durch Ausnutzen
 // der Transitivitaet
 return false;
 }
 }
 return true; // Schleife ohne Fehler zuende gelaufen
}

Version 2: beliebige Elementtypen, beliebige Sortierung

template <typename ElementType, typename LessThanFunctor>

bool check_sorted(std::vector<ElementType> const & v, typename
LessThanFunctor) {
 for (int k = 1; k < v.size(); ++k) {
 if (less_than(v[k], v[k-1])) {

```

```

 return false;
 }
}
return true;
}

```

- Aufruf von Version 2 mit “lambda-function“:

```

std::vector<double> v = {1.0, 2.0, 3.0};
check_sorted(v, [] (double a, double b) {
 return a<b;
}); // true

check_sorted(v, [] (double a, double b) {
 return a>b;
}); // true

```

- Version 3 mit “forward-iterator“:

```

template<typename Iterator, typename LessThanFunctor>
bool check_sorted(Iterator begin, Iterator end,
 LessThanFunctor less_than)
{
 if (begin == end) { // Container leer?
 return true; // leerer Container immer sortiert
 }
 Iterator next = begin;
 ++next; // next zeigt auf Element nach Iterator begin

 for (; next != end; ++begin, ++next) { // Iteratoren zeigen
 // auf zwei benachbarte Elemente
 if (less_than(*next, *begin)) {
 return false;
 }
 }
}

```

- Bemerkungen:

1. Compiler-Fehlermeldungen bei Template-Code sind oft schwer zu implementieren ⇒ Erfahrung nötig
2. mit Templates kann man noch viel raffiniertere Dinge machen, z.B. Traits-Klassen, intelligent libraries, template metaprogramming ⇒ nur für Fortgeschrittene

## 12 Bestimmung der Effizienz von Algorithmen und Datenstrukturen

- 2 Möglichkeiten

1. messe die “wall clock time“ (wie lange muss man auf ein Ergebnis warten)
2. unabhängig von Hardware: algorithmische Komplexität

- “wall-clock-time“ misst man z.B. mit dem Modul `< chrono >`

```
#include <chrono>
#include <iostream>

int main() {
 ... // alles zur Zeitmessung vorbereiten, z.B. Daten einlesen

 auto start = std::chrono::high_resolution_clock::now(); // Startzeit
 merken
 ... // Code, der gemessen werden soll
 auto stop = std::chrono::high_resolution_clock::now(); // Endzeit
 merken
 std::chrono::duration<double> diff = stop-start; // Zeitdifferenz (
 Laufzeit) in Sekunden
 std::cout << "Zeitdauer: " << diff() << " sekunden \n";)
}
```

- in der Praxis nicht so einfach  $\Rightarrow$  Pitfalls:
  - moderne Compiler optimieren oft zu viel, d.h. komplexe Berechnungen werden zur Compilezeit ausgeführt und ersetzt  $\Rightarrow$  gemessene Zeit viel zu kurz gegenüber der Praxis  
Abhilfe: Daten nicht “hard-wired“, sondern z.B. von Platte lesen (*volatile* beim Initialisieren)
  - der Algorithmus ist schneller als die clock  
Abhilfe: rufe den Algorithmus mehrmals in einer Schleife auf
  - die Ausstattung des Programms kann vom Betriebssystem jederzeit für etwas wichtigeres unterbrochen werden  $\Rightarrow$  gemessene Zeit ist zu lang  
Abhilfe: messe mehrmals und nimm die kürzeste Zeit (meist reicht 3-10x)
  - Faustregel: Messung zwischen 0.02-3 Sekunden zur optimalen Nutzung der clock
- Nachteil: Zeit hängt besonders von der Qualität der Implementation, den Daten und der Hardware ab
- algorithmische Komplexität ist davon unabhängig  $\cong$  “theoretisches Effizienzmaß“  
beschreibt, wie sich die Laufzeit verlängert, wenn man mehr Daten hat

$\Rightarrow$  bei effizienten Algorithmen steigt der Aufwand mit  $n$  nur langsam (oder bestenfalls gar nicht)

## 12.1 technisches Effizienzmaß

- berechne, wie viele elementare Schritte der Algorithmus in Abhängigkeit von  $n$  benötigt  
 $\Rightarrow$  komplizierte Formel  $f(n)$
- vereinfache  $f(n)$  in eine einfache Formel  $g(n)$ , die dasselbe wesentliche Verhalten zeigt  
Die Vereinfachung erfolgt mittels O-Notation und ihren Verwandten

## 12.2 $\mathcal{O}$ -Notation/ $\Omega$ -Notation

1.  $g(n)$  ist eine asymptotische (für große  $n$ ) obere Schranke für  $f(n)$  ( $f(n) \leq g(n)$ )  
 $f(n) \in \mathcal{O}(g(n))$  ( $f(n)$  ist in der Komplexitätsklasse  $g(n)$ , wenn es ein  $n_0$  und  $C$  gibt, sodass  $\forall n > n_0 : \Leftrightarrow f(n) \in \mathcal{O}(g(n))$ )
2.  $g(n)$  ist asymptotisch untere Schranke für  $f(n)$  ( $f(n) \geq g(n)$ )  
 $f(n) \in \Omega(g(n)) \Leftrightarrow \exists n_0, C, \text{ sodass } \forall n > n_0 : f(n) \geq C \cdot g(n)$
3.  $g(n)$  ist asymptotisch scharfe Schranke für  $f(n)$  ( $f(n) = g(n)$ )  
 $f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in \mathcal{O}(g(n)) \wedge f(n) \in \Omega(g(n))$

Regeln:

1.  $f(n) \in \Theta(f(n)) \Rightarrow f(n) \in \mathcal{O}(f(n)), f(n) \in \Omega(f(n))$
2.  $f(n) \in \Theta(f(n)) \Rightarrow C \cdot f(n) \in \Theta(f(n))$
3.  $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) \in \mathcal{O}(f(n) \cdot g(n))$
4.  $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) \in \mathcal{O}(\max(f(n), g(n)))$   
 formell:  $f(n) \in \mathcal{O}(g(n)) \Rightarrow \mathcal{O}(f(n)) + \mathcal{O}(g(n)) \in \mathcal{O}(g(n))$   $g(n) \in \mathcal{O}(f(n)) \Rightarrow \mathcal{O}(g(n)) + \mathcal{O}(f(n)) \in \mathcal{O}(f(n))$

– beliebteste Wahl für  $g(n)$ :

- \*  $\mathcal{O}(1)$  “konstante Komplexität“, Bsp: elementare Operationen, Array-Zugriff
- \*  $\mathcal{O}(\log(n))$  “logarithmische Komplexität“, Bsp: Zugriff auf ein Element von `std::map`
- \*  $\mathcal{O}(n)$  “lineare Komplexität“, Bsp: `std::transform`
- \*  $\mathcal{O}(\log(n) \cdot n)$  “ $n \cdot \log(n)$ “, “quasilinear“, Bsp: `std::sort()`
- \*  $\mathcal{O}(n^2)$  “quadratische Komplexität“
- \*  $\mathcal{O}(n^p)$   $p = \text{const.}$  “polynomelle Komplexität“
- \*  $\mathcal{O}(2^n)$  “exponentielle Komplexität“

– Beispiele:

$$f(n) = 1 + 15n + 4n^2 + 7n^3 \in \mathcal{O}(n^3)$$

$$f(n) = n \cdot \log(n) + n^2 \in \mathcal{O}(n \cdot \log(n) + n \cdot n) \in \mathcal{O}(n) \cdot \mathcal{O}(\log(n) + n) \in \mathcal{O}(n) \cdot \mathcal{O}(n) \in \mathcal{O}(n^2)$$

$\Rightarrow$  es gewinnt immer die am stärksten wachsende Funktion

Anwendung 1: Fibonacci-Zahlen:  $f_k = f_{k-2} + f_{k-1}$

| k     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8  |
|-------|---|---|---|---|---|---|---|----|----|
| $f_k$ | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 |

```
int fib1 (int k) { // O(k)
 if (k<2) { // O(1)
 return k; // O(1)
 }
 int f1 = 1; // O(1)
 int f2 = 1; // O(1)

 for (int j = 2; j<=k; ++j) { // O(k)
 int f = f1 + f2; // O(1)
 }
}
```

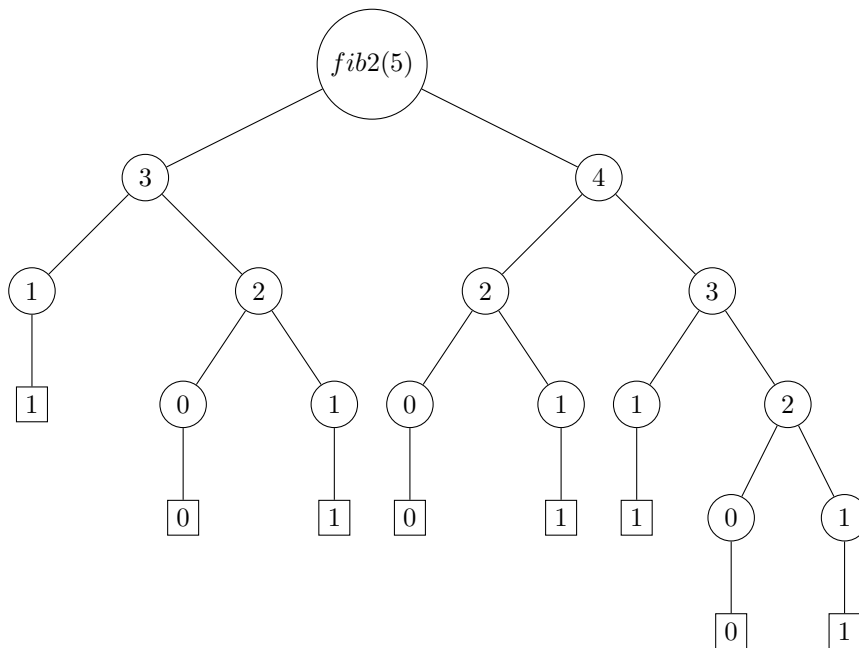
```

 f1 = f2; // O(1)
 f2 = f; // O(1)
}

return f2; // O(1)
}

int fib2 (int k) {
 if (k<2) {
 return k;
 }
 return fib2(k-2)+fib2(k-1));
}

```



⇒ sehr ineffizient, weil alle Fib-Zahlen  $< k$  mehrmals berechnet werden

Sei  $f(k)$  die Anzahl der Schritte, Annahme: jeder Knoten ist  $\mathcal{O}(1) \Rightarrow \mathcal{O}(\text{Knoten})$  Sei  $f'(k)$  die Anzahl der Schritte, oberhalb (oberhalb ist der Baum vollständig (jeder innere Knoten hat genau 2 Kinder))

$$\begin{aligned}
 f'(k) &= 2^{l+1} - 1 \\
 &= 2^{k/2+1} - 1 \\
 &= 2 \cdot 2^{k/2} - 1 \\
 &= 2 \cdot (\sqrt{2})^k - 1 \\
 &\in \Omega(\sqrt{2})^k \\
 &\Rightarrow \text{exponentielle Komplexität}
 \end{aligned}$$

## 13 Zahlendarstellung

Problem: unendlich viele Zahlen, aber die Computer sind endlich

### 13.1 natürliche Zahlen $\mathbb{N}$

$x \geq 0$ , (C++ bietet Typen verschiedener Größe)

| klassisch          | mit Größe (C++) | # Bits       | Bereiche              | Literale |
|--------------------|-----------------|--------------|-----------------------|----------|
| unsigned char      | uint8_t         | ( $\geq$ )8  | 0 – 256               |          |
| unsigned short     | uint16_t        | ( $\geq$ )16 | 0 – 65.535            |          |
| unsigned int       | uint32_t        | ( $\geq$ )32 | 0 – $4 \cdot 10^9$    |          |
| unsigned long      | uint32_t        | 32 oder 64   | 0 – $4 \cdot 10^9$    |          |
| unsigned long long | uint64_t        | 64           | 0 – $2 \cdot 10^{19}$ | 4L       |

Was passiert bei zu großen Zahlen?

- alle Operationen werden Modulo  $2^m$  ausgeführt, wenn der Typ  $m$  Bits hat  
Bsp 1:

```
uint8_t x = 250, y = 100;
uint8_t s = x+t; // 350 % 256 = 94
uint8_t p = x*y; // 2500 % 256 = 168
```

- Pitfalls  
Beispiel 1: Mittelwert eines `uint8_t`-Arrays

```
std::vector<uint8_t> v = {...}
uint8_t sum = 0; // uint32_t od. uint64_t
for (int k = 0; k < v.size(); ++k) {
 sum += v[k];
}
std::cout << "Mittelwert: " << (sum(v.size())) << "\n";
```

`uint32_t sum = 0` verhindert overflow mit hoher Wahrscheinlichkeit

Bsp 2: Count-Down Loop (rückwärts über Array)

```
for (uint8_t k = v.size() - 1; k >= 0; --k) {
 .. // v[k] zugreifen
}
uint8_t x = 0;
--x;
```

- arithmetische Op. Addition in Kapitel "Automaten"  
Substitution kann auf Addition zurückgeführt werden

**Erinnerung: Restklassenarithmetik (Modulo)** alle Zahlen mit dem gleichen Rest modulo  $k$  bilden "Äquivalenzklasse"

hier: kleinste Repräsentanten  $0, \dots, k-1$  mit  $k = 2^m$

Eigenschaft: man kann Vielfache  $n \cdot k$  addieren, ohne Äquivalenzklasse zu ändern  
 $\Rightarrow$  implementiere (Addition besser als Subtraktion)

$$\begin{aligned} & (a - b) \% 2^m \\ &= (a + 2^m - b) \% 2^m \\ &= (a + z) \% 2^m \end{aligned}$$

**bitweise Negation** dreht alle Bits um

$$\begin{aligned} m = 4 & \sim (1001) \Rightarrow (0110) \\ \text{setze : } & (2^m - b) \% 2^m = \sim (b + 1) \% 2^m \\ & b + \sim b = 11 \dots 11 = 2^m - 1 \\ & \sim b + 1 = 2^m - b \end{aligned}$$

Fall 1:

$$\begin{aligned} b > 0 & \Rightarrow \sim b < 2^m - 1 \\ & \Rightarrow \sim (b + 1) < 2^m \\ \sim (b + 1) \% 2^m &= \sim (b + 1) \\ (2^m - b) \% 2^m &= \sim b \% 2^m \end{aligned}$$

Fall 2:

$$\begin{aligned} b = 0 & \Rightarrow \sim b = 2^m - 1 \\ \sim b + 1 &= 2^m \\ (\sim b + 1) \% 2^m &= 0 \\ 2^m - b &= 2^m \end{aligned}$$

## Multiplikation

- neue Operationen  $\ll$  und  $\gg$  (left und right shift) verschiebt die Bits um  $k$  Positionen nach links oder rechts. Die herausgeschobenen Bits werden vergessen, auf der anderen Seite durch 0-Bits ersetzt.

$$\begin{aligned} m = 8 : \quad & 11011101 \ll 3 = 11101000 \\ & 11011101 \gg 3 = 00011011 \end{aligned}$$

- Satz:

$$x \ll k = (x * 2^m) \% 2^m \quad (1)$$

- Operationen  $\&$  und  $|$  sind bitweise “und” bzw. “oder” Verknüpfungen nicht verwechseln mit  $\&\&$  bzw.  $||$  für logische Operatoren für  $m = 8$ :

```

10110011 & 1
00000001

00000001

10110011 | 1
00000001

10110011

```

```

uint8_t mal(uint8_t x, uint8_t y) {
 uint8_t res = 0;
 for (int k = 0; k < 8; ++k) {
 if (y & (1 << k) != 0) {
 res += k;
 }
 x = x << 1; // = x*2
 }
 return res;
}

```

## 13.2 ganze Zahlen $\mathbb{Z}$

| klassisch        | Typ mit Größe  | Bits       | Bereich                    |
|------------------|----------------|------------|----------------------------|
| signed char      | <i>int8_t</i>  | 8          | $-127 \dots 128$           |
| signed short     | <i>int16_t</i> | 16         | $-2^{15} \dots 2^{15} - 1$ |
| signed int       | <i>int32_t</i> | 32         | $-2^{31} \dots 2^{31} - 1$ |
| signed long      | <i>int32_t</i> | 32 oder 64 | $-2^{63} \dots 2^{63} - 1$ |
| signed long long | <i>int64_t</i> | 64         | $-2^{63} \dots 2^{63} - 1$ |

für Restklassen: statt  $0 \dots 2^m$  bei unsigned  
jetzt:  $-2^{m-1} \dots 2^{m-1} - 1$  (symmetrisch um 0)

d.h.  $x < 2^{m-1}$ : Repräsentant bleibt

$x \geq 2^{m-1}$ : neuer Repräsentant,  $x - 2^m$  (gleiche Restklasse)

Konsequenzen:



- bei negativer Zahl ist höchste Bit 1, weil  $x \rightarrow x - 2^m$ , falls  $x \geq 2^{m-1}$
- unäre Negation  $-x$  durch Zweierkomplement:

$$\begin{aligned}
 -x &= (\sim x + 1) \% 2^m \\
 \text{Bsp : } -0 &= (\sim 000000 + 1) \% 2^8 \\
 &= (1111111 + 1) \% 2^8 \\
 &= 100000000 \% 2^8 \\
 &= 0 \\
 \text{Bsp : } -1 &= (\sim 00000001 + 1) \% 2^8 \\
 &= (\sim 11111110 + 1) \% 2^8 \\
 &= 11111111 \% 2^8 \\
 &= 11111111 \\
 &= 2^8 - 1 < 2^8
 \end{aligned}$$

- Ausnahmeregel: für  $\gg$  bei negativen Zahlen Compilerabhängig, meist wird links ein Bit reingeschrieben, damit Zahl negativ bleibt  $\Rightarrow$  es gilt immer noch  $x \gg h = \lfloor x/2^k \rfloor$

### 13.3 reelle Zahlen $\mathbb{R}$

Problem: unendlich viele Zahlen

|                 | Name        | Größe  | Bereich                    | kleinste    | Literale |
|-----------------|-------------|--------|----------------------------|-------------|----------|
| Lösung in $C++$ | float       | 32 Bit | $-10^{-38} \dots 10^{38}$  | $10^{-38}$  | $4.0f$   |
|                 | double      | 64 Bit | $-10^{308} \dots 10^{308}$ | $10^{-308}$ | 4.0      |
|                 | long double |        |                            |             |          |

- der  $C++$  Standard legt Größe nicht fest, aber alle gängigen CPUs benutzen Standard *IEEE754*  
 $C++$  übernimmt Hardware-Implementation
- Ziele bei Definition von reellwertigen Zahlen:
  - hohe Genauigkeit (viele gültige Nachkommastellen)
  - a
- elegante Lösung: halb-logarithmische Darstellung (“floating-point”)
 

Datentyp ist aus rein natürlichen Zahlen zusammengesetzt (aber alles von CPU gekapselt)

  1.  $S$  (1-bit): Vorzeichen ( $0 \approx +$ ,  $1 \approx -$ )
  2.  $M$  (m-bits): Mantisse: Nachkommastellen
  3.  $E$  (e-bits, Basis  $b$ ): Exponent/Größenordnung
- die eigentliche Zahl wird berechnet durch:
 
$$x = (-1)^s \cdot (1 + M \cdot 2^{-m}) \cdot 2^{E-b}$$

| x | $M \cdot 2^{-m}$ | $E - b$ | effektive Darstellung |
|---|------------------|---------|-----------------------|
| 1 | 0                | 0       | $1 \cdot 2^0$         |
| 2 | 0                | 1       | $1 \cdot 2^1$         |
| 3 | 0.5              | 1       | $1.5 \cdot 2^1$       |
| 4 | 0                | 2       | $1 \cdot 2^2$         |
| 5 | 0.25             | 2       | $1.25 \cdot 2^2$      |

**Konsequenz** alle ganzen Zahlen zwischen  $-2^m + \dots + 2^m$  können exakt dargestellt werden und exakte Arithmetik

**Werte für  $m, e, b$  (IEEE754)**

- float:  $m = 23, e = 8, b = 127$   
 $2^{E-b} \in [2^{-126}, 2^{127}] \approx [10^{-38}, 10^{38}]$
- double:  $m = 52, e = 11, b = 1023$   
 $2^{E-b} \in [2^{-1022}, 2^{1023}] \approx [10^{-308}, 10^{308}]$

**Anzahl Nachkommastellen**

- allgemein:  $2^{-m}$
- float:  $2^{-23} \approx 10^{-7}$
- double:  $2^{-52} \approx 2 \cdot 10^{-16}$
- $\varepsilon = 2^{-m}$  = machine epsilon, unit last place (ULP)
- $\varepsilon$  ist die kleinste Zahl, so dass  $(1.0 \cdot \varepsilon)! = 1.0$ , weil Nachkommastellen außerhalb der Mantisse (rechts von  $2^{-m}$ ) ignoriert werden  
 $\Rightarrow$  Problem der Auslöschung von signifikanten Stellen: wenn man zwei fast gleich große Zahlen subtrahiert, löschen sich fast alle Bits der Mantisse  $\Rightarrow$  nur wenige gültige Nachkommastellen überleben
- Bsp 1:  $0.1234567 - 0.1234566 = 0.0000001$  (eine gültige Nachkommastelle)
- Bsp 2:  $10 - \cos(x)$  für kleine  $x$ :

|                                                                   | x         | # gültige Stellen         | Additionstheorem |
|-------------------------------------------------------------------|-----------|---------------------------|------------------|
| für $x \approx 0$ ist $\cos(x) \approx 1 \Rightarrow$ Auslöschung | 0.0001    | 9 (statt 15.5)            | 15.5             |
|                                                                   | $10^{-8}$ | 0 ( $\cos(10^{-8}) = 1$ ) | 15.5             |

Additionstheorem:  $1 - \cos(x) = 2(\sin(x/2.0))^2$

- Bsp 3: quadratische Gleichung  $ax^2 + by + c$  mit  $b > 0$   
 $x_1 = \frac{1}{2a}(-b + \sqrt{b^2 - 4ac})$  falls  $a \cdot c > 0, b^2 \gg 4ac$   
Umstellen:  $x_1 = \frac{1}{2a}(-b + \sqrt{b^2 - 4ac}) \stackrel{=b-\sqrt{b^2-4ac}}{-b-\sqrt{b^2-4ac}}$   
 $\approx -b + b + \varepsilon \approx 0 \Rightarrow$  Auslöschung, wenig gültige Stellen  
 $\frac{1}{2a} \frac{b^2 - (b^2 - 4ac)}{-b - \sqrt{b^2 - 4ac}} = \frac{2c}{-(b + \sqrt{b^2 - 4ac})}$

- Ausnahmeregeln (spezielle Werte)
  - normal:  $E \in [1 \dots 2^e - 2]$
  - $E = 2^e - 1$  (größtmöglicher Wert):
    - für  $M = 0 : x = \pm\infty$  (abhäng. von  $S$ )
    - für  $M > 0 : x = \text{NaN}$  (“Not a Number”)
  - $E = 0$  (kleinster Wert):
    - für  $M = 0 : \pm 0$  (abhäng. von  $S$ )
    - für  $M > 0 : \text{“denormalisierte Zahlen”}$  für sehr kleine Werte

## 14 Buchstabenzeichen

“glyphs” müssen durch Zahlen repräsentiert werden: “Zeichencode”

### Geschichte

- 1963: ASCII (7-bit) Zeichen der engl. Schreibmaschine (keine Umlaute)
- 1964-2000: 8-bit codes mit Umlauten, Akzenten, kyrillisch  
aber 8-bit sind zu wenig, um alles abzudecken  
 $\Rightarrow$  viele konkurrierende 8-bit Codes
- 1991-heute: Unicode  
 anfangs 16-bit, jetzt  $\approx$  21-bit  
 $\Leftarrow$  alles (chinesisch, Emojis, Hyroglyphen)

**Unicode** 3 Codierungen für Unicode:

1. UTF-8: variable length code (pro glyph  $1 \dots 4\text{uint8\_t}$ )
  2. UTF-16: variable length code (pro glyph  $1 \dots 2\text{uint16\_t}$ )
  3. UTF-32: fixed length code (pro glyph  $1\text{uint32\_t}$ )
- char: 8-bit Codes
  - wchar\_t: 16-bit (Windows), 32-bit (Linux)
  - u16char\_t
  - u32int\_t

leider sehr plattformabhängig  $\Rightarrow$  Zeichensalat, wenn inkompatible Codes verwendet werden

$\Rightarrow$  in C++ ICU library (“International Components for Unicode”)

- hat man alle Zeichen korrekt, ist Problem noch nicht gelöst: alphabetische Sortierung sprachabhängig  
 ä: dt. Wörterbuch - wie a; dt. Telefonbuch - wie ae
- Lösung in C++:

```
std::sort(v.begin(), v.end(), std::locale("se_SE.UTF-8")) mit <locale>,
<codecvt>
```

## 14.1 eigene Datentypen

3 Möglichkeiten:

- *enum*: Aufzählungstypen  $\Rightarrow$  Selbststudium
- *struct*: strukturierte Daten, zusammengesetzte Typen
- *class*: wie *struct* auf objekt-orientiert

```
struct Typevalue {
 type_name var_name1;
 type_name var_name2;
 ...
};
```

Beispiel:

```
struct Date {
 int day;
 int month;
 int year;
}; // Datenmember = member variables

Date caster (int year) {
 ... // Osterdaten
 Date res;
 res.day = day;
 res.month = month;
}

// für Übungsaufgabe 8.4
struct Character {
 wchar_t clear;
 wchar_t encrypted;
 int count;
};
```

## 15 Objektorientierte Programmierung

### 15.1 eigene Datentypen mit Kapselung

- eigene Datentypen sind zusammengesetzt aus einfacheren/existierenden Datentypen (Ausnahme *enum*)
- zwei Arten:
  - offene Typen: Benutzer kann auf interne Daten zugreifen, “C-style types” (wichtige Änderungen aus Standardbibliothek aus C übernommen)
  - gekapselte Typen: Benutzer kann nicht auf interne Daten zugreifen (“private”) alle Benutzerzugriffe über ein öffentliches Interface (“public”) Vorteile:

1. komplexe Details zur Verwaltung bleiben verborgen
2. öffentliches Interface (hoffentlich) einfach zu benutzen  
z.B. `std :: vector`
3. interne Details können bei Bedarf geändert werden, ohne dass sich das öffentliche Interface ändert  
⇒ Benutzer muss Code nicht ändern, aber Programm geht schneller  
“Rückwärtskompatibilität”

## Wie erreicht man die Kapselung?

- zwei Schlüsselwörter für eigene Typen:
  1. `class` (Konvention in OOP)
  2. `struct` (von C übernommen)
- zwei Schlüsselwörter für die Kapselung:
  1. `public` (öffentlicher Teil)
  2. `private` (gekapselter Teil)

`class` ist standardmäßig “private”, `struct` ist “public”

```
class MyType {
 ... // private by default
 public:
 ... // jetzt oeffentlich
 private:
 ... // jetzt privat
};

struct MyType {
 ... // oeffentlich by default
 private:
 ... // jetzt privat
 public:
 ... // jetzt wieder oeffentlich
};
```

⇒ Benutzer können nur auf Funktionalität im `public`-Teil zugreifen

- die im zusammengesetzten Typ enthaltenen Daten heißen “member variables” und sind normalerweise `private`
  - kann nachträglich geändert werden  
z.B. complex in real/imaginär → Phase/Betrag
  - Benutzer kann nicht unbeabsichtigt die Konsistenz verletzen

## 15.2 running example

Punkt-Klasse für 2-dimensionalen Punkt

```
class Point {
 double x_; // Koordinate als private
 double y_; // Datenmember ('_' am Ende)
};
```

⇒ dieser Datentyp ist unbenutzbar, weil alles privat

- unverzichtbare öffentliche Funktion zum Initialisieren des Speichers: “Konstruktoren”
- Prozeduren innerhalb der Klasse, Name gleich Äquivalenzklasse
  - Prozeduren sind *void*, *void* wird weggelassen
  - nur Seiteneffekt: neue Objekte initialisieren, also die Konstruktoren der Datenmember aufrufen
- zur Erinnerung: zwei Möglichkeiten für normale Variableninitialisierung:
  - `double z = 1.0;`
  - `double z(1.0)` - nur diese Syntax ist in Konstruktoren erlaubt

```
class Point {
 double x_,
 double y_;

 public:
 Point(double x, double y) // Konstruktoraufrufe vor Prozedurrumpf
 : x_(x) // Member x_ auf Initialwert x
 , y_(y) // Member y_ auf Initialwert y
 {
 // normaler Rumpf der Prozedur, hier leer
 }
};

Point p(1.0, 2.0);
Point q = {3.0, 4.0};
```

**Standardkonstruktor**  $\cong$  Konstruktor ohne Argumente

- initialisiert Objekt in Standardzustand
- bei Zahlen: auf 0 setzen, hier auf Koordinatenursprung

```
class Point {
 ... // wie zuvor
 Point() // Standardkonstruktor
 : x_(0.0)
 , y_(0.0)
 {}
};
```

- um mit Punkt-Objekten zu arbeiten, brauchen wir weitere Funktionen:
  1. Member-Funktionen: innerhalb der Klasse definiert man (wie Konstruktoren), können auf alles *private* zugreifen, können als *private* oder *public* definiert werden
  2. freie Funktionen: normale Funktionen außerhalb der Klasse, die ein Argument des neuen Typs haben können nur auf öffentliches Interface zugreifen
- wichtigste Vertreter der Member-Functions: Zugriffsfunktionen “Getter”: erlauben Benutzer, aktuellen Zustand abzufragen (z.B. *v.size()*)

```
Point p(1.0, 2.0);
p.x() // returns 1.0 (x-Koordinate)
p.y() // returns 2.0 (y-Koordinate)
```

- Member-Funktionen werden mit Punkt-Syntax aufgerufen: *p.x()*  
Objekt vor dem Punkt ist das “nullte” Argument der Funktion  $\Rightarrow$  Compiler macht daraus *x(p)*
- bei der Implementation der Member-Funktion schreibt man “nullte” Argument nicht hin, der Compiler stellt es automatisch unter dem Namen *\*this* zur Verfügung

```
class Point {
 ... // wie vorher

 double x() {
 return (*this).x_;
 }
 double y() {
 return (*this).y_;
 }
}
```

- meist kann man *(\*this).* weggelassen werden, wenn eindeutig ist, welchen Member man meint, fügt der Compiler es automatisch ein
- Getter-Funktionen sind “read-only” (ändern die Member-Variablen nicht)  
man sollte sie deshalb mittels *const* explizit als “read-only” markieren  
Vorteile:
  1. Programmierer kann Member-Variable nicht irrtümlich ändern
  2. Funktion kann auch in Kontexten benutzt werden, wo das Objekt (nulltes Argument) explizit als “read-only” markiert ist

*Point const cp(1.0, 2.0);*

## Punkte ausgeben

- zwei Möglichkeiten:

– Member-Funktion:

```
std::cout << p.to_string() << '\n';
```

– freie Funktion:

```
std::cout << to_string(p) << '\n';
```

```
class Point { // Member-Funktion
... // wie vorher

 std::string to_string() const {
 std::string res;
 res += '[' + std::to_string((*this).x()) + ',' + std::to_string
 ((*this).y()) + ']';
 return res;
 }
};

// oder
std::string to_string() const { // freie Funktion
 std::string res;
 res += '[' + std::to_string(p.x()) + ',' + std::to_string(p.y())
 + ']';
 return res;
}
```

ws man wählt, ist Geschmackssache (freie Funktion ist kompatibel zu `std::to_string`)

## Punkte vergleichen

```
class Point {
... // wie vorher

 bool equals (Point other) const {
 return (*this).x() == other.x() && (*this).y() == other.y();
 }
};

// andere Umgebung
Point p(1.0, 2.0);
Point origin;
assert(p.equals(p));
assert(!p.equals(origin));
```

üblicher: Infix-Notation  $\Rightarrow$  dazu Prefix-Variante `operator ==` implementieren

```
class Point {
... // wie vorher

 bool equals (Point other) const {
 return (*this).x() == other.x() && (*this).y() == other.y();
 }
}
```



```

 bool operator== (Point other) const {
 return (*this).x() == other.x() && (*this).y() == other.y();
 }
 bool operator!= (Point other) const {
 return (*this).x() != other.x() || (*this).y() != other.y();
 }
};

// andere Umgebung
Point p(1.0, 2.0);
Point origin;
assert(p == p);
assert(!(p == origin));
assert(p != origin);

```

**neuen Punkt erzeugen** transponiert, d.h. x-y Koordinaten sind vertauscht

```

Point p(1.0, 2.0);
Point tp = p.transpose(); // unser Ziel

class Point {
 ... // wie vorher

 Point transpose() const {
 Point res((*this).y(), (*this).x());
 return res;
 }
};

```

verschoben

```

Point p(1.0, 2.0);
Point v (3.0, 4.0);
Point vp = p.translate(v); // unser Ziel

class Point {
 ... // wie vorher

 Point translate(Point v) const {
 Point res((*this).x() + v.x(), (*this).y() + v.y());
 return res;
 }
};

```

## 15.3 Member-Funktionen

Jede Klasse hat bestimmte spezielle Member-Funktionen:

- Konstruktor: bringt Objekt in wohldefinierten Anfangszustand
- Destruktor: entsorgt nicht mehr benötigtes Objekt (typischerweise am Ende der Umgebung)
- Zuweisungsoperatoren: um Objekte per Zuweisung ("=") zu überschreiben

**Destruktor** Jede Klasse muss genau einen haben, wenn der Programmierer das nicht explizit implementiert, fügt Compiler ihn automatisch ein

```
class Klassenname {
public:
 ~Klassenname() {
 ... Implementation
 }
};
```

- der automatisierte Destruktor ruft einfach die Destrukturen aller Member-Variablen auf
- meist ist das ausreichend, aber in bestimmten Situationen muss der Programmierer zusätzliche Aktionen implementieren
- Beispiele:
  1. manuelle Speicherverwaltung: Destruktor muss nicht mehr benötigten Speicher an Betriebssystem zurückgeben (z.B. Destruktor von `std::vector`)  
Vorteil der Kapselung: Nutzer merkt davon nichts
  2. manuelles Dateimanagement: Destruktor muss Datei schließen (=Daten aus dem Cache auf die Platte übertragen)
  3. Abmelden von einem Service (Ausloggen, Verbindung beenden)
- spezieller Konstruktor:

**Kopier-Konstruktor** zum Erzeugen einer Kopie eines vorhandenen Objekts, d.h. neue Speicherzelle mit gleichem Inhalt:

```
Point p (1.0, 2.0); // Konstruktor mit Initialwert
Point q = p; // Kopierkonstruktor
Point r(p); // Kopierkonstruktor

int foo (Point q) {
 ...
}
foo(p) // Kopierkonstruktor wegen pass-by-value

int bar (Point const & q) {
 ...
}
bar(p); // q ist neuer Name für p ohne neue Speicherzelle
```

```
class KlassenName{
public:
 KlassenName (KlassenName const & existing) {
 ...
 }
};
```

- der Compiler erzeugt Kopier-Konstruktor automatisch, falls nicht explizit programmiert (= ruft Kopier.Konstruktor für alle Member-Variablen auf)  
meistens richtig, Ausnahmen wie oben

## Standard-Konstruktor (“default constructor”)

- ohne Argumente
- bringt Objekt in Standard-Zustand, z.B. 0 bei Zahlen

```
class KlassenName {
 public:
 KlassenName() {
 ...
 }
};
```

- Compiler erzeugt Standard-Konstruktor automatisch, falls es keinen benutzerdefinierten Konstruktor gibt
- “rule-of-three”: Wenn es nötig ist, einen der drei Funktionen (Destruktor, Kopier-Konstruktor und Zuweisungskonstruktor) explizit zu implementieren, müssen alle drei explizit implementiert werden

## 15.4 Vorteile der Kapselung

- Benutzung der Klasse ist viel einfacher, weil unwichtige Details verborgen sind
- interne Implementation kann geändert werden, ohne den Benutzer zu Folgeänderungen zu zwingen, weil externe Schnittstelle erhalten bleibt

### Beispiel: Point-Klasse

```
class Point {
 double x_, y_;

 public:
 Point()
 : x_(0.0)
 , y_(0.0) {}

 Point(double x, double y)
 : x_(x)
 , y_(y) {}

 double x() const {
 return x_; // = return (*this).x_;
 }

 double y() const {
 return y_;
 }
};
```

```
Alternative: Array Länge 2:
#include <array>
std::array<double, 2> // feste Größe

class Point {
 std::array<double, 2> data_;
```

```

public:
 Point()
 : data_{0.0, 0.0} {}

 Point (double x, double y)
 : data_{x, y} {}

 double x() const {
 return data_[0];
 }

 double y() const {
 return data_[1];
 }
};

```

## 15.5 Operatoren

**Ziel der Objektorientierten Programmierung** Arbeiten mit Nutzer-definierten Datenstrukturen möglichst einfach, wie mit eingebauten (z.B. arithmetische Infix-Operationen)

```

Point p(2.0, 3.0), q(4.0, 5.0)
Point r = 2.5*p + q;
assert(r == Point(9.0, 12.5));

```

- dazu muss man die entsprechenden Prefix-Funktionen implementieren
- Addition:

```

Point operator + (Point p1, Point p2) {
 Point res (p1.x() + p2.x(), p1.y() + p2.y());
 return res;
}

// Alternative
Point operator + (Point cconst & p1, Point const & p2) {
 ... // wie zuvor
}

```

- Subtraktion, elementweise Multiplikation und Division genauso (“+” überall durch “+”, “\*”, “−” ersetzen)
- Skalierungsoperation: Multiplikation von Punkt mit Zahl, d.h. zwei verschiedene Argumenttypen (zwei Versionen für Kommutativität)

```

Point operator * (double s, Point p) {
 Point res (s * p.x(), s * p.y());
 return res;
}

// und
Point operator * (Point p, double s) {
 Point res (p.x() * s, p.y() * s);
 return res;
}

```

- alle diese Versionen können dank “function-overloading” gleichzeitig implementiert sein
- bisher: freie Funktionen
- falls das erste Argument vom Typ Point oder Point const & ist, kann man die Funktionen alternativ als Member-Funktion implementieren

```
class Point {
 ... // wie bisher
 Point operator + (Point const & p2) {
 Point res ((*this).x() + p2.x(), (*this).y() + p2.y());
 return res; // Nulltes Argument anstelle von p2 der freien Funktion
 }
};
```

## Member-Funktionen

- Vorteil von Member-Funktionen: Zugriff auf private Member der Klasse (hier nicht notwendig)
- Nachteil:
  1. nur möglich, wenn das linke Argument vom Klassentyp ist  
( $p * s$  kann Member-Funktion sein,  $s * p$  nicht)
  2. nur möglich, wenn man die Klassendefinition ändern darf

## 15.6 Objekte nachträglich verändern

- bisher: alle Objekte waren “write-once”, d.h. Speicher wurde im Konstruktor initialisiert und war dann unveränderlich  
⇒ Paradigmen der funktionalen Programmierung - “referentielle Integrität”
- prozedurale Programmierung erfordert Möglichkeit, Objekte zu ändern, z.B. um entsprechende Änderungen in der realen Welt widerzuspiegeln
- dazu 3 Möglichkeiten:
  1. Setter-Funktionen (universell nutzbar)

```
class Point {
 ... // wie zuvor
 void setX (double new_x) { // kein const. für Änderung
 (*this).x_ = new_x;
 }

 void setY (double new_y) { // kein const. für Änderung
 (*this).y_ = new_y;
 }

 void set (double new_x, double new_y) {
 (*this).x_ = new_x;
 (*this).y_ = new_y;
 }
};
```

## 2. Index-zugriff, wie bei `std::vector`

```
// wollen:
Point p(2.0, 3.0);
assert(p[0] == 2.0 && p[1] == 3.0); // lesender Zugriff

p[0] = 4.0;
p[1] = 5.0;
assert(p == Point(4.0, 5.0)); // schreibender Zugriff

class Point {
... // wie zuvor
double operator[] (int index) const {
 if (index == 0) {
 return (*this).x_;
 } if (index == 1) {
 return (*this).y_;
 } else {
 // Fehlermeldung
 }
} // lesender Zugriff

double & operator[] (int index) {
 if (index == 0) {
 return (*this).x_;
 } if (index == 1) {
 return (*this).y_;
 } else {
 // Fehlermeldung
 }
} // schreibender Zugriff
};

// Verwendung (Langform):
Point p(2.0, 3.0);
double & x = p[0];
double & y = p[1];
x = 4.0; // ändert indirekt auch die Variablen p.x_, p.y_
y = 5.0;
assert(p == Point(4.0, 5.0));
```

## 3. Zuweisungsoperatoren

```
// wollen:
Point p(2.0, 3.0), q(4.0, 5.0);

p = 1.0;
assert(p == Point(1.0, 1.0));

p = q;
assert(p == Point(4.0, 5.0));

Point & r = q;

class Point {
... // wie zuvor
void operator= (double v) {
 (*this).x_ = v;
}
```

```

 (*this).y_ = v;
 }

 void operator= (Point const & other) {
 (*this).x_ = other.x_;
 (*this).y_ = other.y_;
 } // copy assignment operator
};

```

## Bemerkungen

- implementiert der Programmierer keinen copy assignment Operator, implementiert der Compiler ihn automatisch (wie Kopierkonstruktor): ruft copy assignment für alle Member-Variablen auf
- man implementiert meist:

```

Point & operator= (...) {
 ... // wie zuvor
 return *this;
}

```

Vorteil: man kann Zuweisungen verketteten

- arithmetische Zuweisung:

```

// wollen:
Point p(2.0, 3.0), q(4.0, 5.0);
p += q; // add-assignment
assert(p == Point(6.0, 8.0));

class Point {
 ... // wie zuvor
 Point & operator += (Point const & other) {
 (*this).x_ += other.x_;
 (*this).y_ += other.y_;
 return *this;
 }
};

```

## 16 Klasse: Image

- speichert 2D Bild (analog: Matrix), zunächst nur Graubilder, später Farbbilder
- Beispiel für dynamische Datenstruktur, Größe erst zur Laufzeit bekannt und änderbar
- besteht aus Pixeln ("picture elements"), die mit 2 Indizes x und y angesprochen werden
- Problem: Speicher ist nur 1D  
Lösung: Lege Zeilen hintereinander

```

class Image {
 int width_, height_;
 std::vector<uint16_t> data_;
public:
 Image() //Std-Konstruktor Bildgröße(0,0)
 :width_(0)
 ,height_(0)
 ,data_()
 {}

 Image(unsigned int w, unsigned int u)
 :width_(w)
 ,height_(u)
 ,data_(w*h, 0) // Pixelgröße mit Farbwert schwarz

 int width() const {
 return width_;
 }

 int height() const {
 return height_;
 }

 int size() const { // Gesamtzahl Pixel
 return width_ * height_;
 }

 void resize(unsigned int w, unsigned int h) {
 data_.resize(w*h);
 width_ = w;
 height_ = h;
 }

 uint16_t get(int x, int y) const {
 return data_[x + y*width_];
 }

 void set(int x, int y, uint16_t v) {
 data_[x + y*width_] = v;
 }
};

```

**Zugriff bequemer machen** wünschenswert wäre: 2D Arrays  $\Rightarrow$  verwende stattdessen runde Klammern

```

class Image {
 ... // wie bisher
 uint16_t operator()(int x, int y) const {
 return get(x,y);
 }

 uint16_t & operator()(int x, int y) {
 return data_[x+y*width_];
 }
};

```



```
// jetzt:
uint16_t v = image(1,2);
image(1,2) = 255;
```

### Rückgabe als String

```
std::string to_string (Image const & image) {
 std::string res;
 for (int y=0; y<image.height(); y++) { // iteriert über die Zeilen
 for (int x=0; x<image.width(); x++) { // iteriert über die Spalten
 if (x>0) {
 res += ' ';
 }
 res += std::to_string(image(x,y));
 }
 res += '\n';
 }
 return res;
}
```

### Frage zur Verwendung der Klammern () oder {}?

- vor C++11 gab es nur () oder gar keine Klammern
- Beispiele: Initialisieren mit ()  
Kopierkonstruktor mit ()
- Nachteil: Initialisierung mit Array-Literal wurde nicht unterstützt  
C++11 schließt diese Lücke mittels {}
- Problem: neue Syntax {} muss rückwärtskompatibel mit () sein  
dazu gibt es Regeln:
  1. gibt es einen Konstruktor mit  $k$  Argumenten und einen Array-Konstruktor, dann rufen () den Argument-Konstruktor auf und {} den Array-Konstruktor
  2. gibt es keinen Array-Konstruktor (kein Argument), sind () und {} äquivalent
  3. weitere Regeln: googlen nach “universal construction C++”

## Fehlermeldungen mittels Exceptions

- normalerweise werden Funktionen mit *return* beendet
- tritt in der Funktion ein Fehler auf, kann man den Rückgabebetyp nicht ausrechnen  
⇒ müssen die Funktion anders verlassen
- Exceptions verlassen Funktionen mittels *throw*
  - Argument von *throw*(Rückgabewert) ist ein Exception-Objekt, das den Fehler beschreibt (z.B. Fehlermeldung)

- vereinfachende Exception-Klasse im Header `<stdexcept>`, kann auch eigene definieren  
z.B. `std::runtime_error`

```
class Point {
... // wie bisher
double operator[] (int index) const {
 if (index == 0)
 return x_;
 if (index == 1)
 return y_;
 throw std::runtime_error("Point::operator[];
 index_out_of_range");
}
}
```

- in der aufrufenden Funktion: wirft ein Funktionsaufruf eine Exception, wird standardmäßig die aufrufende Funktion ebenfalls mit "throw" beendet, wobei das Exception-Objekt einfach weitergegeben wird

```
void foo {
 Point p(2,3);
 p[2] = 5; // Exception: index 2 verboten -> foo wird auch beendet
}

int main() {
 foo(); // Exception -> main() wird auch beendet und damit das Programm
 /* alte Compiler geben einfach 'abort' aus, neue die Fehlermeldung
 der Exception
 */
}
```

- um die Exception zu "fangen" und zu behandeln (z.B. Fehler reparieren und retry), braucht man eine try/catch-Umgebung

```
try { // öffnen der Umgebung
 foo(); // Aufruf, der Exception werfen könnte
} ... // weiterer Code, wenn foo() geklappt hat
catch (std::runtime_error & e) { // 2
 std::cerr << "Exception aufgetreten" << e.what() << "\n";
}
```

- Prinzip tritt im try Block eine Exception auf, wird der Block verlassen  
⇒ die Anweisungen hinter dem fehlerhaften Aufruf werden nicht mehr ausgeführt
- folgt ein catch mit passendem Exception-Type, springt die Ausführung in diesen catch-Block  
⇒ es kann beliebig viele catch-Blöcke für verschiedene Exceptions geben
- universal-catch-Block: `catch(std::exception & e)`  
fängt alles auf (genauer alle von `std::exception` abgeleiteten Exceptions)
- Beispiel: warten auf korrekte Benutzereingabe

```

void process_user_input {
 double input = 0.0;
 bool input_valid = false;
 while (!input_valid) {
 try {
 input = get_user_input();
 input_valid = true;
 } catch(std::exception & e) {
 std::cerr << "falsche Eingabe: " << e.what() << "\n Versuche es
 nochmal! \n";
 }
 }
 ... // verarbeite Input
}

```

## Template-Klassen

- wir hatten: Template-Funktionen

```

template <typename T>
T sq (T x) {
 return x*x;
}

```

- wie funktioniert das bei beliebigen Datentypen (z.B. Image-Klasse)
- Beispiel: Image-Klasse soll beliebige Pixeltypen unterstützen, bisher *uint16\_t*, danach *uint8\_t*, float, RGB-Typ
- Vorgehen bei der Templatisierung:
  0. implementiere Klasse und Tests *ohne* Template  
⇒ können nach und nach Templatisieren und jeden Schritt durch Test prüfen
  1. Templatisierung vorbereiten: neue Typnamen einführen mit "typedef OldTypNew-  
TypeName;"
    - (a) in der Testfunktion:

```

void test_image_uint16_t() {
 typedef Image Img;
 Img img(10,20);
 assert(img.width()==10 && img.height()==20);
 assert(img(0,0)==0.0);
 img(0,0) = 255;
 assert(img(0,0)==255);

}

```

- (b) in der Klasse für den Pixeltyp

```

class Image {
public:
 typedef uint16_t PixelType;
}

```

```

private:
 int width_, height_;
 std::vector<PixelType> data_;
public:
 ...
 PixelType operator() (int x, int y) const {
 return data_[x + y*width_];
 }
}

```

⇒ Tests müssen weiterhin funktionieren, weil nur neue Typnamen, gleiche Funktionalität

2. Tests wieder ausführen (müssen genauso funktionieren, wie vorher) - es wurde nichts an der Funktionalität geändert
3. Klasse templatisieren: *typedef* aus a) durch *template*-Deklaration ersetzen

```

template <typename PixelType>
class Point {
 // typedef uint16_t PixelType;
 int width_, height_;
 std::vector<PixelType> data_;
public:
 ...
 PixelType operator() (int x, int y) const {
 return data_[x + y*width_];
 }
} // PixelType als Typname bleibt

```

4. Tests anpassen:

(a)

```

void test_image_uint16_t() {
 typedef Image<uint16_t> Img;
 // nur eine Zeile ändert die Funktionalität
}

```

- (b) freie Funktionen sind noch nicht templatisiert ⇒ vorübergehend auskommentieren, ebenso die zugehörigen Tests
- (c) verbleibende Tests ausführen, müssen wieder funktionieren  
⇒ Image: alle Vorkommen des Typ-Platzhalters *PixelType* durch den Typ *uint16\_t* ersetzt (Template-Instanziierung)
5. eine freie Funktion auswählen, wieder einkommentieren (zusammen mit ihrem Test) und templatisieren (templatisierte Klassen müssen immer mit <> angegeben werden)

```

std::string to_string(Image const & img)
// ->
template <typename PixelType>
std::string to_string(Image<PixelType> const & img)

```

6. Test ausführen, müssen wieder funktionieren
7. zurück zu Schritt 5 bis alle freien Funktionen templatisiert sind
8. weitere Testfunktion für einen anderen PixelTyp schreiben

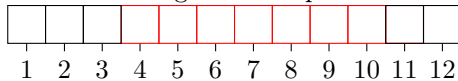
```
void test_image_float() {
 typedef Image<float> Img;
 // Tests jetzt auch mit Dezimalzahlen
}
```

⇒ erst wenn beide Testfunktionen laufen, kann man relativ sicher sein, dass Templatisierung erfolgreich

## Adressen und Zeiger

bisher: Speicherzellen werden über Variablennamen angesprochen (oder mehrere Namen für gleiche Speicherzelle, falls Referenzen)

- Adressen sind Alternative, um Speicherzelle zu identifizieren
- betrachte den gesamten Speicher des Compilers als ein riesiges Array vom Typ "byte"



- der globale Index des Bytes heißt Adresse (hier 3)
- die Adresse des ersten Bytes einer Speicherzelle heißt Adresse einer Speicherzelle  
⇒ kennt man die Adresse einer Speicherzelle und ihren Typ, kann man den Speicher genauso identifizieren wie mit Variablennamen
- Konvention: die Adresse 0 wird nicht verwendet
- die Adresse einer Speicherzelle kann von einer Variablen mit dem Adressoperator erfragt werden (unäre &-Operator)

```
int i = 3;
std::cout << "Adresse von i: " << &i << "\n";
```

- Adressen werden standardmäßig im Hexadezimalsystem angezeigt
- "Zeiger" sind spezielle Variablen, mit denen man Adressen speichern kann

```
int i = 3;
int * ptr_1 = &i; // Initialisierung
int * ptr_2 = ptr_1;
int * ptr_3 = 0; // ungültige Adresse
```

- um vom Zeiger zur Variablen zurückzukommen, benutzt man den Dereferenzierungsoperator *\*ptr*  
Ergebnis: Referenz auf die Speicherzelle, die bei der Adresse beginnt

```

int i = 3;
int * ptr_1 = &i;
int &j = * ptr_1;
j = 5; 77 i ändert sich auch
*ptr_1 = 7; // i und j ändern sich - schreibender Zugriff

// aber:
int k = *ptr_1; // eine neue Variable mit dem selben Wert

```

- Adressoperator und Dereferenzierungsoperator sind Inverse ( $i == *(&i)$ )
- Nullte Argument von Member-Funktionen ist eigentlich ein Zeiger auf die Adresse des aktuellen Objekts (*this*: Zeiger)

## Wozu braucht man Zeiger?

- heute versucht man Zeiger so gut wie möglich in Objekten zu kapseln  
⇒ einfacher, weniger Fehler
- in C gibt es noch keine Referenzen ⇒ benutze Zeiger, um Referenzen zu simulieren

```

// C++
void foo (int &i) {
 i = 5;
}
int j = 3;
foo(j); // j=5

// C
void foo (int *i) {
 *i = 5;
}
int j = 3;
foo(&j);

```

## 16.1 Anwendung der Zeiger

1. Ersatz für Referenzen (besonders in C)
2. für Variablen, die auch ungültig sein können
  - (a) für optionale Argumente

```

int foo (int *a) {
 if (a==0) // ungültige Adresse - Benutzer hat kein Argument ü
 begeben
 {
 ...
 } else {
 ...
 }
}

```

```

schneller:
int foo(int *a =0) { // 0 ist default argument, wenn nichts
 angegeben
 if (a==0) // ungültige Adresse - Benutzer hat kein Argument ü
 bergeben
 {
 ...
 } else {
 ...
 }
 ...
}

```

- (b) wenn eine Funktion fehlschlägt (typisch für C)  
z.B. Öffnen eines Files
3. Variablen, die nacheinander auf verschiedene Speicherzellen zeigen (z.B. Iteratoren in C)
  4. manuelle Speicherverwaltung:
    - Speicherzellen, die mit Schlüsselwort “new” angelegt wurden, gibt der Compiler am Ende der Umgebung nicht automatisch frei
    - wenn nicht mehr benötigt, muss der Programmierer den Speicher mit “delete” manuell freigeben
    - das Resultat von “new” ist Adresse der Speicherzelle
  5. heterogene Container: normale Container (*std::vector<double>*) sind homogen, d.h. alle Elemente haben gleichen Typ  
manchmal braucht man Container, die Objekte verschiedener Typen aufnehmen  
dafür braucht man Zeiger und Vererbung

## heterogene Container

Heterogene Container sind Container mit Elementen verschiedener Typen, in C++ nicht möglich  
Trick: verwende Zeiger auf Interface-Klasse als Elementtyp  
⇒ dynamischer Typ aller Elemente kann unterschiedlich sein  
⇒ Container ist heterogen bezüglich von Aufrufen virtueller Funktionen

```

std::vector<Shape*> drawing;
drawing.push_back(new Line(start1, end1)); // new ist manuelle
 Speicherverwaltung
drawing.push_back(new Circle(center, radius));
drawing.push_back(new Line(start2, end2));
...
Image img(width, height) {
 for (int i = 0, i < drawing.size(); i++) {
 (*drawing[i]).draw(img); // nur Referenz auf i-te Komponente
 }
}

```

- das Composite *shape* enthält ein solches heterogenes Array als Membervariable, seine *draw()*-Funktion kapselt die Schleife
- heterogene Container verändern manuelle Speicherverwaltung  $\Rightarrow$  man darf am Ende nicht das “delete” für die Elemente vergessen  
 $\Rightarrow$  besser: verwende smart-pointer: `std :: vector < std :: shared_ptr < shape >> drawing;`

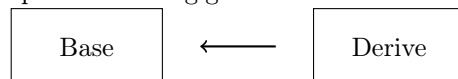
## Kritik an Vererbung

- die Vererbungshierarchie ist fest verdrahtet, aber in verschiedenen Kontexten braucht man eigentlich verschiedene Hierarchien  
z.B. Person-Objekt kann je nach Situation verschiedene Rollen einnehmen (verschiedene Interface-Definitionen)

## Vererbung

war am Anfang der objekt-orientierten Programmierung Lösung für alles

- heute: Hype vorbei, eine Technik unter vielen, sparsam eingesetzt, besonders bei GUI und bei Zeichenprogrammen
- eine (abgeleitete/ Unter-) Klasse kann Funktionalität einer anderen Klasse (Basisklasse) erben, Basisklasse muss dafür nicht sterben
- sprachunabhängige Schreibweise: UML-Diagramme (universal modelling language)



```

class Base {..};

class Derived
: public Base // Vererbung
{..};

```

- die Funktionalität aus der Umgebung Base ist in der Umgebung von Derived sichtbar (wie bei normalen geschachtelten Umgebungen)
- Ausnahmen:
  - \* Konstruktoren und Zuweisungsoperatoren werden nicht mit vererbt
  - \* private Member sind unsichtbar (aber noch vorhanden)
- die in Derived geerbte Funktionalität gehört zur öffentlichen Schnittstelle von Derived  
 $\Rightarrow$  Wiederverwendung der Funktionalität von Base  
Ausnahme: wenn Derived eine neue Member-Variable oder -funktion mit den globalen Namen implementiert, wird die geerbte Variable verdeckt (wie bei normaler Umgebung)
- um Konstruktor von Derived wird der Konstruktor von Base aufgerufen wie ein Konstruktor der Membervariablen



```

class Base {
 int b_;
public:
 Base(int b)
 : b_(b) {}
};

class Derived
: public Base {
 int d_;
public:
 Derived (int b, int d)
 : Base(b)
 , d_(d) {}
};

```

## Anwendungen von Vererbung

1. Spezialisierung
2. Implementationsvererbung
3. Schnittstellenvererbung

**Spezialisierung** beschreibt eine “is -a” Beziehung (Derived is a Base)  
 Basisklasse: untergeordneter Container von Point-Objekten (“Punktwolke”)

```

class Point-Set {
 ... // Funktionen wie Anzahl, Punkte verschieben etc.
};

class Polygon
: public Point-Set
{
 ... // erbt Funktionalität von Point-Set
 // Funktionalität Flächeninhalt, Umfang, Test (Punkt innen/außen)
 // veränderte Funktionalität: zeichnen
};

```

**Implementationsvererbung** wiederverwenden von Funktionalität (sehr umstritten - nur in wenigen Fällen die richtige Lösung)

- wichtig im Zusammenhang mit Spezialisierung z.B. Polygon
- meist bevorzugt man heute Komposition und Delegation

```

class Polynomial {
 std::vector<double> a_;
 /* alle Operationen, die Koeffizienten betreffen,, werden dem Array a_
 weitergeleitet */
};

```

```

 double & operator[] (int i) {
 return a_[i]; // Delegation
 }
};

class Polynomial { // Vererbung
public: std::vector<double>
 std::vector<double> a_;

 // operator[] muss nicht erneut implementiert werden
};

```

Probleme:

- man erbt auch Funktionen, die man nicht möchte, z.B. erase()
- dadurch ist auch der Designer von std::vector in seiner Freiheit beschränkt (er muss auf die Unterklasse Rücksicht nehmen)
- völlig verrückt wäre

```

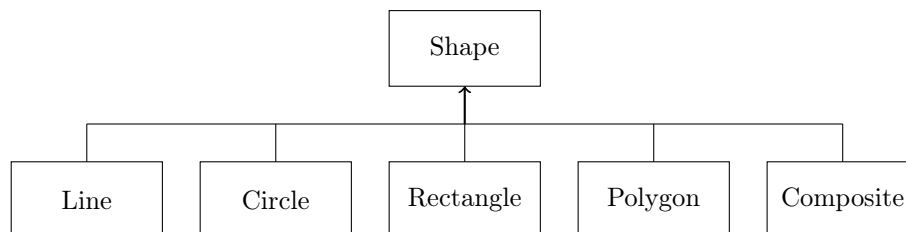
class Point2D {
protected
 double x_, y_;
public:
 ...
};

class Point3D
:public Point2D {
 double z_;
public:
 ...
};

```

## Schnittstellenvererbung

- Basisklasse definiert die öffentliche Schnittstelle, aber implementiert sie nicht
  - abgeleitete Klassen implementieren die Schnittstelle auf verschiedene Art
- Beispiel: Zeichenobjekte



- Interface-Vererbung ist möglich, weil Zeiger/Referenzen auf abgeleitete Klassen in Zeiger/Referenzen der Basisklasse konvertiert werden können

```
Line line(...);
Shape * shape = &line;
```

⇒ bei dieser Konvertierung geht die Typinformation von *line* nicht vollständig verloren, d.h. der Compiler merkt sich, dass die in *Shape* gespeicherte Adresse zu einer Line-Speicherzelle gehört

⇒ *Shape* hat gleichzeitig zwei Typen:

- \* statischer Typ *Shape\**: der Typ der Deklaration
- \* dynamischer Typ *Line*: der ursprüngliche Typ der Speicherzelle

das funktioniert, wenn die Interface-Klasse *Shape* mindestens eine “virtuelle” Member Funktion hat ⇒ dann fügt der Compiler automatisch die notwendige zusätzliche Typinformation in die Speicherzelle ein

```
class Shape {
public:
 virtual void draw (Image & img) const {}

 virtual void drawZoomed (Image & img, double zoom) const {}
 virtual ~Shape() {} // virtueller Destruktor
}
```

statt leerer Implementationen kann man auch “abstrakte Funktionen” definieren, ganz ohne Implementation ⇒ dann ist auch die ganze Klasse abstrakt (Fehlermeldung bei Konstruktoraufwurf) - Zeiger sind erlaubt

es passiert also nicht aus Versehen ein neues Objekt zu erzeugen

jede abgeleitete Klasse muss die virtuellen Funktionen implementieren

```
class Line
:public Shape {
 Point start_, end_;
 uint16_t color_;
public:
 Line(Point s, Point e, uint16_t color)
 :start_(s), end_(e), color_(color) {}

 virtual void draw (Image & img) const {
 ... // nutze Bresenham-Algorithmus
 }
 virtual void drawZoomed (Image & img, double zoom) const {
 ... // zeichne Linie für zoom*start_ bis zoom*end_
 }
 virtual ~line() {}
}
```

- Bedeutung der zwei-Typen-Regel beim Aufruf von *\*Shape.foo()*
  - \* normale Member-Funktionen (nicht virtuell) werden mit dem statischen Typ aufgerufen
    - verwende Implementation der abgeleiteten Klasse

- Nulltes Argument ist Zeiger auf abgeleitete Klasse  
“Polymorphie” - der gleiche Aufruf, kann ganz verschiedene Effekte haben, je nach dem dynamischen Typen der Speicherzelle, auf die Klasse zeigt