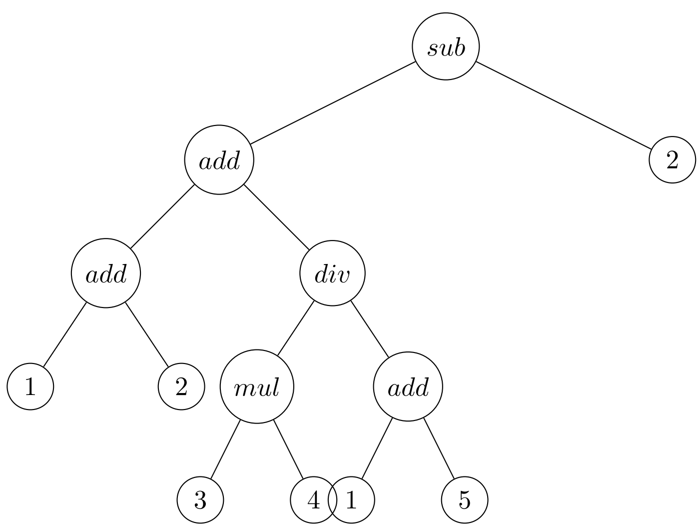
1 Rechenoperationen

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 temorär 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 ⇒ jeder Knoten im Baum bekommt eine Speicherzelle (Maschinensprache: durchnumeriert ; Hochsprache: sprechende Namen)
2. Speicherzellen für die Eingaben *initialisieren* ; Notation: SpZ ← Wert
3. Rechenoperationen in der Reihenfolge des Substitutionsmodells ausführen und in der jeweiligen Speicherzelle speichern ; Notation: SpZ_Ergebnis ← fkt_name SpZ_Arg1 SpZ_Arg2
4. alles in Zahlencode umwandeln
   - Funktionsname ⇒ 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 $x_{1/2} \Rightarrow x^2-px+q = 0 \quad mit \quad p=-\frac{b}{2a}, q=\frac{c}{a} \Rightarrow x_1 = -\frac{b}{2a} + \sqrt{(-\frac{b}{2a}^2 - \frac{c}{a})}$
   $\Leftarrow allgemein : x_{1/2} = p \pm \sqrt{p^2-q}$

   (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

```
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 Funtionen beginnend mit std::. . .
      ii. sind in Module geordnet, z.B. cmath $\widehat{=}$ algebraische Funktionen, iostream $\widehat{=}$ 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;
```

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

Interger-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. Nachkommstellen abschneiden, d.h. Richtung Null runden
3. Gegensatz (z.B. zu Python): floor division $\widehat{=}$ 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;   // aequivalent: a-(b/a)*b
```

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

Bedingungen

- Bei den meisten Algorithmen ist die Reihenfolge der Schritte <u>nicht</u> fix, sondern hängt von den Eingabedaten ab
- Beispiel: Auswahl der Offset $d \to x$ 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 $k! = 1 \cdot 2 \cdot \ldots (k-1) \cdot k$
- 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 ⇒ 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 vorraus (⇒ Memory management notwendig ⇒ später)
  - Entitäten, die sich zeitlich verändern schwer modellierbar, teilweise unnatürlich
- Korrolar: Man kann keine externen Resourcen (z.B. die Console/Drucker, Bildschirm) ansprechen (weil zeitlich veränderlich)
  "keine Seiteneffekte"
- Lösung: Einführung einer Multi-Paradigmensprachen, z.B. Kombination von funktionaler mit prozeduraler Programmierung

5 Prozeduale 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

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	$\varnothing$

- $C++$ beginnt mit der Zählung meist bei 0 "*zero-based*"
- vergisst man Inkrementierung counter = 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: double sqrt (double y) Methode: <u>iterative Verbesserung</u> 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 $x^\star$, sodass $f(x^\star) = 0$ oder $|f(x^\star)| < \epsilon$

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

```
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 abzubrechen:
  - *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 %" ==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
- "Kappselung": die interne Strukter 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* - gekappselt 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 (“concalculate“)

```
std::string s3 = s + "i,k;   // "xyi,k"
std::string s3 = s + s;    // "xyxy"
std::string s3 = "abc" + "def";   // Bug - Literale unterstu
```

- Add-Assignement: 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[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)
```