

1 Rechenoperationen

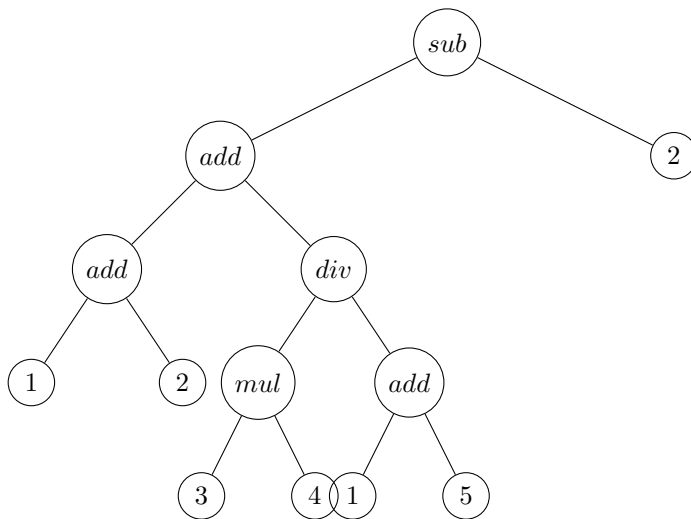
1. Baum besteht aus Knoten (Kreise) und Kanten (Pfeile)
2. Kanten verbinden Knoten mit ihren Kind-Knoten
3. jeder Knoten (außer der Wurzel) hat genau ein Elternteil
4. Knoten ohne Kinder heißen “Blätter“ (leaf-nodes)
5. Teilbaum
 - (a) wähle beliebigen Knoten
 - (b) entferne temporär dessen Eltern-Kante
 - 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 $x_{1/2} \Rightarrow x^2-px+q=0$ mit $p = -\frac{b}{2a}, q = \frac{c}{a} \Rightarrow x_1 = -\frac{b}{2a} + \sqrt{(-\frac{b}{2a})^2 - \frac{c}{a}}$
 \Leftarrow allgemein : $x_{1/2} = p \pm \sqrt{p^2 - q}$
- (c) Präfix:
 $x_1 \leftarrow add(div(div(b, a), -2), sqrt(sub(mul(div(div(b, a), -2), div(div(b, a), -2)), div(c, a))))$
mit Zwischenergebnissen und Infix-Notation: $p \leftarrow b/c/-2$ oder $p \leftarrow -0,5*b/a$ $q \leftarrow c/a$
 $discriminant \leftarrow sqrt(p * P - q)$
 $x_{1/2} \leftarrow p \pm discriminant$

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

```
typ_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 ≡ algebraische Funktionen, iostream ≡ 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
}

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

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 aufgerund, 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:

$$(a/b) * b + \text{rest} = a$$

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

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:

$$z = y - 2001 \quad \Delta = 365 * z + z/4 - z/100 + z/400$$
- 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:

$$w = (w_11y + x - 1) \bmod 7$$

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_we

- kanonische Beispiele: Absolutbetrag, Vorzeichenfunktion

Bedingungen programmieren:

- relationale Operatoren: Vergleich von zwei Argumenten
 $<, >, <=, >=, !=$

- 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);
}
```

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
 $fakultaet(3) \rightarrow fakultaet(2) \rightarrow fakultaet(1) \rightarrow fakultaet(0) \Rightarrow$
 $return\ 3*fakultaet(2) \leftarrow return\ 2*fakultaet(1) \leftarrow return\ 1*fakultaet(0) \leftarrow return\ 1$

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;
}

```

- \Rightarrow Folgen:

- mächtiger, aber ermöglicht völlig neue Bugs \Rightarrow 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

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

```

Anwendung: Wurzelberechnung Ziel: $\text{double sqrt}(\text{double } y)$ 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“) ≈ Schnittstelle des Typs Interface

```

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

```

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 unterstu
```

- 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 (1 < 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)

```

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
⇒ Umgebung jeder Funktion sind “Kinder“ der globalen Umgebung (Ausnahme: Namensräume) ⇒ 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 ⇒ 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 ⇒ Fehlermeldung
- ist der Name in mehreren Umgebungen vorhanden, gilt das zuerst gefundene (Typ, Wert)

⇒ Programmierer muss selbst darauf achten, dass:

1. bei der Suche die gewünschte Speicherzelle gefunden wird ⇒ benutze “sprechende“ Namen
2. der aktuelle Wert der richtige ist ⇒ beachte Reihenfolge der Befehle!

7 Umgebungen

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
 $r = 2$

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*

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 ⇒ 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 (≥ 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 ≥ 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 ⇒ es gibt sie in allen modernen Programmiersprachen, aber die Details sind sehr unterschiedlich
 - Analogie: Zeiger einer Uhr, Cursor in Textverarbeitung
⇒ 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
 \Rightarrow das aktuelle Element ist immer ein Schlüssel/Wert-Paar
(`*iter`).*first* \Rightarrow Schlüssel
(`*iter`).*second* \Rightarrow Wert

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

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