



Felix Friedrich, Malte Schwerhoff

Informatik

Vorlesung am D-MATH/D-PHYS der ETH Zürich

Herbst 2019

Willkommen

zur Vorlesung Informatik

am MATH/PHYS Department der ETH Zürich.

Ort und Zeit:

Tuesday 13:15 - 15:00, ML D28, ML E12.

Pause 14:00 - 14:15, leichte Verschiebung möglich.

Vorlesungs-Webseite

`http://lec.inf.ethz.ch/ifmp`

Team

Chefassistent
Assistenten

Vytautas Astrauskas
Benjamin Rothenberger
Claire Dick
Edoardo Mazzoni
Enis Ulqinaku
Janet Greutmann
Kevin Kaiwen Zhang
Moritz Schneider
Raul Rao
Sammy Christen
Tobias Klenze

Charlotte Franke
David Sommer
Eliza Wszola
Gaspard Zoss
Jannik Kochert
Manuel Mekkattu
Orhan Saeedi
Reza Sefidgar
Tanja Kaister
Viera Klasovita

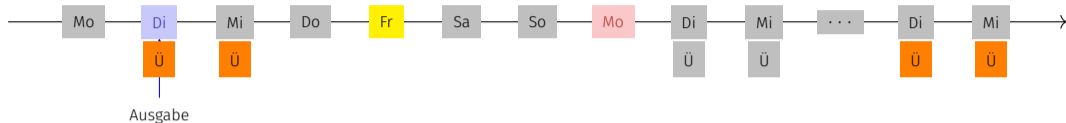
Dozenten

Dr. Malte Schwerhoff / Dr. Felix Friedrich

Einschreibung in Übungsgruppen

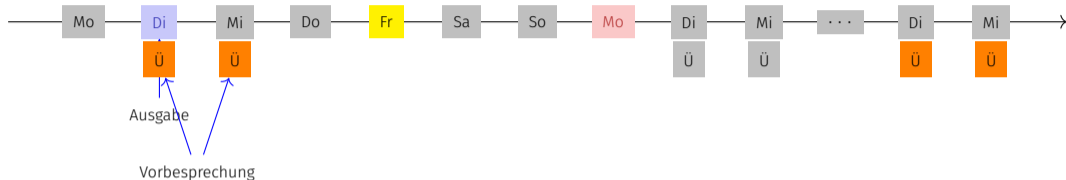
- Gruppeneinteilung selbstständig via Webseite
- Einschreibung breits offen

Ablauf



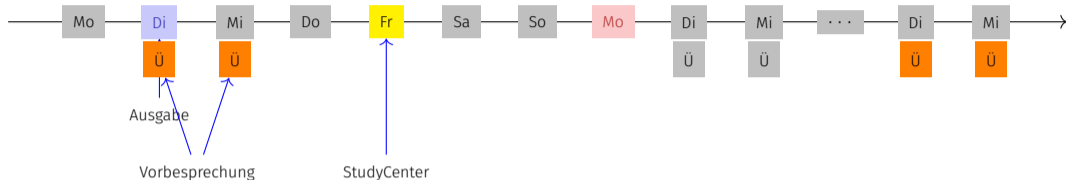
- Übungsblattausgabe zur Vorlesung (online)
- Vorbesprechung in der folgenden Übung (am selben/nächsten Tag)
- StudyCenter (studycenter.ethz.ch)
- Abgabe der Serie spätestens am Tag vor der nächsten Vorlesung (23:59h)
- Nachbesprechung der Serie in der übernächsten Übung. Feedback zu den Abgaben innerhalb einer Woche nach Abgabe.

Ablauf



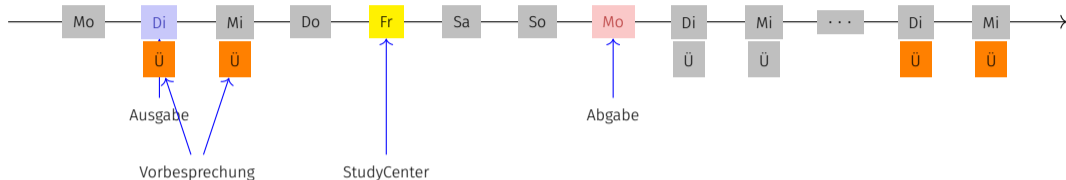
- Übungsblattausgabe zur Vorlesung (online)
- Vorbereitung in der folgenden Übung (am selben/nächsten Tag)
- StudyCenter (studycenter.ethz.ch)
- Abgabe der Serie spätestens am Tag vor der nächsten Vorlesung (23:59h)
- Nachbesprechung der Serie in der übernächsten Übung. Feedback zu den Abgaben innerhalb einer Woche nach Abgabe.

Ablauf



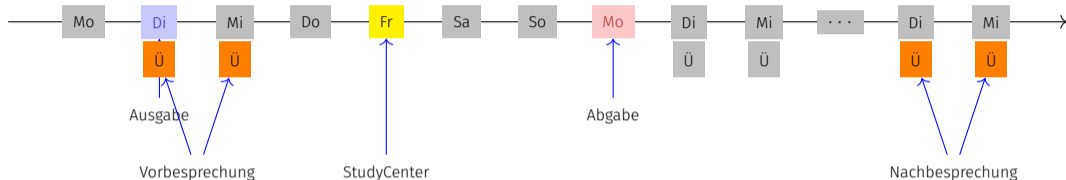
- Übungsblattausgabe zur Vorlesung (online)
- Vorbesprechung in der folgenden Übung (am selben/nächsten Tag)
- StudyCenter (studycenter.ethz.ch)
- Abgabe der Serie spätestens am Tag vor der nächsten Vorlesung (23:59h)
- Nachbesprechung der Serie in der übernächsten Übung. Feedback zu den Abgaben innerhalb einer Woche nach Abgabe.

Ablauf



- Übungsblattausgabe zur Vorlesung (online)
- Vorbereitungen in der folgenden Übung (am selben/nächsten Tag)
- StudyCenter (studycenter.ethz.ch)
- Abgabe der Serie spätestens am Tag vor der nächsten Vorlesung (23:59h)
- Nachbesprechung der Serie in der übernächsten Übung. Feedback zu den Abgaben innerhalb einer Woche nach Abgabe.

Ablauf



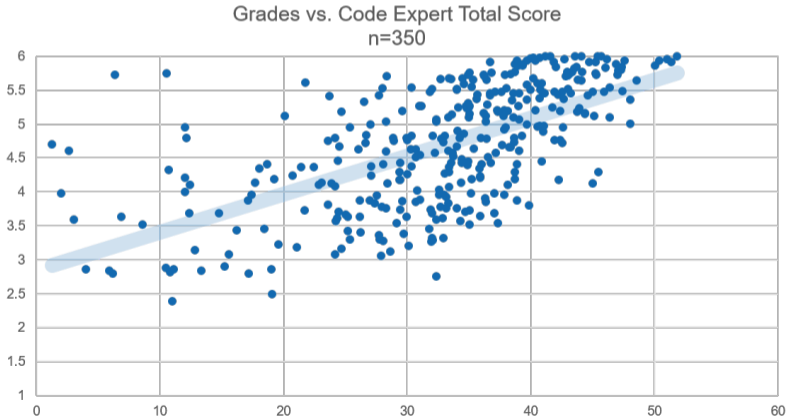
- Übungsblattausgabe zur Vorlesung (online)
- Vorbereitung in der folgenden Übung (am selben/nächsten Tag)
- StudyCenter (studycenter.ethz.ch)
- Abgabe der Serie spätestens am Tag vor der nächsten Vorlesung (23:59h)
- Nachbesprechung der Serie in der übernächsten Übung. Feedback zu den Abgaben innerhalb einer Woche nach Abgabe.

Zu den Übungen

- Bearbeitung der wöchentlichen Uebungsserien ist also freiwillig, wird aber **dringend** empfohlen!

Zu den Übungen

- Bearbeitung der wöchentlichen Uebungsserien ist also freiwillig, wird aber **dringend** empfohlen!



Relevantes für die Prüfung

Prüfungsstoff für die Endprüfung (in der Prüfungssession 2018) schliesst ein

- Vorlesungsinhalt (Vorlesung, Handout) und
- Übungsinhalte (Übungsstunden, Übungsaufgaben).

Relevantes für die Prüfung

Prüfung ist schriftlich.

Es wird sowohl praktisches Wissen (Programmierfähigkeit) als auch theoretisches Wissen (Hintergründe, Systematik) geprüft.

Unser Angebot (VVZ)

- Ihre Programmierübungen werden (halb)automatisch bewertet. Durch Bearbeitung der wöchentlichen Übungsserien kann ein Bonus von maximal 0.25 Notenpunkten erarbeitet werden, der an die Prüfung mitgenommen wird.
- Der Bonus ist proportional zur erreichten Punktzahl von speziell markierten Bonusaufgaben, wobei volle Punktzahl einem Bonus von 0.25 entspricht. Die Zulassung zu speziell markierten Bonusaufgaben hängt von der erfolgreichen Absolvierung anderer Übungsaufgaben ab. Der erreichte Notenbonus verfällt, sobald die Vorlesung neu gelesen wird.

Unser Angebot (Konkret)

- Insgesamt 3 Bonusaufgaben; 2/3 der Punkte reichen für 0.25 Bonuspunkte für die Prüfung
- Sie können also z.B. 2 Bonusaufgaben zu 100% lösen, oder 3 Bonusaufgaben zu je 66%, oder ...
- Bonusaufgaben müssen durch erfolgreich gelöste Übungsserien freigeschaltet (→ Experience Points) werden
- Es müssen wiederum nicht alle Übungsserien vollständig gelöst werden, um eine Bonusaufgabe freizuschalten
- Details: Kurswebseite, Übungsstunden, Online-Übungssystem (Code Expert)

Akademische Lauterkeit

Regel

Sie geben nur eigene Lösungen ab, welche Sie selbst verfasst und verstanden haben.

Wir prüfen das (zum Teil automatisiert) nach und behalten uns insbesondere mündliche Prüfungsgespräche vor.

Sollten Sie zu einem Gespräch eingeladen werden: geraten Sie nicht in Panik. Es gilt primär die Unschuldsvermutung. Wir wollen wissen, ob Sie verstanden haben, was Sie abgegeben haben.

1. Einführung

Informatik: Definition und Geschichte, Algorithmen, Turing Maschine, Höhere Programmiersprachen, Werkzeuge der Programmierung, Das erste C++ Programm und seine syntaktischen und semantischen Bestandteile

Was ist Informatik?

Was ist Informatik?

- Die Wissenschaft der **systematischen Verarbeitung von Informationen**,...

Was ist Informatik?

- Die Wissenschaft der **systematischen Verarbeitung von Informationen**,...
 - ...insbesondere der automatischen Verarbeitung mit Hilfe von Digitalrechnern.
- (Wikipedia, nach dem „Duden Informatik“)

Informatik vs. Computer

Computer science is not about machines, in the same way that astronomy is not about telescopes.

Mike Fellows, US-Informatiker (1991)

Informatik vs. Computer

- Die Informatik beschäftigt sich heute auch mit dem Entwurf von schnellen Computern und Netzwerken...

Informatik vs. Computer

- Die Informatik beschäftigt sich heute auch mit dem Entwurf von schnellen Computern und Netzwerken...
- ...aber nicht als Selbstzweck, sondern zur effizienteren **systematischen Verarbeitung von Informationen.**

Informatik \neq EDV-Kenntnisse

EDV-Kenntnisse: *Anwenderwissen („Computer Literacy“)*

- Umgang mit dem Computer
- Bedienung von Computerprogrammen (für Texterfassung, E-Mail, Präsentationen,...)

Informatik \neq EDV-Kenntnisse

Informatik: *Grundlagenwissen*

- Wie funktioniert ein Computer?
- Wie schreibt man ein Computerprogramm?

Zurück in die Gegenwart: Inhalt dieser Vorlesung

- Systematisches Problemlösen mit Algorithmen und der Programmiersprache C++.
- Also: **nicht nur,**
aber auch Programmierkurs.

Algorithmus: Kernbegriff der Informatik

Algorithmus:

- Handlungsanweisung zur schrittweisen Lösung eines Problems

Algorithmus: Kernbegriff der Informatik

Algorithmus:

- Handlungsanweisung zur schrittweisen Lösung eines Problems
- Ausführung erfordert keine Intelligenz, nur Genauigkeit (sogar Computer können es)

Algorithmus: Kernbegriff der Informatik

Algorithmus:

- Handlungsanweisung zur schrittweisen Lösung eines Problems
- Ausführung erfordert keine Intelligenz, nur Genauigkeit (sogar Computer können es)
- nach *Muhammed al-Chwarizmi*, Autor eines arabischen Rechen-Lehrbuchs (um 825)



“Dixit algorizmi...” (lateinische Übersetzung)

Der älteste nichttriviale Algorithmus

Euklidischer Algorithmus (aus Euklids *Elementen*, 3. Jh. v. Chr.)

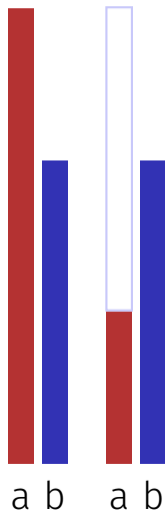


a b

- Eingabe: ganze Zahlen $a > 0, b > 0$
- Ausgabe: ggT von a und b

Der älteste nichttriviale Algorithmus

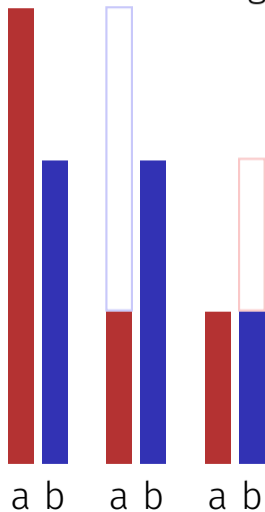
Euklidischer Algorithmus (aus Euklids *Elementen*, 3. Jh. v. Chr.)



- Eingabe: ganze Zahlen $a > 0, b > 0$
- Ausgabe: ggT von a und b

Der älteste nichttriviale Algorithmus

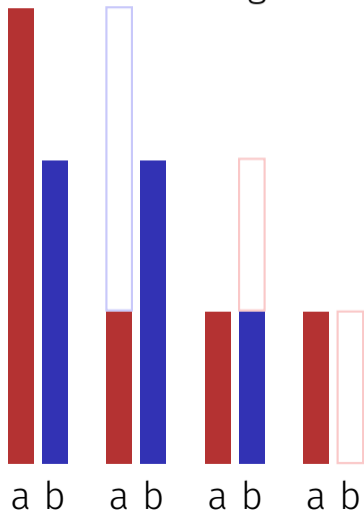
Euklidischer Algorithmus (aus Euklids *Elementen*, 3. Jh. v. Chr.)



- Eingabe: ganze Zahlen $a > 0, b > 0$
- Ausgabe: ggT von a und b

Der älteste nichttriviale Algorithmus

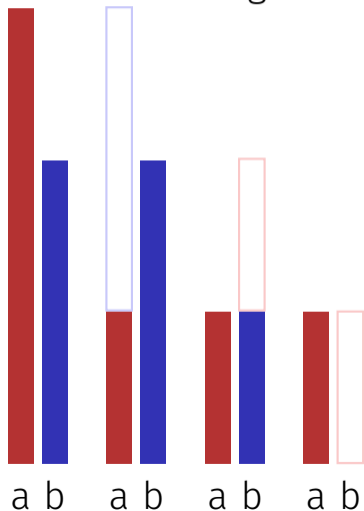
Euklidischer Algorithmus (aus Euklids *Elementen*, 3. Jh. v. Chr.)



- Eingabe: ganze Zahlen $a > 0, b > 0$
- Ausgabe: ggT von a und b

Der älteste nichttriviale Algorithmus

Euklidischer Algorithmus (aus Euklids *Elementen*, 3. Jh. v. Chr.)



- Eingabe: ganze Zahlen $a > 0, b > 0$
- Ausgabe: ggT von a und b

Solange $b \neq 0$

Wenn $a > b$ dann

$$a \leftarrow a - b$$

Sonst:

$$b \leftarrow b - a$$

Ergebnis: a .

Algorithmen: 3 Abstraktionsstufen

1. **Kernidee** (abstrakt):
Die Essenz eines Algorithmus' („Heureka-Moment“)

Algorithmen: 3 Abstraktionsstufen

1. **Kernidee** (abstrakt):
Die Essenz eines Algorithmus' („Heureka-Moment“)
2. **Pseudocode** (semi-detailliert):
Für Menschen gemacht (Bildung, Korrektheit- und Effizienzdiskussionen, Beweise)

Algorithmen: 3 Abstraktionsstufen

1. **Kernidee** (abstrakt):
Die Essenz eines Algorithmus' („Heureka-Moment“)
2. **Pseudocode** (semi-detailliert):
Für Menschen gemacht (Bildung, Korrektheit- und Effizienzdiskussionen, Beweise)
3. **Implementierung** (sehr detailliert):
Für Mensch & Computer gemacht (les- & ausführbar, bestimmte Programmiersprache, verschiedene Implementierungen möglich)

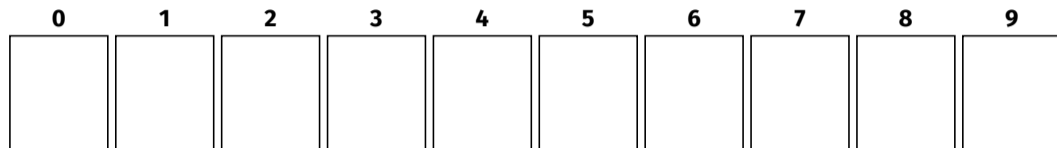
Algorithmen: 3 Abstraktionsstufen

1. **Kernidee** (abstrakt):
Die Essenz eines Algorithmus' („Heureka-Moment“)
2. **Pseudocode** (semi-detailliert):
Für Menschen gemacht (Bildung, Korrektheit- und Effizienzdiskussionen, Beweise)
3. **Implementierung** (sehr detailliert):
Für Mensch & Computer gemacht (les- & ausführbar, bestimmte Programmiersprache, verschiedene Implementierungen möglich)

Euklid: Kernidee und Pseudocode gesehen, Implementierung noch nicht

Euklid in der Box

Speicher



Links

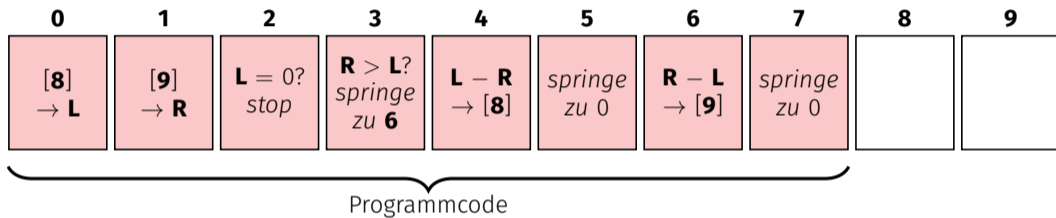
Rechts



Register

Euklid in der Box

Speicher



Links

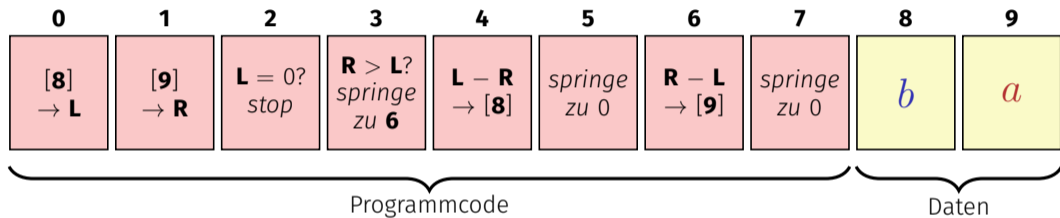
Rechts



Register

Euklid in der Box

Speicher



Links

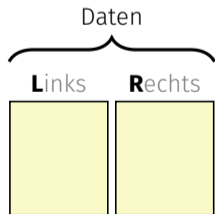
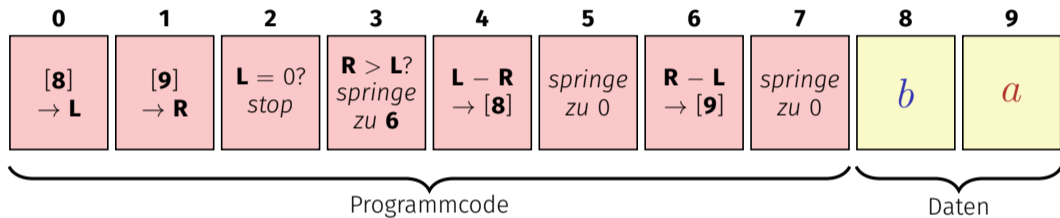
Rechts



Register

Euklid in der Box

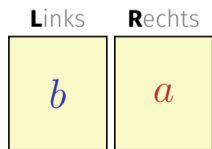
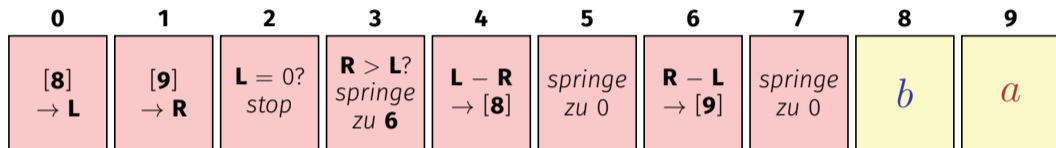
Speicher



Register

Euklid in der Box

Speicher



Solange $b \neq 0$

Wenn $a > b$ dann

$$a \leftarrow a - b$$

Sonst:

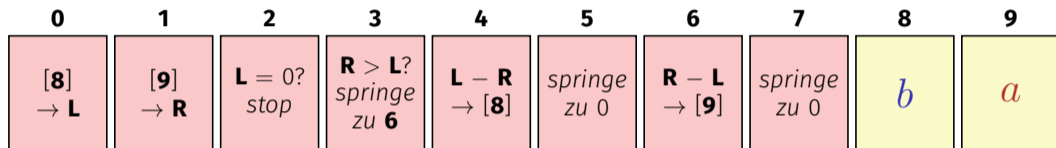
$$b \leftarrow b - a$$

Ergebnis: a .

Register

Euklid in der Box

Speicher



Solange $b \neq 0$

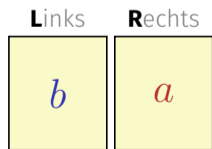
Wenn $a > b$ dann

$$a \leftarrow a - b$$

Sonst:

$$b \leftarrow b - a$$

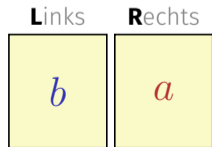
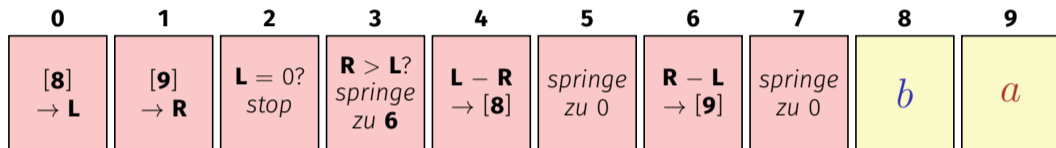
Ergebnis: a .



Register

Euklid in der Box

Speicher



Register

Solange $b \neq 0$

Wenn $a > b$ dann

$$a \leftarrow a - b$$

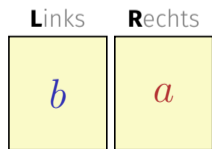
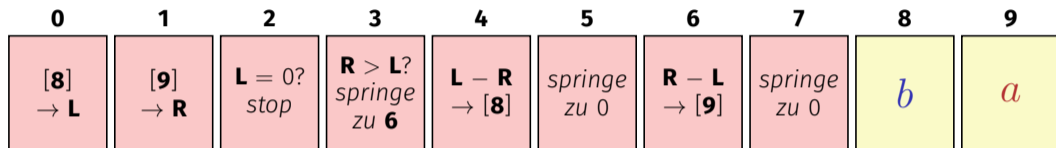
Sonst:

$$b \leftarrow b - a$$

Ergebnis: a .

Euklid in der Box

Speicher



Register

Solange $b \neq 0$

Wenn $a > b$ dann

$$a \leftarrow a - b$$

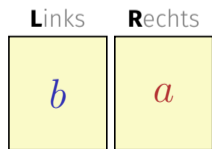
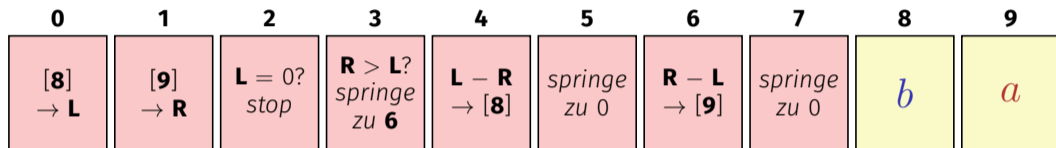
Sonst:

$$b \leftarrow b - a$$

Ergebnis: a .

Euklid in der Box

Speicher

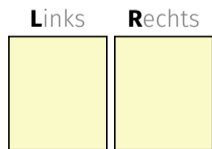
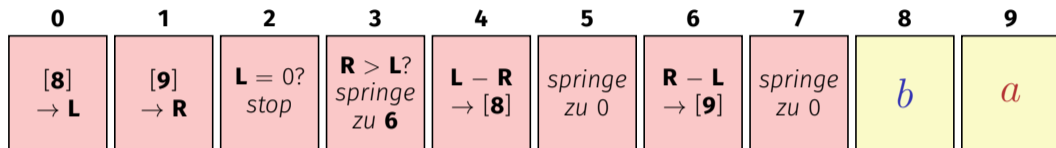


Register

Solange $b \neq 0$
Wenn $a > b$ dann
 $a \leftarrow a - b$
Sonst:
 $b \leftarrow b - a$
Ergebnis: a .

Euklid in der Box

Speicher



Register

Solange $b \neq 0$

Wenn $a > b$ dann

$$a \leftarrow a - b$$

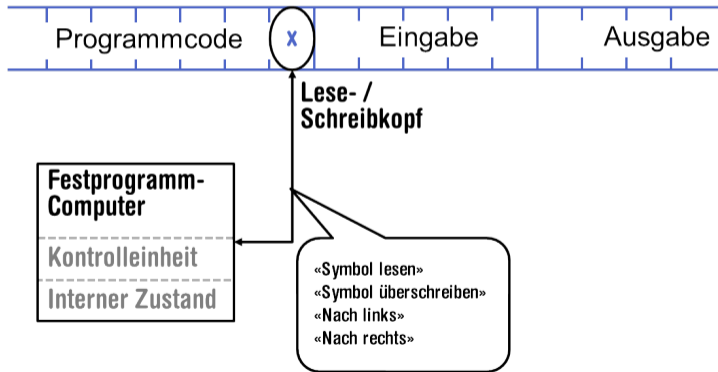
Sonst:

$$b \leftarrow b - a$$

Ergebnis: a .

Computer – Konzept

Eine geniale Idee: Universelle Turingmaschine (Alan Turing, 1936)
Folge von Symbolen auf Ein- und Ausgabeband

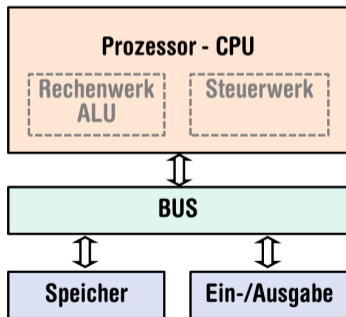


Alan Turing

Computer – Umsetzung

- Z1 – Konrad Zuse (1938)
- ENIAC – John Von Neumann (1945)

Von Neumann Architektur



Konrad Zuse



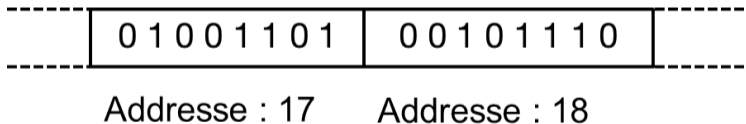
John von Neumann

Speicher für Daten *und* Programm

- Folge von Bits aus $\{0, 1\}$.
- Programmzustand: Werte aller Bits.
- Zusammenfassung von Bits zu Speicherzellen (oft: 8 Bits = 1 Byte).

Speicher für Daten *und* Programm

- Jede Speicherzelle hat eine Adresse.
- Random Access: Zugriffszeit auf Speicherzelle (nahezu) unabhängig von ihrer Adresse.



Programmieren

- Mit Hilfe einer **Programmiersprache** wird dem Computer eine Folge von Befehlen erteilt, damit er genau das macht, was wir wollen.
- Die Folge von Befehlen ist das **(Computer)-Programm**.



The Harvard Computers, Menschliche Berufsrechner, ca.1890

Rechengeschwindigkeit

In der mittleren Zeit, die der Schall von mir zu Ihnen unterwegs ist...

¹Uniprozessor Computer bei 1GHz

Rechengeschwindigkeit

In der mittleren Zeit, die der Schall von mir zu Ihnen unterwegs ist...



arbeitet ein heutiger Desktop-PC mehr als 100

¹Uniprozessor Computer bei 1GHz

Rechengeschwindigkeit

In der mittleren Zeit, die der Schall von mir zu Ihnen unterwegs ist...



30 m $\hat{=}$ mehr als 100.000.000 Instruktionen

arbeitet ein heutiger Desktop-PC mehr als 100 Millionen Instruktionen ab.¹

¹Uniprozessor Computer bei 1GHz

Warum Programmieren?

- Da hätte ich ja gleich Informatik studieren können ...

Warum Programmieren?

- Da hätte ich ja gleich Informatik studieren können ...
- Es gibt doch schon für alles Programme ...

Warum Programmieren?

- Da hätte ich ja gleich Informatik studieren können ...
- Es gibt doch schon für alles Programme ...
- Programmieren interessiert mich nicht ...

Warum Programmieren?

- Da hätte ich ja gleich Informatik studieren können ...
- Es gibt doch schon für alles Programme ...
- Programmieren interessiert mich nicht ...
- Weil Informatik hier leider ein Pflichtfach ist ...

Warum Programmieren?

- Da hätte ich ja gleich Informatik studieren können ...
- Es gibt doch schon für alles Programme ...
- Programmieren interessiert mich nicht ...
- Weil Informatik hier leider ein Pflichtfach ist ...
- ...

Mathematik war früher die Lingua franca der Naturwissenschaften an allen Hochschulen. Und heute ist dies die Informatik.

Lino Guzzella, Präsident der ETH Zürich 2015-2018, NZZ Online, 1.9.2017

(Lino Guzzella ist übrigens nicht Informatiker, sondern Maschineningenieur und Prof. für Thermotronik 😊)

Darum Programmieren!

- Jedes Verständnis moderner Technologie erfordert Wissen über die grundlegende Funktionsweise eines Computers.
- Programmieren (mit dem Werkzeug Computer) wird zu einer Kulturtechnik wie Lesen und Schreiben (mit den Werkzeugen Papier und Bleistift)

Darum Programmieren!

- Jedes Verständnis moderner Technologie erfordert Wissen über die grundlegende Funktionsweise eines Computers.
- Programmieren (mit dem Werkzeug Computer) wird zu einer Kulturtechnik wie Lesen und Schreiben (mit den Werkzeugen Papier und Bleistift)
- Programmieren ist *die* Schnittstelle zwischen Ingenieurwissenschaften und Informatik – der interdisziplinäre Grenzbereich wächst zusehends.

Darum Programmieren!

- Jedes Verständnis moderner Technologie erfordert Wissen über die grundlegende Funktionsweise eines Computers.
- Programmieren (mit dem Werkzeug Computer) wird zu einer Kulturtechnik wie Lesen und Schreiben (mit den Werkzeugen Papier und Bleistift)
- Programmieren ist *die* Schnittstelle zwischen Ingenieurwissenschaften und Informatik – der interdisziplinäre Grenzbereich wächst zusehends.
- Programmieren macht Spass (und ist nützlich)!

Programmiersprachen

- Sprache, die der Computer „versteht“, ist sehr primitiv (Maschinensprache).
- Einfache Operationen müssen in (extrem) viele Einzelschritte aufgeteilt werden.
- Sprache variiert von Computer zu Computer.

Höhere Programmiersprachen

darstellbar als Programmtext, der

- von Menschen *verstanden* werden kann
- vom Computermodell *unabhängig* ist
→ Abstraktion!

Warum C++?

Andere populäre höhere Programmiersprachen: Java, C#, Python, Javascript, Swift, Kotlin, Go,

Warum C++?

Andere populäre höhere Programmiersprachen: Java, C#, Python, Javascript, Swift, Kotlin, Go,

Allgemeiner Konsens

- „Die“ Programmiersprache für Systemprogrammierung: C
- C hat erhebliche Schwächen. Grösste Schwäche: fehlende Typsicherheit.

Warum C++?

Over the years, C++'s greatest strength and its greatest weakness has been its C-Compatibility – B. Stroustrup

Syntax und Semantik

- Programme müssen, wie unsere Sprache, nach gewissen Regeln geformt werden.
 - **Syntax:** Zusammenfügingsregeln für elementare Zeichen (Buchstaben).
 - **Semantik:** Interpretationsregeln für zusammengefügte Zeichen.

Syntax und Semantik

- Programme müssen, wie unsere Sprache, nach gewissen Regeln geformt werden.
 - **Syntax:** Zusammenfügingsregeln für elementare Zeichen (Buchstaben).
 - **Semantik:** Interpretationsregeln für zusammengefügte Zeichen.
- Entsprechende Regeln für ein Computerprogramm sind einfacher, aber auch strenger, denn Computer sind vergleichsweise dumm.

Deutsch vs. C++

Deutsch

Allein sind nicht gefährlich, Rasen ist gefährlich!
(Wikipedia: Mehrdeutigkeit)

C++

```
// computation  
int b = a * a; // b = a2  
b = b * b;    // b = a4
```

Syntax und Semantik von C++

Syntax:

- Wann ist ein Text ein C++-Programm?
- D.h. ist es *grammatikalisch* korrekt?
- → Kann vom Computer überprüft werden

Semantik:

- Was *bedeutet* ein Programm?
- Welchen Algorithmus *implementiert* ein Programm?
- → Braucht menschliches Verständnis

Was braucht es zum Programmieren?

- **Editor:** Programm zum Ändern, Erfassen und Speichern von C++-Programmtext
- **Compiler:** Programm zum Übersetzen des Programmtexts in Maschinsprache

Was braucht es zum Programmieren?

- **Editor:** Programm zum Ändern, Erfassen und Speichern von C++-Programmtext
- **Compiler:** Programm zum Übersetzen des Programmtexts in Maschinsprache
- **Computer:** Gerät zum Ausführen von Programmen in Maschinsprache
- **Betriebssystem:** Programm zur Organisation all dieser Abläufe (Dateiverwaltung, Editor-, Compiler- und Programmaufruf)

Das erste C++ Programm

```
// Program: power8.cpp
// Raise a number to the eighth power.
#include <iostream>
int main() {
    // input
    std::cout << "Compute a^8 for a =? ";
    int a;
    std::cin >> a;
    // computation
    int b = a * a; // b = a^2
    b = b * b;     // b = a^4
    // output b * b, i.e., a^8
    std::cout << a << "^8 = " << b * b << "\n";
    return 0;
}
```

Wichtigste Bestandteile... Anweisungen

```
// Program: power8.cpp
// Raise a number to the eighth power.
#include <iostream>
int main() {
    // input
    std::cout << "Compute a^8 for a =? ";
    int a;
    std::cin >> a; ← Mache etwas (lies a ein)!
    // computation
    int b = a * a; // b = a^2
    b = b * b;    // b = a^4
    // output b * b, i.e., a^8
    std::cout << a << "^8 = " << b * b << "\n";
    return 0;
}
```

```
// Program: power8.cpp
// Raise a number to the eighth power.
#include <iostream>
int main() {
    // input
    std::cout << "Compute a^8 for a =? ";
    int a;
    std::cin >> a;
    // computation
    int b = a * a; // b = a^2 ← Berechne einen Wert (a^2)!
    b = b * b;     // b = a^4
    // output b * b, i.e., a^8
    std::cout << a << "^8 = " << b * b << "\n";
    return 0;
}
```

„Beiwerk“: Kommentare

```
// Program: power8.cpp
// Raise a number to the eighth power.
#include <iostream>
int main() {
    // input
    std::cout << "Compute a^8 for a =? ";
    int a;
    std::cin >> a;
    // computation
    int b = a * a; // b = a^2
    b = b * b;     // b = a^4
    // output b * b, i.e., a^8
    std::cout << a << "^8 = " << b * b << "\n";
    return 0;
}
```


„Beiwerk“: Kommentare

```
// Program: power8.cpp
// Raise a number to the eighth power. ←
#include <iostream>
int main() {
    // input ←
    std::cout << "Compute a^8 for a =? ";
    int a;
    std::cin >> a;
    // computation ←
    int b = a * a; // b = a^2
    b = b * b;     // b = a^4
    // output b * b, i.e., a^8 ←
    std::cout << a << "^8 = " << b * b << "\n";
    return 0;
}
```

Kommentare

Dem Compiler ist's egal...

```
#include <iostream>
int main(){std::cout << "Compute a^8 for a =? ";
int a; std::cin >> a; int b = a * a; b = b * b;
std::cout << a << "^8 = " << b*b << "\n";return 0;}
```

Dem Compiler ist's egal...

```
#include <iostream>
int main(){std::cout << "Compute a^8 for a =? ";
int a; std::cin >> a; int b = a * a; b = b * b;
std::cout << a << "^8 = " << b*b << "\n";return 0;}
```

... uns aber nicht!

„Beiwerk“: Include und Main-Funktion

```
// Program: power8.cpp
// Raise a number to the eighth power.
#include <iostream>
int main() {
    // input
    std::cout << "Compute a^8 for a =? ";
    int a;
    std::cin >> a;
    // computation
    int b = a * a; // b = a^2
    b = b * b;     // b = a^4
    // output b * b, i.e., a^8
    std::cout << a << "^8 = " << b * b << "\n";
    return 0;
}
```

„Beiwerk“: Include und Main-Funktion

```
// Program: power8.cpp
// Raise a number to the eighth power.
#include <iostream> ← Include-Direktive
int main() {
    // input
    std::cout << "Compute a^8 for a =? ";
    int a;
    std::cin >> a;
    // computation
    int b = a * a; // b = a^2
    b = b * b;     // b = a^4
    // output b * b, i.e., a^8
    std::cout << a << "^8 = " << b * b << "\n";
    return 0;
}
```

„Beiwerk“: Include und Main-Funktion

```
// Program: power8.cpp
// Raise a number to the eighth power.
#include <iostream>
int main() { ————— Funktionsdeklaration der main-Funktion
    // input
    std::cout << "Compute a^8 for a =? ";
    int a;
    std::cin >> a;
    // computation
    int b = a * a; // b = a^2
    b = b * b;     // b = a^4
    // output b * b, i.e., a^8
    std::cout << a << "^8 = " << b * b << "\n";
    return 0;
}
```

Anweisungen: Mache etwas!

```
int main() {  
    // input  
    std::cout << "Compute a^8 for a =? ";  
    int a;  
    std::cin >> a;  
    // computation  
    int b = a * a; // b = a^2  
    b = b * b;     // b = a^4  
    // output b * b, i.e., a^8  
    std::cout << a << "^8 = " << b * b << "\n";  
    return 0;  
}
```

Anweisungen: Mache etwas!

```
int main() {  
    // input  
    std::cout << "Compute a^8 for a =? ";  
    int a;  
    std::cin >> a;  
    // computation  
    int b = a * a; // b = a^2  
    b = b * b; // b = a^4  
    // output b * b, i.e., a^8  
    std::cout << a << "^8 = " << b * b << "\n";  
    return 0;  
}
```

Ausdrucksanweisungen

Anweisungen: Mache etwas!

```
int main() {  
    // input  
    std::cout << "Compute a^8 for a =? ";  
    int a;  
    std::cin >> a;  
    // computation  
    int b = a * a; // b = a^2  
    b = b * b;     // b = a^4  
    // output b * b, i.e., a^8  
    std::cout << a << "^8 = " << b * b << "\n";  
    return 0; ← Rückgabeeanweisung  
}
```

Anweisungen – Effekte

```
int main() {  
    // input  
    std::cout << "Compute a^8 for a =? ";  
    int a;  
    std::cin >> a;  
    // computation  
    int b = a * a;  
    b = b * b;  
    // output b * b, i.e., a^8  
    std::cout << a << "^8 = " << b * b << "\n";  
    return 0;  
}
```

Effekt: Ausgabe des Strings Compute

Effekt: Eingabe einer Zahl und Speichern in a

Effekt: Speichern des berechneten Wertes von $a \cdot a$ in b

Effekt: Speichern des berechneten Wertes von $b \cdot b$ in b

Effekt: Rückgabe des Wertes 0

Effekt: Ausgabe des Wertes von a und berechneten Wertes von $b \cdot b$

Anweisungen – Variablendefinitionen

```
int main() {  
    // input  
    std::cout << "Compute a^8 for a =? ";  
    int a; ← Deklarationsanweisungen  
    std::cin >> a;  
    // computation  
    int b = a * a; ← // b = a^2  
    b = b * b;      // b = a^4  
    // output b * b, i.e., a^8  
    std::cout << a << "^8 = " << b * b << "\n";  
    return 0;  
}
```

Typ-
namen

Variablen

- repräsentieren
(wechselnde) Werte
- haben
 - **Name**
 - **Typ**
 - **Wert**
 - **Adresse**

Variablen

- repräsentieren (wechselnde) Werte
- haben
 - **Name**
 - **Typ**
 - **Wert**
 - **Adresse**

`int a;` definiert Variable mit

- Name: `a`
- Typ: `int`
- Wert: (vorerst) undefiniert
- Adresse: durch Compiler (und Linker, Laufzeit) bestimmt

Ausdrücke: Berechne einen Wert!

Ausdrücke

- repräsentieren *Berechnungen*

Ausdrücke: Berechne einen Wert!

Ausdrücke

- repräsentieren *Berechnungen*
- sind entweder **primär** (**b**)

Ausdrücke: Berechne einen Wert!

Ausdrücke

- repräsentieren *Berechnungen*
- sind entweder **primär** (b)
- oder **zusammengesetzt** ($b*b$)...

Ausdrücke: Berechne einen Wert!

Ausdrücke

- repräsentieren *Berechnungen*
- sind entweder **primär** (b)
- oder **zusammengesetzt** ($b*b$)...
- ...aus anderen Ausdrücken, mit Hilfe von **Operatoren**

Ausdrücke: Berechne einen Wert!

Ausdrücke

- repräsentieren *Berechnungen*
- sind entweder **primär** (b)
- oder **zusammengesetzt** ($b*b$)...
- ...aus anderen Ausdrücken, mit Hilfe von **Operatoren**
- haben einen Typ und einen Wert

Ausdrücke: Berechne einen Wert!

Ausdrücke

- repräsentieren *Berechnungen*
- sind entweder **primär** (b)
- oder **zusammengesetzt** ($b*b$)...
- ...aus anderen Ausdrücken, mit Hilfe von **Operatoren**
- haben einen Typ und einen Wert

Analogie: Baukasten

Ausdrücke

```
// input
std::cout << "Compute a^8 for a =? ";
int a;
std::cin >> a;

// computation
int b = a * a; // b = a^2
b = b * b;    // b = a^4

// output b * b, i.e., a^8
std::cout << a << "^8 = " << b * b << ".\n";

return 0;
```

Ausdrücke

```
// input
std::cout << "Compute a^8 for a =? ";
int a;
std::cin >> a;
```

← Variablenname, primärer Ausdruck (+ Name und Adresse)

```
// computation
int b = a * a; // b = a^2
b = b * b; // b = a^4
```

← Variablenname, primärer Ausdruck (+ Name und Adresse)

```
// output b * b, i.e., a^8
std::cout << a << "^8 = " << b * b << ".\n";
```

```
return 0;
```

← Literal, primärer Ausdruck

```
// input  
std::cout << "Compute a^8 for a =? ";  
int a;  
std::cin >> a;
```

Zusammengesetzter Ausdruck

```
// computation  
int b = a * a; // b = a^2  
b = b * b; // b = a^4
```

```
// output b * b, i.e., a^8  
std::cout << a << "^8 = " << b * b << ".\n";  
  
return 0;
```

Zusammengesetzter Ausdruck

```
// input
std::cout << "Compute a^8 for a =? ";
int a;
std::cin >> a;
```

```
// computation
int b = a * a; // b = a^2
```

```
b = b * b ← Zweifach zusammengesetzter Ausdruck
```

```
// output b * b, i.e., a^8
std::cout << a << "^8 = " << b * b << ".\n";
```

↑
return Vierfach zusammengesetzter Ausdruck

Literale

- repräsentieren konstante Werte
- haben festen **Typ** und **Wert**
- sind „syntaktische Werte“

- `0` hat Typ `int`, Wert 0.
- `1.2e5` hat Typ `double`, Wert $1.2 \cdot 10^5$.

L-Werte und R-Werte

```
// input
std::cout << "Compute a^8 for a =? ";
int a;
std::cin >> a;

// computation
int b = a * a; // b = a^2
b = b * b;    // b = a^4

// output b * b, i.e., a^8
std::cout << a << "^8 = " << b * b << ".\n";
return 0;
```

L-Werte und R-Werte

```
// input
std::cout << "Compute a^8 for a =? ";
int a;
std::cin >> a;
```

L-Wert (Ausdruck + Adresse)

```
// computation
int b = a * a; // b = a^2
b = b * b; // b = a^4
```

L-Wert (Ausdruck + Adresse)

```
// output b * b, i.e., a^8
std::cout << a << "^8 = " << b * b << ".\n";
return 0;
```

R-Wert (Ausdruck, der kein L-Wert ist)

L-Werte und R-Werte

```
// input
std::cout << "Compute a^8 for a =? ";
int a;
std::cin >> a;

// computation
int b = a * a; // b = a^2
b = b * b; // b = a^4

// output b * b, i.e., a^8
std::cout << a << "^8 = " << b * b << ".\n";
return 0;
```

The image shows C++ code with annotations. A red box highlights the string "Compute a^8 for a =? " in the first line, with a red arrow pointing to it from the label "R-Wert". Another red box highlights the expression "b * b" in the line "b = b * b;", with a red arrow pointing to it from the label "R-Wert". A third red box highlights the expression "b * b" in the line "std::cout << a << '^8 = ' << b * b << '.\n";", with a red arrow pointing to it from the label "R-Wert".

L-Werte und R-Werte

L-Wert (“**L**inks vom Zuweisungsoperator”)

- Ausdruck mit **Adresse**
- **Wert** ist der Inhalt an der Speicheradresse entsprechend dem Typ des Ausdrucks.

L-Werte und R-Werte

L-Wert (“**L**inks vom Zuweisungsoperator”)

- Ausdruck mit **Adresse**
- **Wert** ist der Inhalt an der Speicheradresse entsprechend dem Typ des Ausdrucks.
- L-Wert kann seinen Wert ändern (z.B. per Zuweisung).

Beispiel: Variablenname

L-Werte und R-Werte

R-Wert (“**R**echts vom Zuweisungsoperator”)

- Ausdruck der kein L-Wert ist

Beispiel: Literal 0

L-Werte und R-Werte

R-Wert (“**R**echts vom Zuweisungsoperator”)

- Ausdruck der kein L-Wert ist
- Jeder L-Wert kann als R-Wert benutzt werden (aber nicht umgekehrt).

Beispiel: Literal 0

L-Werte und R-Werte

R-Wert (“**R**echts vom Zuweisungsoperator”)

- Ausdruck der kein L-Wert ist
- Jeder L-Wert kann als R-Wert benutzt werden (aber nicht umgekehrt). Jedes e-Bike kann als normales Fahrrad benutzt werden, aber nicht umgekehrt.

Beispiel: Literal 0

L-Werte und R-Werte

R-Wert (“**R**echts vom Zuweisungsoperator”)

- Ausdruck der kein L-Wert ist
- Jeder L-Wert kann als R-Wert benutzt werden (aber nicht umgekehrt).
- Ein R-Wert kann seinen Wert *nicht ändern*.

Beispiel: Literal 0

```
// input
std::cout << "Compute a^8 for a =? ";
int a;
std::cin >> a;

// computation
int b = a * a; // b = a^2
b = b * b;     // b = a^4

// output b * b, i.e., a^8
std::cout << a << "^8 = " << b * b << "\n";
return 0;
```

```
// input
std::cout << "Compute a^8 for a=? ";
int a;
std::cin >> a;

// computation
int b = a * a; // b = a^2
b = b * b;    // b = a^4

// output b * b, i.e., a^8
std::cout << a << "^8 = " << b * b << "\n";
return 0;
```

Linker Operand (Ausgabestrom)

Ausgabe-Operator

Rechter Operand (String)

```
// input
std::cout << "Compute a^8 for a =? ";
int a;
std::cin >> a;
// computation
int b = a;
b = b * b;    // b = a^4

// output b * b, i.e., a^8
std::cout << a << "^8 = " << b * b << "\n";
return 0;
```

Rechter Operand (Variablenname)

Eingabe-Operator

Linker Operand (Eingabetrom)

```
// input
std::cout << "Compute a^8 for a =? ";
int a;
std::cin >> a;

// computation
int b = a * a; // b = a^2
b = b * b; // b = a^4
// ou a^8
std::cout << a << "^8 = " << b * b << "\n";
return 0;
```

Zuweisungsoperator

Multiplikationsoperator

2. Ganze Zahlen

Auswertung arithmetischer Ausdrücke, Assoziativität und Präzedenz, arithmetische Operatoren, Wertebereich der Typen **int**, **unsigned int**

Beispiel: power8.cpp

```
int a; // Input
int r; // Result

std::cout << "Compute a^8 for a = ?";
std::cin >> a;

r = a * a; // r = a^2
r = r * r; // r = a^4

std::cout << "a^8 = " << r*r << '\n';
```

Celsius to Fahrenheit

```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.
#include <iostream>

int main() {
    // Input
    std::cout << "Temperature in degrees Celsius =? ";
    int celsius;
    std::cin >> celsius;

    // Computation and output
    std::cout << celsius << " degrees Celsius are "
              << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
    return 0;
}
```


Celsius to Fahrenheit

```
// Program: fahrenheit.cpp
// Convert temperatures from Celsius to Fahrenheit.
#include <iostream>

int main() {
    // Input
    std::cout << "Temperature in degrees Celsius =? ";
    int celsius;
    std::cin >> celsius;

    // Computation and output
    std::cout << celsius << " degrees Celsius are "
              << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
    return 0;
}
```

9 * celsius / 5 + 32

9 * celsius / 5 + 32

- Arithmetischer Ausdruck,

9 * celsius / 5 + 32

9 * celsius / 5 + 32

- Arithmetischer Ausdruck,
- drei Literale, eine Variable, drei Operatorsymbole

9 * celsius / 5 + 32

9 * **celsius** / 5 + 32

- Arithmetischer Ausdruck,
- drei Literale, **eine Variable**, drei Operatorsymbole

9 * celsius / 5 + 32

9 * celsius / 5 + 32

- Arithmetischer Ausdruck,
- drei Literale, eine Variable, drei Operatorsymbole

9 * celsius / 5 + 32

9 * celsius / 5 + 32

- Arithmetischer Ausdruck,
 - drei Literale, eine Variable, drei Operatorsymbole
- Wie ist der Ausdruck geklammert?

Punkt vor Strichrechnung

`9 * celsius / 5 + 32`

bedeutet

`(9 * celsius / 5) + 32`

Regel 1: Präzedenz

Multiplikative Operatoren ($*$, $/$, $\%$) haben höhere Präzedenz („binden stärker“) als additive Operatoren ($+$, $-$)

Assoziativität

Von links nach rechts

`9 * celsius / 5 + 32`

bedeutet

`((9 * celsius) / 5) + 32`

Assoziativität

Regel 2: Assoziativität

Arithmetische Operatoren ($*$, $/$, $\%$, $+$, $-$) sind linksassoziativ: bei gleicher Präzedenz erfolgt Auswertung von links nach rechts

Stelligkeit

Vorzeichen

$-3 - 4$

bedeutet

$(-3) - 4$

Stelligkeit

Regel 3: Stelligkeit

Unäre Operatoren $+$, $-$ vor binären $+$, $-$.

Klammerung

Jeder Ausdruck kann mit Hilfe der

- Assoziativitäten
- Präzedenzen
- Stelligkeiten

der beteiligten Operatoren eindeutig geklammert werden.

Ausdrucksbäume

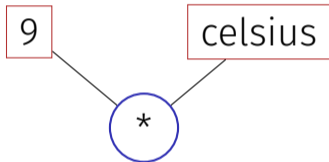
Klammerung ergibt Ausdrucksbaum

`9 * celsius / 5 + 32`

Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

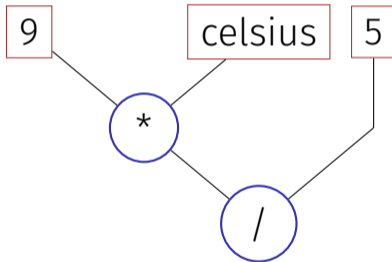
`(9 * celsius) / 5 + 32`



Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

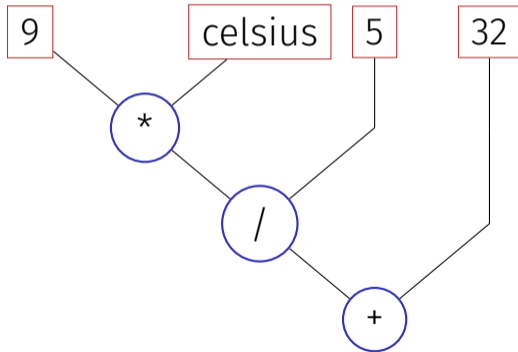
`((9 * celsius) / 5) + 32`



Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

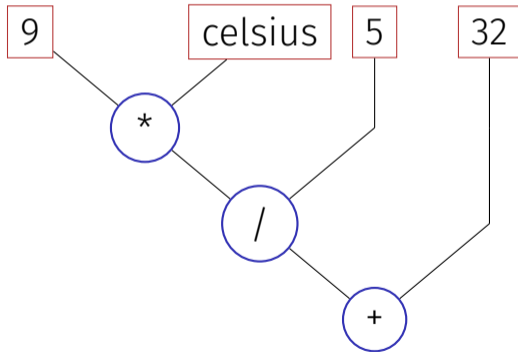
`((9 * celsius) / 5) + 32)`



Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

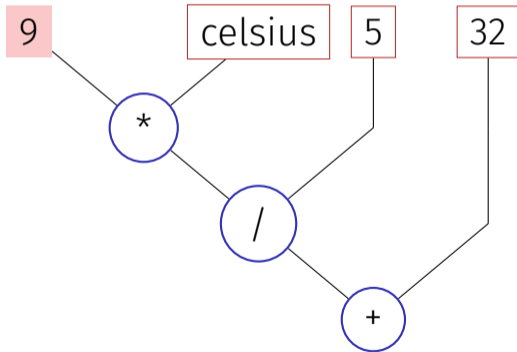
9 * celsius / 5 + 32



Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

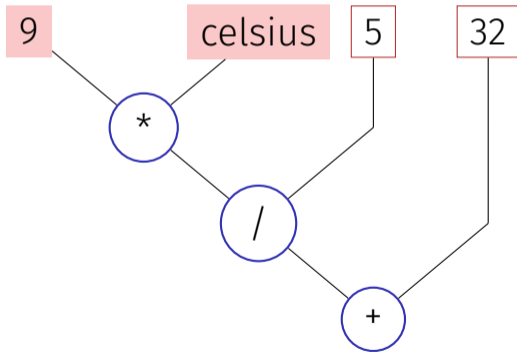
9 * celsius / 5 + 32



Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

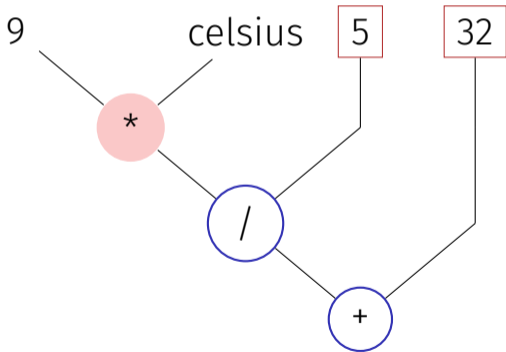
9 * celsius / 5 + 32



Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

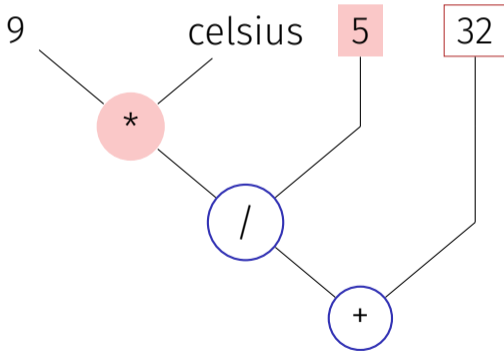
9 * celsius / 5 + 32



Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

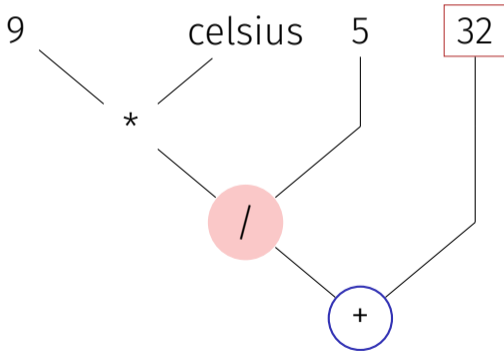
9 * celsius / 5 + 32



Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

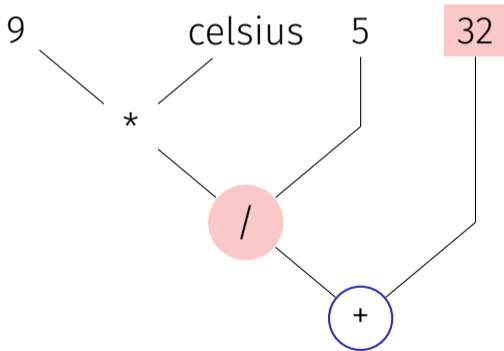
9 * celsius / 5 + 32



Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

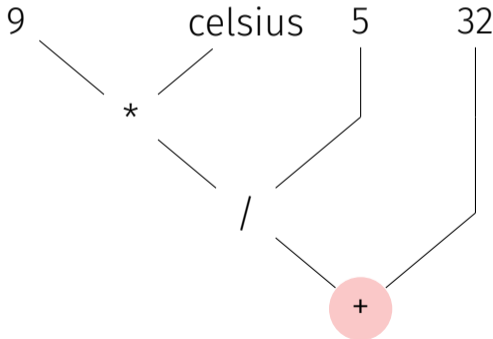
9 * celsius / 5 + 32



Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

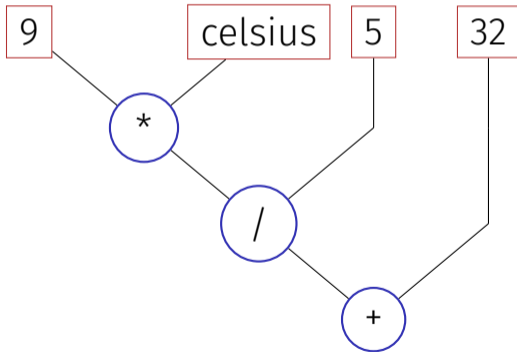
9 * celsius / 5 + 32



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

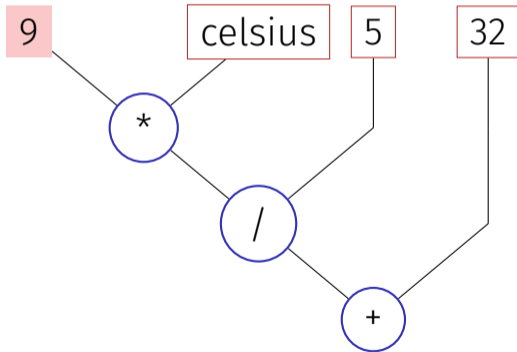
$$9 * \text{celsius} / 5 + 32$$



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

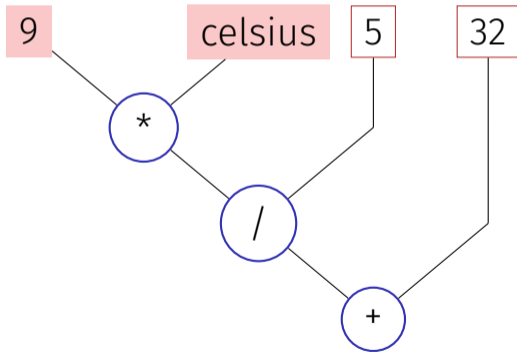
$$9 * \text{celsius} / 5 + 32$$



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

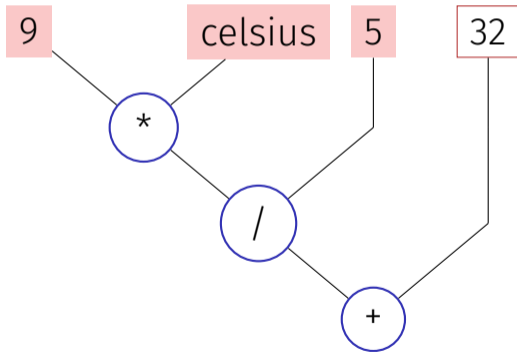
$$9 * \text{celsius} / 5 + 32$$



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

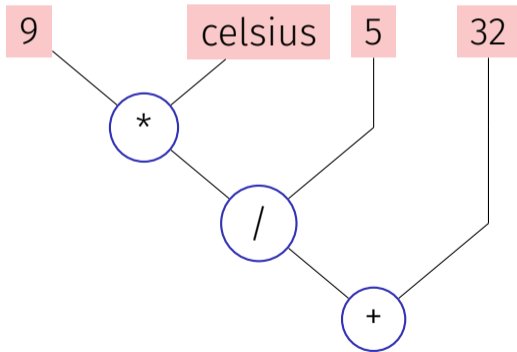
$$9 * \text{celsius} / 5 + 32$$



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

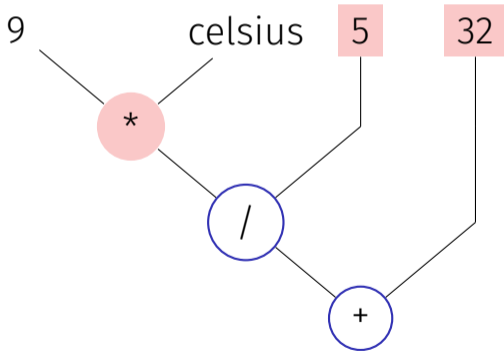
$$9 * \text{celsius} / 5 + 32$$



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

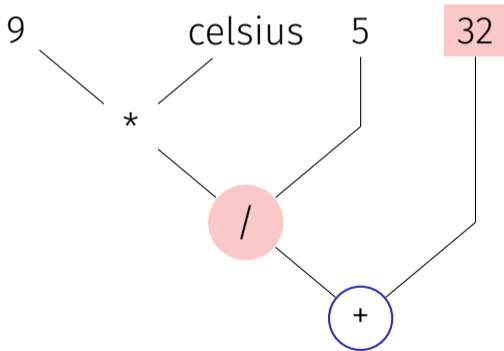
$$9 * \text{celsius} / 5 + 32$$



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

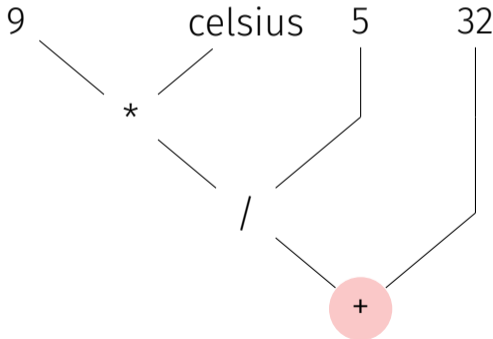
$$9 * \text{celsius} / 5 + 32$$



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

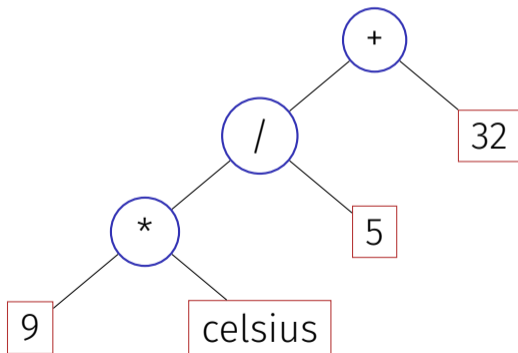
$$9 * \text{celsius} / 5 + 32$$



Ausdrucksbäume – Notation

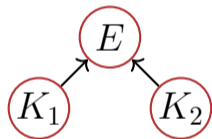
Üblichere Notation: Wurzel oben

9 * celsius / 5 + 32



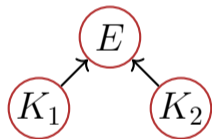
Auswertungsreihenfolge – formaler

Gültige Reihenfolge: Jeder Knoten wird erst **nach** seinen Kindern ausgewertet.



Auswertungsreihenfolge – formaler

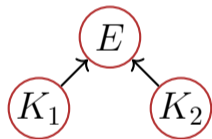
Gültige Reihenfolge: Jeder Knoten wird erst **nach** seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

Auswertungsreihenfolge – formaler

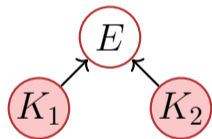
Gültige Reihenfolge: Jeder Knoten wird erst **nach** seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

Auswertungsreihenfolge – formaler

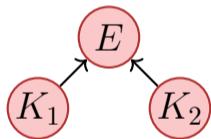
Gültige Reihenfolge: Jeder Knoten wird erst **nach** seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

Auswertungsreihenfolge – formaler

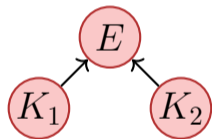
Gültige Reihenfolge: Jeder Knoten wird erst **nach** seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

Auswertungsreihenfolge – formaler

Gültige Reihenfolge: Jeder Knoten wird erst **nach** seinen Kindern ausgewertet.

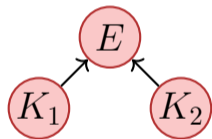


C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

- „Guter Ausdruck“: jede gültige Reihenfolge führt zum gleichen Ergebnis.

Auswertungsreihenfolge – formaler

Gültige Reihenfolge: Jeder Knoten wird erst **nach** seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

- „Guter Ausdruck“: jede gültige Reihenfolge führt zum gleichen Ergebnis.
- Beispiel eines „schlechten Ausdrucks“: $a*(a=2)$

Auswertungsreihenfolge

Richtlinie

Vermeide das Verändern von Variablen, welche im selben Ausdruck noch einmal verwendet werden!

Arithmetische Operatoren

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Unäres +	+	1	16	rechts
Negation	-	1	16	rechts
Multiplikation	*	2	14	links
Division	/	2	14	links
Modulo	%	2	14	links
Addition	+	2	13	links
Subtraktion	-	2	13	links

Einschub: Zuweisungsausdruck – nun genauer

- Bereits bekannt: $\mathbf{a} = \mathbf{b}$ bedeutet Zuweisung von \mathbf{b} (R-Wert) an \mathbf{a} (L-Wert). Rückgabe: L-Wert.

Einschub: Zuweisungsausdruck – nun genauer

- Bereits bekannt: $\mathbf{a} = \mathbf{b}$ bedeutet Zuweisung von \mathbf{b} (R-Wert) an \mathbf{a} (L-Wert). Rückgabe: L-Wert.
- Was bedeutet $\mathbf{a} = \mathbf{b} = \mathbf{c}$?

Einschub: Zuweisungsausdruck – nun genauer

- Bereits bekannt: $\mathbf{a} = \mathbf{b}$ bedeutet Zuweisung von \mathbf{b} (R-Wert) an \mathbf{a} (L-Wert). Rückgabe: L-Wert.
- Was bedeutet $\mathbf{a} = \mathbf{b} = \mathbf{c}$?
- Antwort: Zuweisung rechtsassoziativ, also

Einschub: Zuweisungsausdruck – nun genauer

- Bereits bekannt: $\mathbf{a = b}$ bedeutet Zuweisung von \mathbf{b} (R-Wert) an \mathbf{a} (L-Wert). Rückgabe: L-Wert.
- Was bedeutet $\mathbf{a = b = c}$?
- Antwort: Zuweisung rechtsassoziativ, also

$$\mathbf{a = b = c} \quad \iff \quad \mathbf{a = (b = c)}$$

Mehrfachzuweisung: $\mathbf{a = b = 0} \implies \mathbf{b=0; a=0}$

Division

- Operator / realisiert ganzzahlige Division

5 / 2 hat Wert 2

Division

- Operator / realisiert ganzzahlige Division

```
5 / 2 hat Wert 2
```

- In `fahrenheit.cpp`

```
9 * celsius / 5 + 32
```

```
15 degrees Celsius are 59 degrees Fahrenheit
```

Division

- Operator `/` realisiert ganzzahlige Division

```
5 / 2 hat Wert 2
```

- In `fahrenheit.cpp`

```
9 * celsius / 5 + 32
```

```
15 degrees Celsius are 59 degrees Fahrenheit
```

Division

- Operator / realisiert ganzzahlige Division

```
5 / 2 hat Wert 2
```

- In `fahrenheit.cpp`

```
9 * celsius / 5 + 32
```

```
15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent...

```
9 / 5 * celsius + 32
```

Division

- Operator / realisiert ganzzahlige Division

```
5 / 2 hat Wert 2
```

- In `fahrenheit.cpp`

```
9 * celsius / 5 + 32
```

```
15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent...

```
1 * celsius + 32
```

Division

- Operator / realisiert ganzzahlige Division

```
5 / 2 hat Wert 2
```

- In `fahrenheit.cpp`

```
9 * celsius / 5 + 32
```

```
15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent...

```
15 + 32
```

Division

- Operator / realisiert ganzzahlige Division

```
5 / 2 hat Wert 2
```

- In `fahrenheit.cpp`

```
9 * celsius / 5 + 32
```

```
15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent...

```
47
```

Division

- Operator / realisiert ganzzahlige Division

```
5 / 2 hat Wert 2
```

- In `fahrenheit.cpp`

```
9 * celsius / 5 + 32
```

```
15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematisch äquivalent...aber nicht in C++!

```
9 / 5 * celsius + 32
```

```
15 degrees Celsius are 47 degrees Fahrenheit
```

Präzisionsverlust

Richtlinie

- Auf möglichen Präzisionsverlust achten
- Potentiell verlustbehaftete Operationen möglichst spät durchführen, um „Fehlerescalation“ zu vermeiden

Division und Modulo

- Modulo-Operator berechnet Rest der ganzzahligen Division

$5 / 2$ hat Wert 2, $5 \% 2$ hat Wert 1.

- Es gilt

$$(-a)/b == -(a/b)$$

Division und Modulo

- Modulo-Operator berechnet Rest der ganzzahligen Division

$5 / 2$ hat Wert 2, $5 \% 2$ hat Wert 1.

- Es gilt

$(-a) / b == -(a / b)$

- Es gilt auch:

$(a / b) * b + a \% b$ hat den Wert von **a**.

Division und Modulo

- Modulo-Operator berechnet Rest der ganzzahligen Division

$5 / 2$ hat Wert 2, $5 \% 2$ hat Wert 1.

- Es gilt

$$(-a) / b == -(a / b)$$

- Es gilt auch:

$$(a / b) * b + a \% b \text{ hat den Wert von } a.$$

- Daraus lässt sich herleiten, welche Ergebnisse Division und Modulo mit negativen Zahlen ergeben (müssen)

Inkrement und Dekrement

- Erhöhen / Erniedrigen einer Zahl um 1 ist eine häufige Operation
- geht für einen L-Wert so:

```
expr = expr + 1.
```

Inkrement und Dekrement

```
expr = expr + 1.
```

Nachteile

- relativ lang

Inkrement und Dekrement

```
expr = expr + 1.
```

Nachteile

- relativ lang
- **expr** wird zweimal ausgewertet
 - Später: L-wertige Ausdrücke deren Auswertung „teuer“ ist

Inkrement und Dekrement

```
expr = expr + 1.
```

Nachteile

- relativ lang
- **expr** wird zweimal ausgewertet
 - Später: L-wertige Ausdrücke deren Auswertung „teuer“ ist
 - **expr** könnte einen Effekt haben (aber sollte nicht, siehe Richtlinie)

In-/Dekrement Operatoren

Post-Inkrement

`expr++`

Wert von **expr** wird um 1 erhöht, der **alte** Wert von **expr** wird (als R-Wert) zurückgegeben

In-/Dekrement Operatoren

Prä-Inkrement

`++expr`

Wert von **expr** wird um 1 erhöht, der **neue** Wert von **expr** wird (als L-Wert) zurückgegeben

In-/Dekrement Operatoren

Post-Dekrement

`expr--`

Wert von **expr** wird um 1 verringert, der **alte** Wert von **expr** wird (als R-Wert) zurückgegeben

In-/Dekrement Operatoren

Prä-Dekrement

--**expr**

Wert von **expr** wird um 1 verringert, der **neue** Wert von **expr** wird (als L-Wert) zurückgegeben

In-/Dekrement Operatoren

```
int a = 7;  
std::cout << ++a << "\n";  
std::cout << a++ << "\n";  
std::cout << a << "\n";
```

In-/Dekrement Operatoren

```
int a = 7;  
std::cout << ++a << "\n"; // 8  
std::cout << a++ << "\n";  
std::cout << a << "\n";
```

In-/Dekrement Operatoren

```
int a = 7;  
std::cout << ++a << "\n"; // 8  
std::cout << a++ << "\n"; // 8  
std::cout << a << "\n";
```

In-/Dekrement Operatoren

```
int a = 7;  
std::cout << ++a << "\n"; // 8  
std::cout << a++ << "\n"; // 8  
std::cout << a << "\n"; // 9
```

Arithmetische Zuweisungen

`a += b`

\Leftrightarrow

`a = a + b`

Arithmetische Zuweisungen

$$\begin{aligned} a \ += \ b \\ \Leftrightarrow \\ a \ = \ a \ + \ b \end{aligned}$$

Analog für $-$, $*$, $/$ und $\%$

Binäre Zahlendarstellungen

Binäre Darstellung (Bits aus $\{0, 1\}$)

$$b_n b_{n-1} \dots b_1 b_0$$

entspricht der Zahl $b_n \cdot 2^n + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0$

Binäre Zahlendarstellungen

Binäre Darstellung (Bits aus $\{0, 1\}$)

$$b_n b_{n-1} \dots b_1 b_0$$

entspricht der Zahl $b_n \cdot 2^n + \dots + b_1 \cdot 2 + b_0$

Binäre Zahlendarstellungen

Binäre Darstellung (Bits aus $\{0, 1\}$)

$$b_n b_{n-1} \dots b_1 b_0$$

entspricht der Zahl $b_n \cdot 2^n + \dots + b_1 \cdot 2 + b_0$

101011

Binäre Zahlendarstellungen

Binäre Darstellung (Bits aus $\{0, 1\}$)

$$b_n b_{n-1} \dots b_1 b_0$$

entspricht der Zahl $b_n \cdot 2^n + \dots + b_1 \cdot 2 + b_0$

101011 entspricht **32+8+2+1**.

Binäre Zahlendarstellungen

Binäre Darstellung (Bits aus $\{0, 1\}$)

$$b_n b_{n-1} \dots b_1 b_0$$

entspricht der Zahl $b_n \cdot 2^n + \dots + b_1 \cdot 2 + b_0$

101011 entspricht **43**.

Binäre Zahlendarstellungen

Binäre Darstellung (Bits aus $\{0, 1\}$)

$$b_n b_{n-1} \dots b_1 b_0$$

entspricht der Zahl $b_n \cdot 2^n + \dots + b_1 \cdot 2 + b_0$

101011 entspricht **43**.

Niedrigstes Bit, Least Significant Bit (LSB)

Höchstes Bit, Most Significant Bit (MSB)

Hexadezimale Zahlen

Zahlen zur Basis 16. Darstellung

$$h_n h_{n-1} \dots h_1 h_0$$

entspricht der Zahl

$$h_n \cdot 16^n + \dots + h_1 \cdot 16 + h_0.$$

Schreibweise in C++: vorangestelltes **0x**

0xff entspricht **255**.

Hex Nibbles

hex	bin	dec
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
a	1010	10
b	1011	11
c	1100	12
d	1101	13
e	1110	14
f	1111	15

Wozu Hexadezimalzahlen?

- Ein Hex-Nibble entspricht **genau** 4 Bits.

Wozu Hexadezimalzahlen?

- Ein Hex-Nibble entspricht **genau** 4 Bits.
- „Kompakte Darstellung von Binärzahlen“.

Beispiel: Hex-Farben

#00FF00

r g b

Beispiel: Hex-Farben

#FFFFFF00

r g b

Beispiel: Hex-Farben

#808080

r g b

Beispiel: Hex-Farben

#FF0050



r g b

Wertebereich des Typs int

```
// Output the smallest and the largest value of type int.
#include <iostream>
#include <limits>

int main() {
    std::cout << "Minimum int value is "
              << std::numeric_limits<int>::min() << ".\n"
              << "Maximum int value is "
              << std::numeric_limits<int>::max() << ".\n";
    return 0;
}
```

Wertebereich des Typs int

```
// Output the smallest and the largest value of type int.
#include <iostream>
#include <limits>

int main() {
    std::cout << "Minimum int value is "
              << std::numeric_limits<int>::min() << ".\n"
              << "Maximum int value is "
              << std::numeric_limits<int>::max() << ".\n";
    return 0;
}
```

```
Minimum int value is -2147483648.
Maximum int value is 2147483647.
```


Wertebereich des Typs int

```
// Output the smallest and the largest value of type int.
#include <iostream>
#include <limits>

int main() {
    std::cout << "Minimum int value is "
               << std::numeric_limits<int>::min() << ".\n"
               << "Maximum int value is "
               << std::numeric_limits<int>::max() << ".\n";
    return 0;
}
```

Minimum int value is -2147483648.
Maximum int value is 2147483647.
Woher kommen diese Zahlen?

Wertebereich des Typs `int`

- Repräsentation mit B Bits. Wertebereich

$$\{-2^{B-1}, \dots, -1, 0, 1, \dots, 2^{B-1} - 2, 2^{B-1} - 1\}$$

Wertebereich des Typs `int`

- Repräsentation mit B Bits. Wertebereich

$$\{-2^{B-1}, \dots, -1, 0, 1, \dots, 2^{B-1} - 2, 2^{B-1} - 1\}$$

- Auf den meisten Plattformen $B = 32$

Wertebereich des Typs `int`

- Repräsentation mit B Bits. Wertebereich

$$\{-2^{B-1}, \dots, -1, 0, 1, \dots, 2^{B-1} - 2, 2^{B-1} - 1\}$$

- Für den Typ `int` garantiert C++ $B \geq 16$

Überlauf und Unterlauf

- Arithmetische Operationen (+, -, *) können aus dem Wertebereich herausführen.
- Ergebnisse können inkorrekt sein.

```
power8.cpp: 158 = -1732076671
```

- Es gibt **keine Fehlermeldung!**

Der Typ `unsigned int`

- Wertebereich

$$\{0, 1, \dots, 2^B - 1\}$$

- Alle arithmetischen Operationen gibt es auch für `unsigned int`.
- Literale: `1u`, `17u` ...

Gemischte Ausdrücke

- Operatoren können Operanden verschiedener Typen haben (z.B. `int` und `unsigned int`).

```
17 + 17u
```

- Solche gemischten Ausdrücke sind vom „allgemeineren“ Typ `unsigned int`.
- `int`-Operanden werden **konvertiert** nach `unsigned int`.

Konversion

int Wert	Vorzeichen	unsigned int Wert
x	≥ 0	x
x	< 0	$x + 2^B$

Konversion

int Wert	Vorzeichen	unsigned int Wert
x	≥ 0	x
x	< 0	$x + 2^B$

Dank cleverer Repräsentation (Zweierkomplement) muss intern gar nicht addiert werden

Zahlen mit Vorzeichen

Hinweis: Die verbleibenden Folien zur vorzeichenbehafteten Zahlendarstellung, dem Rechnen mit Binärzahlen sowie der Zweierkomplementdarstellung sind *nicht* klausurrelevant

Vorzeichenbehaftete Zahlendarstellung

- Soweit klar (hoffentlich): Binäre Zahlendarstellung ohne Vorzeichen, z.B.

$$[b_{31}b_{30} \dots b_0]_u \cong b_{31} \cdot 2^{31} + b_{30} \cdot 2^{30} + \dots + b_0$$

- Suche möglichst konsistente Lösung

Die Darstellung mit Vorzeichen sollte möglichst viel mit der vorzeichenlosen Lösung „gemein haben“. Positive Zahlen sollten sich in beiden Systemen algorithmisch möglichst gleich verhalten.

Rechnen mit Binärzahlen (4 Stellen)

Einfache Addition

$$\begin{array}{r} 2 \\ +3 \\ \hline 5 \end{array}$$

$$\begin{array}{r} 0010 \\ +0011 \\ \hline 0101_2 = 5_{10} \end{array}$$

Rechnen mit Binärzahlen (4 Stellen)

Einfache Subtraktion

$$\begin{array}{r} 5 \\ -3 \\ \hline 2 \end{array}$$

$$\begin{array}{r} 0101 \\ -0011 \\ \hline 0010_2 = 2_{10} \end{array}$$

Rechnen mit Binärzahlen (4 Stellen)

Addition mit Überlauf

$$\begin{array}{r} 7 \\ +10 \\ \hline 17 \end{array} \quad \begin{array}{r} 0111 \\ +1010 \\ \hline (1)0001_2 = 1_{10}(= 17 \bmod 16) \end{array}$$


Rechnen mit Binärzahlen (4 Stellen)

Subtraktion mit Unterlauf

$$\begin{array}{r} 5 \\ +(-10) \\ \hline -5 \end{array} \quad \begin{array}{r} 0101 \\ 1010 \\ \hline (\dots 11)1011_2 \end{array} = 11_{10}(= -5 \bmod 16)$$

Warum das funktioniert

Modulo-Arithmetik: Rechnen im Kreis³



$11 \equiv 23 \equiv -1 \equiv \dots \pmod{12}$

$4 \equiv 16 \equiv \dots \pmod{12}$

$3 \equiv 15 \equiv \dots \pmod{12}$

³Die Arithmetik funktioniert auch mit Dezimalzahlen (und auch für die Multiplikation)

Negative Zahlen (3 Stellen)

	a	$-a$
0	000	
1	001	
2	010	
3	011	
4	100	
5	101	
6	110	
7	111	

Negative Zahlen (3 Stellen)

	a	$-a$	
0	000	000	0
1	001		
2	010		
3	011		
4	100		
5	101		
6	110		
7	111		

Negative Zahlen (3 Stellen)

	a	$-a$	
0	000	000	0
1	001	111	-1
2	010		
3	011		
4	100		
5	101		
6	110		
7	111		

Negative Zahlen (3 Stellen)

	a	$-a$	
0	000	000	0
1	001	111	-1
2	010	110	-2
3	011		
4	100		
5	101		
6	110		
7	111		

Negative Zahlen (3 Stellen)

	a	$-a$	
0	000	000	0
1	001	111	-1
2	010	110	-2
3	011	101	-3
4	100		
5	101		
6	110		
7	111		

Negative Zahlen (3 Stellen)

	a	$-a$	
0	000	000	0
1	001	111	-1
2	010	110	-2
3	011	101	-3
4	100	100	-4
5	101		
6	110		
7	111		

Negative Zahlen (3 Stellen)

	a	$-a$	
0	000	000	0
1	001	111	-1
2	010	110	-2
3	011	101	-3
4	100	100	-4
5	101		
6	110		
7	111		

Negative Zahlen (3 Stellen)

	a	$-a$	
0	000	000	0
1	001	111	-1
2	010	110	-2
3	011	101	-3
4	100	100	-4
5	101		
6	110		
7	111		

Das höchste Bit entscheidet über das Vorzeichen *und* es trägt zum Zahlwert bei.

3. Wahrheitswerte

Boolesche Funktionen; der Typ `bool`; logische und relationale Operatoren; Kurzschlussauswertung

Wo wollen wir hin?

```
int a;  
std::cin >> a;  
if (a % 2 == 0)  
    std::cout << "even";  
else  
    std::cout << "odd";
```

Wo wollen wir hin?

```
int a;  
std::cin >> a;  
if (a % 2 == 0)  
    std::cout << "even";  
else  
    std::cout << "odd";
```

Verhalten hängt ab vom Wert eines *Booleschen Ausdrucks*

Wo wollen wir hin?

```
int a;  
std::cin >> a;  
if (a % 2 == 0)  
    std::cout << "even";  
else  
    std::cout << "odd";
```

Verhalten hängt ab vom Wert eines **Booleschen Ausdrucks**

Boolesche Werte in der Mathematik

Boolesche Ausdrücke können zwei mögliche Werte annehmen:

0 oder **1**

Boolesche Werte in der Mathematik

Boolesche Ausdrücke können zwei mögliche Werte annehmen:

0 oder **1**

- **0** entspricht „**falsch**“
- **1** entspricht „**wahr**“

Der Typ `bool` in C++

- Repräsentiert **Wahrheitswerte**

Der Typ `bool` in C++

- Repräsentiert **Wahrheitswerte**
- Literale `false` und `true`

Der Typ `bool` in C++

- Repräsentiert **Wahrheitswerte**
- Literale `false` und `true`
- Wertebereich `{false, true}`

```
bool b = true; // Variable mit Wert true (wahr)
```

Relationale Operatoren

a < b (kleiner als)

Zahlentyp \times Zahlentyp \rightarrow **bool**

R-Wert \times R-Wert \rightarrow R-Wert

Relationale Operatoren

a < b (kleiner als)

```
bool b = (1 < 3); // b =
```

Relationale Operatoren

a < b (kleiner als)

```
bool b = (1 < 3); // b = true (wahr)
```

Relationale Operatoren

`a >= b` (grösser gleich)

```
int a = 0;  
bool b = (a >= 3); // b =
```

Relationale Operatoren

`a >= b` (grösser gleich)

```
int a = 0;  
bool b = (a >= 3); // b = false (falsch)
```

Relationale Operatoren

`a == b` (gleich)

```
int a = 4;  
bool b = (a % 3 == 1); // b =
```

Relationale Operatoren

`a == b` (gleich)

```
int a = 4;  
bool b = (a % 3 == 1); // b = true (wahr)
```


Relationale Operatoren

`a != b` (ungleich)

```
int a = 1;  
bool b = (a != 2*a-1); // b =
```

Relationale Operatoren

`a != b` (ungleich)

```
int a = 1;  
bool b = (a != 2*a-1); // b = false (falsch)
```

Boolesche Funktionen in der Mathematik

- Boolesche Funktion

$$f : \{0, 1\}^2 \rightarrow \{0, 1\}$$

- 0 entspricht „falsch“.
- 1 entspricht „wahr“.

- „Logisches Und“

$$f : \{0, 1\}^2 \rightarrow \{0, 1\}$$

- 0 entspricht „falsch“.
- 1 entspricht „wahr“.

x	y	$\text{AND}(x, y)$
0	0	0
0	1	0
1	0	0
1	1	1

Logischer Operator &&

a && b (logisches Und)

```
int n = -1;  
int p = 3;  
bool b = (n < 0) && (0 < p); //
```

Logischer Operator &&

a && b (logisches Und)

```
int n = -1;  
int p = 3;  
bool b = (n < 0) && (0 < p); // b = true (wahr)
```

- „Logisches Oder“

$$f : \{0, 1\}^2 \rightarrow \{0, 1\}$$

- 0 entspricht „falsch“.
- 1 entspricht „wahr“.

x	y	$\text{OR}(x, y)$
0	0	0
0	1	1
1	0	1
1	1	1

Logischer Operator ||

`a || b` (logisches Oder)

```
int n = 1;  
int p = 0;  
bool b = (n < 0) || (0 < p); //
```


Logischer Operator ||

`a || b` (logisches Oder)

```
int n = 1;  
int p = 0;  
bool b = (n < 0) || (0 < p); // b = false (falsch)
```

- „Logisches Nicht“

$$f : \{0, 1\} \rightarrow \{0, 1\}$$

- 0 entspricht „falsch“.
- 1 entspricht „wahr“.

x	NOT(x)
0	1
1	0

Logischer Operator !

!b (logisches Nicht)

bool → **bool**

R-Wert → R-Wert

Logischer Operator !

!b (logisches Nicht)

```
int n = 1;  
bool b = !(n < 0); //
```

Logischer Operator !

!b (logisches Nicht)

```
int n = 1;  
bool b = !(n < 0); // b = true (wahr)
```

Präzedenzen

`!b && a`

Präzedenzen

`!b && a`
⇕
`(!b) && a`

Präzedenzen

a && b || c && d

Präzedenzen

`a && b || c && d`
⇕
`(a && b) || (c && d)`

Präzedenzen

a || b && c || d

Präzedenzen

`a || b && c || d`
⇕
`a || (b && c) || d`

Präzedenzen

```
7 + x < y && y != 3 * z || ! b
```

Präzedenzen

Der unäre logische Operator !
bindet stärker als

```
7 + x < y && y != 3 * z || (!b)
```

Präzedenzen

Der unäre logische Operator !

bindet stärker als

binäre arithmetische Operatoren. Diese

binden stärker als

```
(7 + x) < y && y != (3 * z) || (!b)
```

Präcedenzen

Der unäre logische Operator !

bindet stärker als

binäre arithmetische Operatoren. Diese

binden stärker als

relationale Operatoren,

und diese binden stärker als

```
((7 + x) < y) && (y != (3 * z)) || (!b)
```

Präcedenzen

Der unäre logische Operator !

bindet stärker als

binäre arithmetische Operatoren. Diese

binden stärker als

relationale Operatoren,

und diese binden stärker als

binäre logische Operatoren.

```
((7 + x) < y) && (y != (3 * z)) || (!b)
```

Einige Klammern auf den vorher gezeigten Folien waren unnötig.

Vollständigkeit

- AND, OR und NOT sind die in C++ verfügbaren Booleschen Funktionen.

- AND, OR und NOT sind die in C++ verfügbaren Booleschen Funktionen.
- Alle anderen *binären* Booleschen Funktionen sind daraus erzeugbar.

x	y	$\text{XOR}(x, y)$
0	0	0
0	1	1
1	0	1
1	1	0

$$\text{XOR}(x, y) = \text{AND}(\text{OR}(x, y), \text{NOT}(\text{AND}(x, y))).$$

$$\text{XOR}(x, y) = \text{AND}(\text{OR}(x, y), \text{NOT}(\text{AND}(x, y))).$$

$$x \oplus y = (x \vee y) \wedge \neg(x \wedge y).$$

$$\text{XOR}(x, y) = \text{AND}(\text{OR}(x, y), \text{NOT}(\text{AND}(x, y))).$$

$$x \oplus y = (x \vee y) \wedge \neg(x \wedge y).$$

$$(x \ || \ y) \ \&\& \ !(x \ \&\& \ y)$$

Vollständigkeit Beweis

- Identifiziere binäre Boolesche Funktionen mit ihrem charakteristischen Vektor.

Vollständigkeit Beweis

- Identifiziere binäre Boolesche Funktionen mit ihrem charakteristischen Vektor.

x	y	$\text{XOR}(x, y)$
0	0	0
0	1	1
1	0	1
1	1	0

Vollständigkeit Beweis

- Identifiziere binäre Boolesche Funktionen mit ihrem charakteristischen Vektor.

x	y	$\text{XOR}(x, y)$
0	0	0
0	1	1
1	0	1
1	1	0

Charakteristischer Vektor: 0110

Vollständigkeit Beweis

- Identifiziere binäre Boolesche Funktionen mit ihrem charakteristischen Vektor.

x	y	$\text{XOR}(x, y)$
0	0	0
0	1	1
1	0	1
1	1	0

Charakteristischer Vektor: 0110

$$\text{XOR} = f_{0110}$$

Vollständigkeit Beweis

- Schritt 1: erzeuge die *elementaren* Funktionen f_{0001} , f_{0010} , f_{0100} , f_{1000}

$$f_{0001} = \text{AND}(x, y)$$

$$f_{0010} = \text{AND}(x, \text{NOT}(y))$$

$$f_{0100} = \text{AND}(y, \text{NOT}(x))$$

$$f_{1000} = \text{NOT}(\text{OR}(x, y))$$

Vollständigkeit Beweis

- Schritt 2: erzeuge alle Funktionen durch „Veroderung“ elementarer Funktionen

$$f_{1101} = \text{OR}(f_{1000}, \text{OR}(f_{0100}, f_{0001}))$$

Vollständigkeit Beweis

- Schritt 2: erzeuge alle Funktionen durch „Veroderung“ elementarer Funktionen

$$f_{1101} = \text{OR}(f_{1000}, \text{OR}(f_{0100}, f_{0001}))$$

- Schritt 3: erzeuge f_{0000}

$$f_{0000} = 0.$$

bool vs int: Konversion

- **bool** kann überall dort verwendet werden, wo **int** gefordert ist

bool vs int: Konversion

- **bool** kann überall dort verwendet werden, wo **int** gefordert ist

bool	→	int
<i>true</i>	→	1
<i>false</i>	→	0

bool vs int: Konversion

- **bool** kann überall dort verwendet werden, wo **int** gefordert ist – und umgekehrt.

bool	→	int
<i>true</i>	→	1
<i>false</i>	→	0

int	→	bool
$\neq 0$	→	<i>true</i>
0	→	<i>false</i>

bool vs int: Konversion

- **bool** kann überall dort verwendet werden, wo **int** gefordert ist – und umgekehrt.

bool	→	int
<i>true</i>	→	1
<i>false</i>	→	0

int	→	bool
$\neq 0$	→	<i>true</i>
0	→	<i>false</i>

```
bool b = 3; // b=true
```


bool vs int: Konversion

- **bool** kann überall dort verwendet werden, wo **int** gefordert ist – und umgekehrt.
- Viele existierende Programme verwenden statt **bool** den Typ **int**.

Das ist schlechter Stil, der noch auf die Sprache C zurückgeht.

bool	→	int
<i>true</i>	→	1
<i>false</i>	→	0

int	→	bool
≠0	→	<i>true</i>
0	→	<i>false</i>

```
bool b = 3; // b=true
```

DeMorgansche Regeln

■ $!(a \ \&\& \ b) == (!a \ || \ !b)$

DeMorgansche Regeln

■ $!(a \ \&\& \ b) == (!a \ || \ !b)$

! (reich *und* schön) == (arm *oder* hässlich)

DeMorgansche Regeln

■ $!(a \ \&\& \ b) == (!a \ || \ !b)$

■ $!(a \ || \ b) == (!a \ \&\& \ !b)$

! (reich *und* schön) == (arm *oder* hässlich)

Anwendung: Entweder ... Oder (XOR)

`(x || y) && !(x && y)`

Anwendung: Entweder ... Oder (XOR)

`(x || y) && !(x && y)` x oder y, und nicht beide

Anwendung: Entweder ... Oder (XOR)

`(x || y) && !(x && y)` x oder y, und nicht beide

`(x || y) && (!x || !y)`

Anwendung: Entweder ... Oder (XOR)

`(x || y) && !(x && y)` x oder y, und nicht beide

`(x || y) && (!x || !y)` x oder y, und eines nicht

Anwendung: Entweder ... Oder (XOR)

`(x || y) && !(x && y)` x oder y, und nicht beide

`(x || y) && (!x || !y)` x oder y, und eines nicht

`!(!x && !y) && !(x && y)`

Anwendung: Entweder ... Oder (XOR)

`(x || y) && !(x && y)` x oder y, und nicht beide

`(x || y) && (!x || !y)` x oder y, und eines nicht

`!(!x && !y) && !(x && y)` nicht keines, und nicht beide

Anwendung: Entweder ... Oder (XOR)

`(x || y) && !(x && y)` x oder y, und nicht beide

`(x || y) && (!x || !y)` x oder y, und eines nicht

`!(!x && !y) && !(x && y)` nicht keines, und nicht beide

`!(!x && !y || x && y)`

Anwendung: Entweder ... Oder (XOR)

`(x || y) && !(x && y)` x oder y, und nicht beide

`(x || y) && (!x || !y)` x oder y, und eines nicht

`!(!x && !y) && !(x && y)` nicht keines, und nicht beide

`!(!x && !y || x && y)` nicht: keines oder beide

Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken Operanden* zuerst aus.
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet.

Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken Operanden zuerst* aus.
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet.

x hat Wert 6 \Rightarrow

```
x != 0 && z / x > y
```

Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken Operanden zuerst* aus.
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet.

x hat Wert 6 \Rightarrow

```
true && z / x > y
```

Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken Operanden zuerst* aus.
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet.

x hat Wert 6 \Rightarrow

```
true && z / x > y
```


Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken Operanden zuerst* aus.
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet.

x hat Wert 0 \Rightarrow

```
x != 0 && z / x > y
```

Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken Operanden zuerst* aus.
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet.

x hat Wert 0 \Rightarrow

```
false && z / x > y
```

Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken Operanden* zuerst aus.
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet.

x hat Wert 0 \Rightarrow

`false` (falsch)

Kurzschlussauswertung

- Logische Operatoren `&&` und `||` werten den *linken Operanden zuerst* aus.
- Falls das Ergebnis dann schon feststeht, wird der rechte Operand *nicht mehr* ausgewertet.

x hat Wert 0 \Rightarrow

```
x != 0 && z / x > y
```

\Rightarrow Keine Division durch 0

4. Defensives Programmieren

Konstanten und Assertions

Fehlerquellen

- Fehler, die der Compiler findet:
syntaktische und manche semantische Fehler

Fehlerquellen

- Fehler, die der Compiler findet:
syntaktische und manche semantische Fehler
- Fehler, die der Compiler nicht findet:
Laufzeitfehler (immer semantisch)

Der Compiler als Freund: Konstanten

Konstanten

- sind Variablen mit unveränderbarem Wert

```
const int speed_of_light = 299792458;
```

- Verwendung: `const` vor der Definition

Der Compiler als Freund: Konstanten

Konstanten

- sind Variablen mit unveränderbarem Wert

```
const int speed_of_light = 299792458;
```

- Verwendung: `const` vor der Definition

Der Compiler als Freund: Konstanten

Konstanten

- sind Variablen mit unveränderbarem Wert

```
const int speed_of_light = 299792458;
```

- Verwendung: **const** vor der Definition

Der Compiler als Freund: Konstanten

- Compiler kontrolliert Einhaltung des **const**-Versprechens

```
const int speed_of_light = 299792458;  
...  
speed_of_light = 300000000;
```

Compilerfehler!



- Hilfsmittel zur Vermeidung von Fehlern: Konstanten erlauben garantierte Einhaltung des Versprechens „Wert ändert sich nicht“

Der Compiler als Freund: Konstanten

- Compiler kontrolliert Einhaltung des **const**-Versprechens

```
const int speed_of_light = 299792458;  
...  
speed_of_light = 300000000;
```

Compilerfehler!



- Hilfsmittel zur Vermeidung von Fehlern: Konstanten erlauben garantierte Einhaltung des Versprechens „Wert ändert sich nicht“

Der Compiler als Freund: Konstanten

- Compiler kontrolliert Einhaltung des `const`-Versprechens

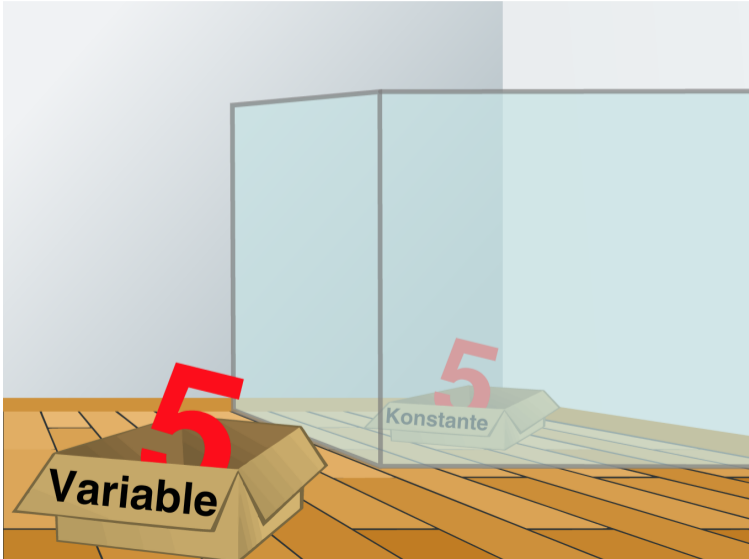
```
const int speed_of_light = 299792458;  
...  
speed_of_light = 300000000;
```

Compilerfehler!



- Hilfsmittel zur Vermeidung von Fehlern: Konstanten erlauben garantierte Einhaltung des Versprechens „Wert ändert sich nicht“

Konstanten: Variablen hinter Glas



Die `const`-Richtlinie

Const-Richtlinie

Denke bei *jeder Variablen* darüber nach, ob sie im Verlauf des Programmes jemals ihren Wert ändern wird oder nicht. Im letzteren Falle verwende das Schlüsselwort **`const`**, um die Variable zu einer Konstanten zu machen.

Ein Programm, welches diese Richtlinie befolgt, heisst **`const`**-korrekt.

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens

» It's not a bug, it's a feature! «

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens
2. Überprüfe an vielen kritischen Stellen, ob das Programm auf dem richtigen Weg ist

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens
2. Überprüfe an vielen kritischen Stellen, ob das Programm auf dem richtigen Weg ist
3. Hinterfrage auch das (scheinbar) Offensichtliche, es könnte sich ein simpler Tippfehler eingeschlichen haben

Gegen Laufzeitfehler: *Assertions*

`assert (expr)`

- hält das Programm an, falls der boolesche Ausdruck `expr` nicht wahr ist

Gegen Laufzeitfehler: *Assertions*

`assert (expr)`

- hält das Programm an, falls der boolesche Ausdruck `expr` nicht wahr ist
- benötigt `#include <cassert>`

Gegen Laufzeitfehler: *Assertions*

`assert (expr)`

- hält das Programm an, falls der boolesche Ausdruck `expr` nicht wahr ist
- benötigt `#include <cassert>`
- kann abgeschaltet werden (potentieller Geschwindigkeitsgewinn)

Assertions für den $ggT(x, y)$

Überprüfe, ob das Programm auf dem richtigen Weg ist ...

```
// Input x and y
std::cout << "x =? ";
std::cin >> x;
std::cout << "y =? ";
std::cin >> y;
```

Eingabe der Argumente für
die Berechnung

```
// Check validity of inputs
assert(x > 0 && y > 0);
```

```
... // Compute gcd(x,y), store result in variable a
```

Assertions für den $ggT(x, y)$

Überprüfe, ob das Programm auf dem richtigen Weg ist ...

```
// Input x and y
std::cout << "x =? ";
std::cin >> x;
std::cout << "y =? ";
std::cin >> y;
```

```
// Check validity of inputs
```

```
assert(x > 0 && y > 0);
```

← Vorbedingung für die weitere Berechnung

```
... // Compute gcd(x,y), store result in variable a
```


Assertions für den $ggT(x, y)$

... und hinterfrage das Offensichtliche! ...

...

```
assert(x > 0 && y > 0);
```

← Vorbedingung für die weitere Berechnung

```
... // Compute gcd(x,y), store result in variable a
```

```
assert (a >= 1);
```

```
assert (x % a == 0 && y % a == 0);
```

```
for (int i = a+1; i <= x && i <= y; ++i)
```

```
    assert(!(x % i == 0 && y % i == 0));
```

Assertions für den $ggT(x, y)$

... und hinterfrage das Offensichtliche! ...

...

```
assert(x > 0 && y > 0);
```

```
... // Compute gcd(x,y), store result in variable a
```

```
assert (a >= 1);
```

```
assert (x % a == 0 && y % a == 0);
```

```
for (int i = a+1; i <= x && i <= y; ++i)
```

```
    assert(!(x % i == 0 && y % i == 0));
```

Verschiedene
Eigenschaften
des ggT
überprüfen

Assertions abschalten

```
#define NDEBUG // To ignore assertions  
#include<cassert>
```

```
...
```

```
assert(x > 0 && y > 0); // Ignored
```

```
... // Compute gcd(x,y), store result in variable a
```

```
assert(a >= 1); // Ignored
```

```
...
```

Fail-Fast mit Assertions

- Reale Software: viele C++-Dateien, komplexer Kontrollfluss



Fail-Fast mit Assertions

- Reale Software: viele C++-Dateien, komplexer Kontrollfluss



Fail-Fast mit Assertions

- Reale Software: viele C++-Dateien, komplexer Kontrollfluss
- Fehler machen sich erst spät(er) bemerkbar → Fehlersuche erschwert



Fail-Fast mit Assertions

- Reale Software: viele C++-Dateien, komplexer Kontrollfluss
- Fehler machen sich erst spät(er) bemerkbar → Fehlersuche erschwert
- Assertions: Fehler frühzeitig bemerken



5. Kontrollanweisungen I

Auswahanweisungen, Iterationsanweisungen, Terminierung,
Blöcke

Kontrollfluss

- Bisher: *linear* (von oben nach unten)
- Interessante Programme nutzen „Verzweigungen“ und „Sprünge“

```
// Project Hangman
...
while (game_not_over) {
    ...
    if (word.contains(guess)) {
        ...
    } else {
        ...
    }
}
```

Auswahlanweisungen

realisieren Verzweigungen

- **if** Anweisung
- **if-else** Anweisung

if-Anweisung

```
if ( condition )  
    statement
```

if-Anweisung

```
if ( condition )  
    statement
```

```
int a;  
std::cin >> a;  
if (a % 2 == 0)  
    std::cout << "even";
```

if-Anweisung

```
if ( condition )  
    statement
```

Ist *condition* wahr, dann wird *statement* ausgeführt.

```
int a;  
std::cin >> a;  
if (a % 2 == 0)  
    std::cout << "even";
```

if-Anweisung

```
if ( condition )  
    statement
```

```
int a;  
std::cin >> a;  
if (a % 2 == 0)  
    std::cout << "even";
```

Ist *condition* wahr, dann wird *statement* ausgeführt.

- *statement*: beliebige Anweisung (*Rumpf* der **if**-Anweisung)
- *condition*: konvertierbar nach **bool**

if-else-Anweisung

```
if ( condition )  
    statement1  
else  
    statement2
```

if-else-Anweisung

```
if ( condition )  
    statement1  
else  
    statement2
```

```
int a;  
std::cin >> a;  
if (a % 2 == 0)  
    std::cout << "even";  
else  
    std::cout << "odd";
```


if-else-Anweisung

```
if ( condition )  
    statement1  
else  
    statement2
```

Ist *condition* wahr, so wird *statement1* ausgeführt, andernfalls wird *statement2* ausgeführt.

```
int a;  
std::cin >> a;  
if (a % 2 == 0)  
    std::cout << "even";  
else  
    std::cout << "odd";
```

if-else-Anweisung

```
if ( condition )  
    statement1  
else  
    statement2
```

```
int a;  
std::cin >> a;  
if (a % 2 == 0)  
    std::cout << "even";  
else  
    std::cout << "odd";
```

Ist *condition* wahr, so wird *statement1* ausgeführt, andernfalls wird *statement2* ausgeführt.

- *condition*: konvertierbar nach **bool**.
- *statement1*: Rumpf des **if**-Zweiges
- *statement2*: Rumpf des **else**-Zweiges

Layout!

```
int a;  
std::cin >> a;  
if (a % 2 == 0)  
    std::cout << "even";  
else  
    std::cout << "odd";
```

Layout!

```
int a;  
std::cin >> a;  
if (a % 2 == 0)  
    std::cout << "even"; ← Einrückung  
else  
    std::cout << "odd"; ← Einrückung
```

Iterationsanweisungen

realisieren Schleifen:

- **for**-Anweisung
- **while**-Anweisung
- **do**-Anweisung

Berechne $1 + 2 + \dots + n$

```
// input
std::cout << "Compute the sum 1+...+n for n=";
unsigned int n;
std::cin >> n;

// computation of sum_{i=1}^n i
unsigned int s = 0;
for (unsigned int i = 1; i <= n; ++i)
    s += i;

// output
std::cout << "1+...+" << n << " = " << s << ".\n";
```

Berechne $1 + 2 + \dots + n$

```
// input
std::cout << "Compute the sum 1+...+n for n=?";
unsigned int n;
std::cin >> n;

// computation of sum_{i=1}^n i
unsigned int s = 0;
for (unsigned int i = 1; i <= n; ++i)
    s += i;

// output
std::cout << "1+...+" << n << " = " << s << ".\n";
```

for-Anweisung am Beispiel

```
for ( unsigned int i=1; i <= n ; ++i )  
    s += i;
```

Annahmen: $n == 2$, $s == 0$

i

s

for-Anweisung am Beispiel

```
for ( unsigned int i=1; i <= n ; ++i )  
    s += i;
```

Annahmen: $n == 2, s == 0$

i	s
i==1	

for-Anweisung am Beispiel

```
for ( unsigned int i=1; i <= n ; ++i )  
    s += i;
```

Annahmen: $n == 2$, $s == 0$

i	s
i==1	i <= 2?

for-Anweisung am Beispiel

```
for ( unsigned int i=1; i <= n ; ++i )  
    s += i;
```

Annahmen: $n == 2$, $s == 0$

i	s
i==1	wahr

for-Anweisung am Beispiel

```
for ( unsigned int i=1; i <= n ; ++i )  
    s += i;
```

Annahmen: $n == 2, s == 0$

i		s
i==1	wahr	s == 1

for-Anweisung am Beispiel

```
for ( unsigned int i=1; i <= n ; ++i )  
    s += i;
```

Annahmen: $n == 2, s == 0$

i		s
i==1	wahr	s == 1
i==2		

for-Anweisung am Beispiel

```
for ( unsigned int i=1; i <= n ; ++i )  
    s += i;
```

Annahmen: $n == 2, s == 0$

i		s
i==1	wahr	s == 1
i==2	i <= 2?	

for-Anweisung am Beispiel

```
for ( unsigned int i=1; i <= n ; ++i )  
    s += i;
```

Annahmen: $n == 2, s == 0$

i		s
i==1	wahr	s == 1
i==2	wahr	

for-Anweisung am Beispiel

```
for ( unsigned int i=1; i <= n ; ++i )  
    s += i;
```

Annahmen: $n == 2$, $s == 0$

i		s
i==1	wahr	s == 1
i==2	wahr	s == 3

for-Anweisung am Beispiel

```
for ( unsigned int i=1; i <= n ; ++i )  
    s += i;
```

Annahmen: $n == 2, s == 0$

i		s
i==1	wahr	s == 1
i==2	wahr	s == 3
i==3		

for-Anweisung am Beispiel

```
for ( unsigned int i=1; i <= n ; ++i )  
    s += i;
```

Annahmen: $n == 2$, $s == 0$

i		s
i==1	wahr	s == 1
i==2	wahr	s == 3
i==3	i <= 2?	

for-Anweisung am Beispiel

```
for ( unsigned int i=1; i <= n ; ++i )  
    s += i;
```

Annahmen: $n == 2$, $s == 0$

i		s
i==1	wahr	s == 1
i==2	wahr	s == 3
i==3	falsch	

for-Anweisung am Beispiel

```
for ( unsigned int i=1; i <= n ; ++i )  
    s += i;
```

Annahmen: $n == 2$, $s == 0$

i		s
i==1	wahr	s == 1
i==2	wahr	s == 3
i==3	falsch	
		s == 3

for-Anweisung: Syntax

```
for (init statement; condition; expression)  
    body statement
```

for-Anweisung: Syntax

```
for (init statement; condition; expression)  
    body statement
```

- *init statement*: Ausdrucksanweisung, Deklarationsanweisung, Nullanweisung

for-Anweisung: Syntax

```
for (init statement; condition; expression)  
    body statement
```

- *init statement*: Ausdrucksanweisung, Deklarationsanweisung, Nullanweisung
- *condition*: konvertierbar nach **bool**

for-Anweisung: Syntax

```
for (init statement; condition; expression)  
    body statement
```

- *init statement*: Ausdrucksanweisung, Deklarationsanweisung, Nullanweisung
- *condition*: konvertierbar nach **bool**
- *expression*: beliebiger Ausdruck

for-Anweisung: Syntax

```
for (init statement; condition; expression)  
    body statement
```

- *init statement*: Ausdrucksanweisung, Deklarationsanweisung, Nullanweisung
- *condition*: konvertierbar nach **bool**
- *expression*: beliebiger Ausdruck
- *body statement*: beliebige Anweisung (*Rumpf* der for-Anweisung)

for-Anweisung: Terminierung

```
for (unsigned int i = 1; i <= n; ++i)
    s += i;
```

Hier und meistens:

- *expression* ändert einen Wert, der in *condition* vorkommt.

for-Anweisung: Terminierung

```
for (unsigned int i = 1; i <= n; ++i)  
    s += i;
```

Hier und meistens:

- Nach endlich vielen Iterationen wird *condition* falsch:
Terminierung.

Endlosschleifen

- Endlosschleifen sind leicht zu produzieren:

```
for ( ; ; ) ;
```

- Die *leere condition* ist wahr.
- Die *leere expression* hat keinen Effekt.
- Die *Nullanweisung* hat keinen Effekt.

Endlosschleifen

- Endlosschleifen sind leicht zu produzieren:

```
for ( ; ; ) ;
```

- Die *leere condition* ist wahr.
 - Die *leere expression* hat keinen Effekt.
 - Die *Nullanweisung* hat keinen Effekt.
- ... aber nicht automatisch zu erkennen.

```
for (init; cond; expr) stmt;
```

Halteproblem

Unentscheidbarkeit des Halteproblems

Es gibt kein C++ Programm, das für jedes C++- Programm P und jede Eingabe I korrekt feststellen kann, ob das Programm P bei Eingabe von I terminiert.

⁴Alan Turing, 1936. Theoretische Fragestellungen dieser Art waren für Alan Turing die Hauptmotivation für die Konstruktion seiner Rechenmaschine.

Halteproblem

Unentscheidbarkeit des Halteproblems

Es gibt kein C++ Programm, das für jedes C++- Programm P und jede Eingabe I korrekt feststellen kann, ob das Programm P bei Eingabe von I terminiert.

Das heisst, die Korrektheit von Programmen kann *nicht* automatisch überprüft werden.⁴

⁴Alan Turing, 1936. Theoretische Fragestellungen dieser Art waren für Alan Turing die Hauptmotivation für die Konstruktion seiner Rechenmaschine.

Beispiel: Primzahltest

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \dots, n - 1\}$ ein Teiler von n ist.

Beispiel: Primzahltest

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \dots, n - 1\}$ ein Teiler von n ist.

Eine Schleife, die das testet:

```
unsigned int d;  
for (d=2; n%d != 0; ++d);
```

Beispiel: Primzahltest

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \dots, n - 1\}$ ein Teiler von n ist.

Eine Schleife, die das testet:

```
unsigned int d;  
for (d=2; n%d != 0; ++d);
```

(Rumpf ist die Null-Anweisung)

Primzahltest: Terminierung

```
unsigned int d;  
for (d=2; n%d != 0; ++d); // for n >= 2
```

- Fortschritt: Startwert **d=2**, dann in jeder Iteration plus 1 (**++d**)

Primzahltest: Terminierung

```
unsigned int d;  
for (d=2; n%d != 0; ++d); // for n >= 2
```

- Fortschritt: Startwert **d=2**, dann in jeder Iteration plus 1 (**++d**)
- Abbruch: **n%d != 0** evaluiert zu **false** sobald ein Teiler erreicht wurde — spätestens, wenn **d == n**

Primzahltest: Terminierung

```
unsigned int d;  
for (d=2; n%d != 0; ++d); // for n >= 2
```

- Fortschritt: Startwert **d=2**, dann in jeder Iteration plus 1 (**++d**)
- Abbruch: **n%d != 0** evaluiert zu **false** sobald ein Teiler erreicht wurde — spätestens, wenn **d == n**
- Fortschritt garantiert, dass Abbruchbedingung erreicht wird

Primzahltest: Korrektheit

```
unsigned int d;  
for (d=2; n%d != 0; ++d); // for n >= 2
```

Jeder mögliche Teiler $2 \leq d \leq n$ wird ausprobiert. Falls die Schleife mit $d == n$ terminiert, dann und genau dann ist n prim.

Blöcke

- Blöcke gruppieren mehrere Anweisungen zu einer neuen Anweisung

```
{statement1 statement2 ... statementN}
```

Blöcke

- Blöcke gruppieren mehrere Anweisungen zu einer neuen Anweisung
- Beispiel: Rumpf der main Funktion

```
int main() {  
    ...  
}
```


Blöcke

- Blöcke gruppieren mehrere Anweisungen zu einer neuen Anweisung
- Beispiel: Schleifenrumpf

```
for (unsigned int i = 1; i <= n; ++i) {  
    s += i;  
    std::cout << "partial sum is " << s << "\n";  
}
```

Blöcke

- Blöcke gruppieren mehrere Anweisungen zu einer neuen Anweisung
- Beispiel: if / else

```
if (d < n) // d is a divisor of n in {2,...,n-1}
    std::cout << n << " = " << d << " * " << n / d << ".\n";
else {
    assert (d == n);
    std::cout << n << " is prime.\n";
}
```

6. Kontrollanweisungen II

Sichtbarkeit, Lokale Variablen, While-Anweisung,
Do-Anweisung, Sprunganweisungen

Sichtbarkeit

Deklaration in einem Block ist ausserhalb des Blocks nicht *sichtbar*.

```
int main()
{
  {
    int i = 2;
  }
  std::cout << i; // Fehler: undeklariertes Name
  return 0;
} „Blickrichtung“
```

Potenzieller Gültigkeitsbereich

Im Block

```
{  
    ...  
    int i = 2;  
    ...  
}
```

Im Funktionsrumpf

```
int main() {  
    ...  
    int i = 2;  
    ...  
    return 0;  
}
```

In Kontrollanweisung

```
for (int i = 0; i < 10; ++i) {s += i; ... }
```

Potenzieller Gültigkeitsbereich

Im Block

```
scope | {  
      |     ...  
      |     int i = 2;  
      |     ...  
      | }
```

Im Funktionsrumpf

```
scope | int main() {  
      |     ...  
      |     int i = 2;  
      |     ...  
      |     return 0;  
      | }
```

In Kontrollanweisung

```
for (int i = 0; i < 10; ++i) {s += i; ... }
```

scope

Gültigkeitsbereich

```
int main()
{
    int i = 2;
    for (int i = 0; i < 5; ++i)
        // outputs 0,1,2,3,4
        std::cout << i;
    // outputs 2
    std::cout << i;
    return 0;
}
```

Potenzieller Gültigkeitsbereich

```
int main()
{
    int i = 2;
    for (int i = 0; i < 5; ++i)
        // outputs 0,1,2,3,4
        std::cout << i;
    // outputs 2
    std::cout << i;
    return 0;
}
```


Wirklicher Gültigkeitsbereich

```
int main()
{
  int i = 2;
  for (int i = 0; i < 5; ++i)
    // outputs 0,1,2,3,4
    std::cout << i;
  // outputs 2
  std::cout << i;
  return 0;
}
```

Lokale Variablen

```
int main()
{
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        std::cout << ++i; // outputs
        int k = 2;
        std::cout << --k; // outputs
    }
}
```

Lokale Variablen

```
int main()
{
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        std::cout << ++i; // outputs 6, 7, 8, 9, 10
        int k = 2;
        std::cout << --k; // outputs 1, 1, 1, 1, 1
    }
}
```

Lokale Variablen

```
int main()
{
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        std::cout << ++i; // outputs
        int k = 2;
        std::cout << --k; // outputs
    }
}
```

Lokale Variablen (Deklaration in einem Block) haben *automatische Speicherdauer*.

while Anweisung

```
while (condition)  
    statement
```

while Anweisung

```
while (condition)  
    statement
```

ist äquivalent zu

```
for (; condition; )  
    statement
```

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases} , i \geq 1.$

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases} , i \geq 1.$

n=5: 5

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases} , i \geq 1.$

n=5: 5, 16

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases} , i \geq 1.$

n=5: 5, 16, 8

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases} , i \geq 1.$

n=5: 5, 16, 8, 4

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases} , i \geq 1.$

n=5: 5, 16, 8, 4, 2

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases} , i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases} , i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases} , i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4, 2

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases} , i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4, 2, 1

Die Collatz-Folge

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & , \text{ falls } n_{i-1} \text{ gerade} \\ 3n_{i-1} + 1 & , \text{ falls } n_{i-1} \text{ ungerade} \end{cases} , i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4, 2, 1, ... (Repetition bei 1)

do Anweisung

```
do  
  statement  
while (condition);
```

do Anweisung

```
do  
  statement  
while (condition);
```

ist äquivalent zu

```
statement  
while (condition)  
  statement
```

break und continue in der Praxis

- Vorteil: Können verschachtelte **if-else**-Blöcke (oder komplexe Disjunktionen) vermeiden

break und continue in der Praxis

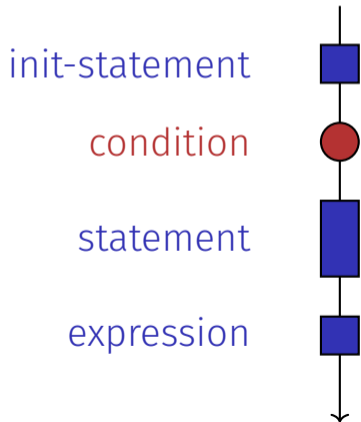
- Vorteil: Können verschachtelte **if-else**-Blöcke (oder komplexe Disjunktionen) vermeiden
- Aber führen zu mehr Sprüngen und somit zu potentiell komplexerem Kontrollfluss

break und continue in der Praxis

- Vorteil: Können verschachtelte **if-else**-Blöcke (oder komplexe Disjunktionen) vermeiden
- Aber führen zu mehr Sprüngen und somit zu potentiell komplexerem Kontrollfluss
- Ihr Einsatz ist daher umstritten und sollte mit Vorsicht geschehen

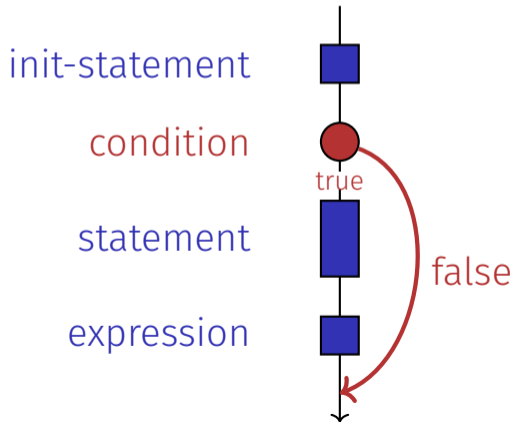
Kontrollfluss for

for (*init statement* *condition* ; *expression*)
statement



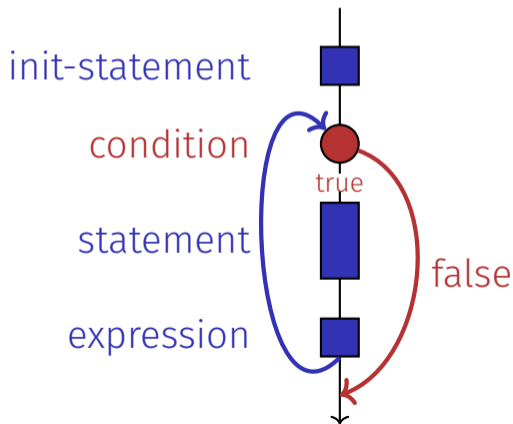
Kontrollfluss for

for (*init statement* *condition* ; *expression*)
statement

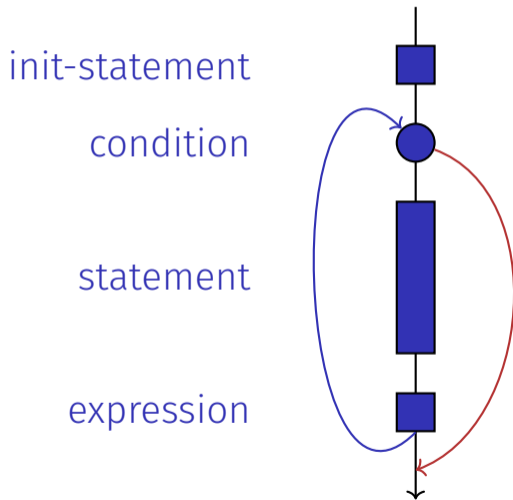


Kontrollfluss for

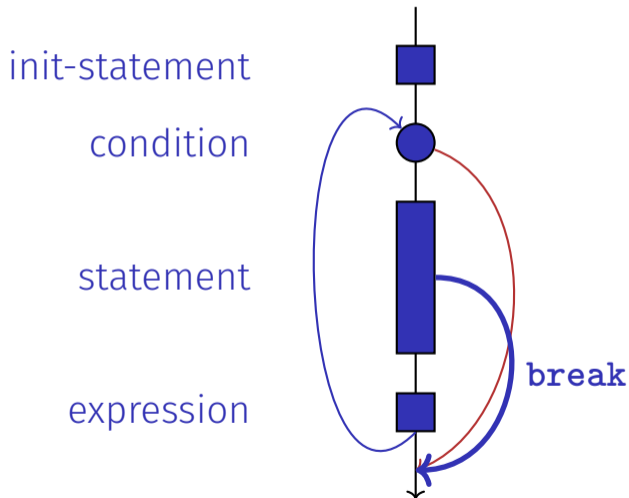
for (*init statement* *condition* ; *expression*)
statement



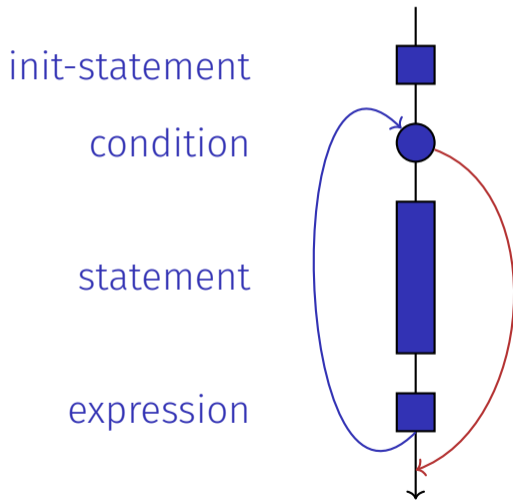
Kontrollfluss `break` und `continue` in `for`



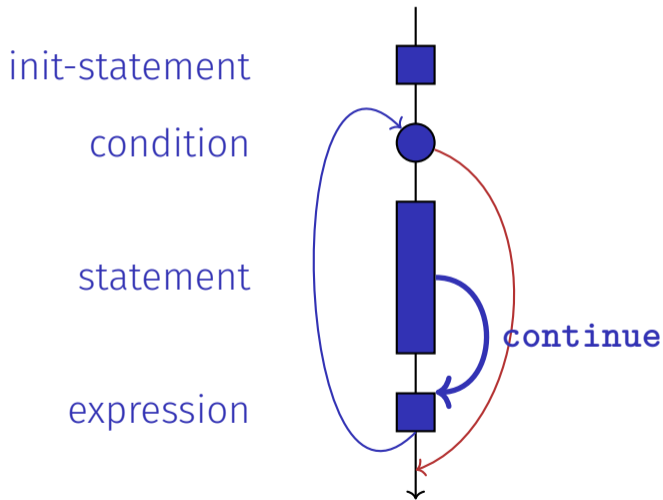
Kontrollfluss *break* und *continue* in *for*



Kontrollfluss `break` und `continue` in `for`



Kontrollfluss `break` und `continue` in `for`



Kontrollfluss: Die guten alten Zeiten?

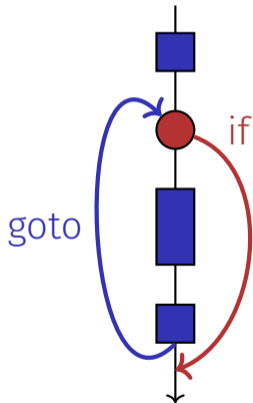
Beobachtung

Wir brauchen eigentlich nur **ifs** und Sprünge an beliebige Stellen im Programm (**goto**).

Kontrollfluss: Die guten alten Zeiten?

Beobachtung

Wir brauchen eigentlich nur **ifs** und Sprünge an beliebige Stellen im Programm (**goto**).



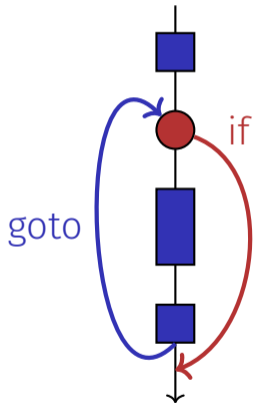
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

Wir brauchen eigentlich nur **ifs** und Sprünge an beliebige Stellen im Programm (**goto**).

Sprachen, die darauf basieren:

- Maschinensprache



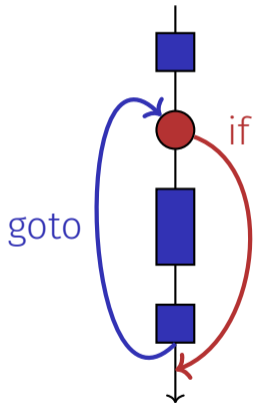
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

Wir brauchen eigentlich nur **ifs** und Sprünge an beliebige Stellen im Programm (**goto**).

Sprachen, die darauf basieren:

- Maschinensprache
- Assembler („höhere“ Maschinensprache)



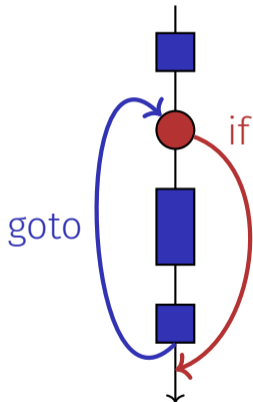
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

Wir brauchen eigentlich nur **ifs** und Sprünge an beliebige Stellen im Programm (**goto**).

Sprachen, die darauf basieren:

- Maschinensprache
- Assembler („höhere“ Maschinensprache)
- BASIC, die erste Programmiersprache für ein allgemeines Publikum (1964)



BASIC und die Home-Computer...

...ermöglichten einer ganzen Generation von Jugendlichen das Programmieren.



Home-Computer Commodore C64 (1982)

Spaghetti-Code mit goto

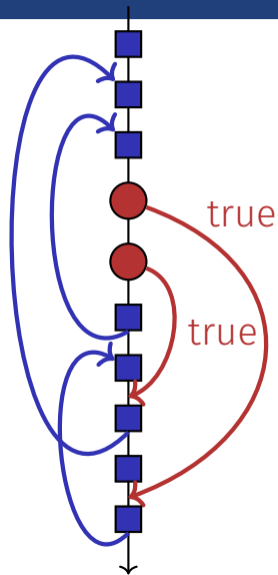
Ausgabe von ??????????
mit der Programmiersprache BASIC:

```
10 N=2
20 D=1
30 D=D+1
40 IF N=D GOTO 100
50 IF N/D = INT(N/D) GOTO 70
60 GOTO 30
70 N=N+1
80 GOTO 20
100 PRINT N
110 GOTO 70
```

Spaghetti-Code mit goto

Ausgabe aller Primzahlen
mit der Programmiersprache BASIC:

```
10 N=2
20 D=1
30 D=D+1
40 IF N=D GOTO 100
50 IF N/D = INT(N/D) GOTO 70
60 GOTO 30
70 N=N+1
80 GOTO 20
100 PRINT N
110 GOTO 70
```



Die „richtige“ Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

Die „richtige“ Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen

Die „richtige“ Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code

Die „richtige“ Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code
- Einfacher Kontrollfluss

Die „richtige“ Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code
- Einfacher Kontrollfluss
- Einfache Ausdrücke

Die „richtige“ Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- Wenige Anweisungen
- Wenige Zeilen Code
- Einfacher Kontrollfluss
- Einfache Ausdrücke

Ziele sind oft nicht gleichzeitig erreichbar.

Ungerade Zahlen in $\{0, \dots, 100\}$

Erster (korrekter) Versuch:

```
for (unsigned int i = 0; i < 100; ++i) {  
    if (i % 2 == 0)  
        continue;  
    std::cout << i << "\n";  
}
```

Ungerade Zahlen in $\{0, \dots, 100\}$

Weniger Anweisungen, **weniger** Zeilen:

```
for (unsigned int i = 0; i < 100; ++i) {  
    if (i % 2 != 0)  
        std::cout << i << "\n";  
}
```

Ungerade Zahlen in $\{0, \dots, 100\}$

Weniger Anweisungen, **einfacherer** Kontrollfluss:

```
for (unsigned int i = 1; i < 100; i += 2)
    std::cout << i << "\n";
```

Ungerade Zahlen in $\{0, \dots, 100\}$

Weniger Anweisungen, **einfacherer** Kontrollfluss:

```
for (unsigned int i = 1; i < 100; i += 2)
    std::cout << i << "\n";
```

Das ist hier die „richtige“ Iterationsanweisung

Notenausgabe

1. Funktionale Anforderung:

6 → "Excellent ... You passed!"

5,4 → "You passed!"

3 → "Close, but ... You failed!"

2,1 → "You failed!"

sonst → "Error!"

Notenausgabe

1. Funktionale Anforderung:

```
6 → "Excellent ... You passed!"  
5,4 → "You passed!"  
3 → "Close, but ... You failed!"  
2,1 → "You failed!"  
sonst → "Error!"
```

2. Ausserdem: Text- und Codeduplikation vermeiden

Notenausgabe mit if-Anweisungen

```
int grade;
...
if (grade == 6) std::cout << "Excellent ... ";
if (4 <= grade && grade <= 6) {
    std::cout << "You passed!";
} else if (1 <= grade && grade < 4) {
    if (grade == 3) std::cout << "Close, but ... ";
    std::cout << "You failed!";
} else std::cout << "Error!";
```

Notenausgabe mit if-Anweisungen


```
int grade;
...
if (grade == 6) std::cout << "Excellent ... ";
if (4 <= grade && grade <= 6) {
    std::cout << "You passed!";
} else if (1 <= grade && grade < 4) {
    if (grade == 3) std::cout << "Close, but ... ";
    std::cout << "You failed!";
} else std::cout << "Error!";
```

Nachteil: Kontrollfluss – und somit Programmverhalten – nicht gerade offensichtlich

Notenausgabe mit `switch`-Anweisung


```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```

Notenausgabe mit `switch`-Anweisung

```
switch (grade) {  Springe zu passendem case  
  case 6: std::cout << "Excellent ... ";  
  case 5:  
  case 4: std::cout << "You passed!";  
    break;  
  case 3: std::cout << "Close, but ... ";  
  case 2:  
  case 1: std::cout << "You failed!";  
    break;  
  default: std::cout << "Error!";  
}
```

Notenausgabe mit switch-Anweisung

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```



Durchfallen

Notenausgabe mit switch-Anweisung

```
switch (grade) {  
  case 6: std::cout << "Excellent ... ";  
  case 5:  
  case 4: std::cout << "You passed!";  
    break; ← Verlasse switch  
  case 3: std::cout << "Close, but ... ";  
  case 2:  
  case 1: std::cout << "You failed!";  
    break;  
  default: std::cout << "Error!";  
}
```

Durchfallen

Notenausgabe mit switch-Anweisung

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```



Notenausgabe mit `switch`-Anweisung

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break; ← Durchfallen  
        ← Verlasse switch  
    default: std::cout << "Error!";  
}
```

Notenausgabe mit switch-Anweisung

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!"; ← In allen anderen Fällen  
}
```

Notenausgabe mit `switch`-Anweisung

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```

Vorteil: Kontrollfluss klar erkennbar

Die `switch`-Anweisung

```
switch (expression)  
statement
```

- *expression*: Ausdruck, konvertierbar in einen integralen Typ
- *statement*: beliebige Anweisung, in welcher **case** und **default**-Marken erlaubt sind, **break** hat eine spezielle Bedeutung.

Die `switch`-Anweisung

```
switch (expression)  
statement
```

- *expression*: Ausdruck, konvertierbar in einen integralen Typ
- *statement* : beliebige Anweisung, in welcher **case** und **default**-Marken erlaubt sind, **break** hat eine spezielle Bedeutung.
- Benutzung des Durchfallens in der Praxis umstritten, Einsatz gut abwägen (entsprechende Compilerwarnung kann aktiviert werden)

7. Fließkommazahlen I

Typen **float** und **double**; Gemischte Ausdrücke und Konversionen; Lücken im Wertebereich;

„Richtig“ Rechnen

```
// Input
std::cout << "Temperature in degrees Celsius =? ";
int celsius;
std::cin >> celsius;

// Computation and output
std::cout << celsius << " degrees Celsius are "
          << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
```

28 degrees Celsius are 82 degrees Fahrenheit.

„Richtig“ Rechnen

```
// Input
std::cout << "Temperature in degrees Celsius =? ";
int celsius;
std::cin >> celsius;

// Computation and output
std::cout << celsius << " degrees Celsius are "
          << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\n";
```

28 degrees Celsius are 82 degrees Fahrenheit.



richtig wäre 82.4

„Richtig“ Rechnen

```
// Input
std::cout << "Temperature in degrees Celsius =? ";
float celsius; // Enable fractional numbers
std::cin >> celsius;

// Computation and output
std::cout << celsius << " degrees Celsius are "
          << 9 * celsius / 5 + 32 << " degrees Fahrenheit.\\n";
```

28 degrees Celsius are 82.4 degrees Fahrenheit.

Fixkommazahlen

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

Fixkommazahlen

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

$$82.4 = 0000082.400$$

Fixkommazahlen

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

$$82.4 = 0000082.400$$

Nachteile

- Wertebereich wird *noch* kleiner als bei ganzen Zahlen.

Fixkommazahlen

- feste Anzahl Vorkommastellen (z.B. 7)
- feste Anzahl Nachkommastellen (z.B. 3)

0.0824 = 0000000.082 ← dritte Stelle abgeschnitten

Nachteile

- Repräsentierbarkeit hängt von der Stelle des Kommas ab.

Fließkommazahlen

- Beobachtung: Unterschiedlich „effiziente“ Darstellungen einer Zahl, z.B.

$$\begin{aligned}0.0824 &= 0.00824 \cdot 10^1 = 0.824 \cdot 10^{-1} \\ &= 8.24 \cdot 10^{-2} = 824 \cdot 10^{-4}\end{aligned}$$

Anzahl *signifikanter Stellen* bleibt konstant

Fließkommazahlen

- Beobachtung: Unterschiedlich „effiziente“ Darstellungen einer Zahl, z.B.

$$\begin{aligned}0.0824 &= 0.00824 \cdot 10^1 &= 0.824 \cdot 10^{-1} \\ &= 8.24 \cdot 10^{-2} &= 824 \cdot 10^{-4}\end{aligned}$$

Anzahl *signifikanter Stellen* bleibt konstant

- Fließkommarepräsentation daher:
 - Feste Anzahl signifikanter Stellen (z.B. 10),
 - Plus Position des Kommas mittels Exponenten
 - Zahl ist $\text{Signifikand} \times 10^{\text{Exponent}}$

Typen `float` und `double`

- sind die fundamentalen C++ Typen für Fließkommazahlen
- approximieren den Körper der reellen Zahlen $(\mathbb{R}, +, \times)$ in der Mathematik

Typen `float` und `double`

- sind die fundamentalen C++ Typen für Fließkommazahlen
- approximieren den Körper der reellen Zahlen $(\mathbb{R}, +, \times)$ in der Mathematik
- haben grossen Wertebereich, ausreichend für viele Anwendungen:
 - **float**: ca. 7 Stellen, Exponent bis ± 38
 - **double**: ca. 15 Stellen, Exponent bis ± 308

Typen `float` und `double`

- sind die fundamentalen C++ Typen für Fließkommazahlen
- approximieren den Körper der reellen Zahlen $(\mathbb{R}, +, \times)$ in der Mathematik
- haben grossen Wertebereich, ausreichend für viele Anwendungen:
 - **float**: ca. 7 Stellen, Exponent bis ± 38
 - **double**: ca. 15 Stellen, Exponent bis ± 308
- sind auf den meisten Rechnern sehr schnell (Hardwareunterstützung)

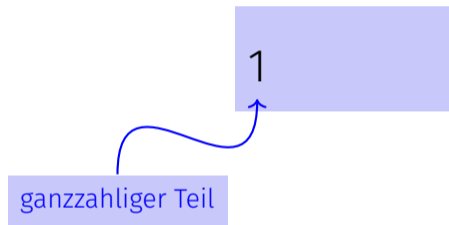
Arithmetische Operatoren

Wie bei **int**, aber ...

- Divisionsoperator / modelliert „echte“ (reelle, nicht ganzzahlige) Division
- Kein Modulo-Operator, d.h. kein %

Literale

unterscheiden sich von Ganzzahlliteralen

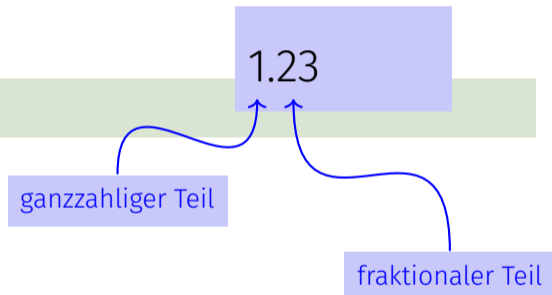


Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

■ Dezimalkomma

`1.0` : Typ `double`, Wert 1



Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma

`1.0` : Typ `double`, Wert 1

1 e-7

ganzzahliger Teil

Exponent

- oder Exponent.

`1e3` : Typ `double`, Wert 1000

Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

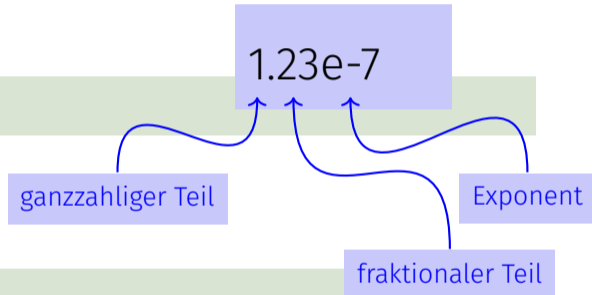
■ Dezimalkomma

`1.0` : Typ `double`, Wert 1

■ und / oder Exponent.

`1e3` : Typ `double`, Wert 1000

`1.23e-7` : Typ `double`, Wert $1.23 \cdot 10^{-7}$



Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

■ Dezimalkomma

`1.0` : Typ `double`, Wert 1

`1.27f` : Typ `float`, Wert 1.27

■ und / oder Exponent.

`1e3` : Typ `double`, Wert 1000

`1.23e-7` : Typ `double`, Wert $1.23 \cdot 10^{-7}$

`1.23e-7f` : Typ `float`, Wert $1.23 \cdot 10^{-7}$

1.23e-7f

ganzzahliger Teil

Exponent

fraktionaler Teil

Rechnen mit float: Beispiel

Approximation der Euler-Zahl

$$e = \sum_{i=0}^{\infty} \frac{1}{i!} \approx 2.71828\dots$$

mittels der ersten 10 Terme.

Rechnen mit float: Eulersche Zahl

```
std::cout << "Approximating the Euler number... \n";

// values for i-th iteration, initialized for i = 0
float t = 1.0f; // term 1/i!
float e = 1.0f; // i-th approximation of e

// iteration 1, ..., n
for (unsigned int i = 1; i < 10; ++i) {
    t /= i;    // 1/(i-1)! -> 1/i!
    e += t;
    std::cout << "Value after term " << i << ": "
                << e << "\n";
}
```

Rechnen mit float: Eulersche Zahl

```
Value after term 1: 2
Value after term 2: 2.5
Value after term 3: 2.66667
Value after term 4: 2.70833
Value after term 5: 2.71667
Value after term 6: 2.71806
Value after term 7: 2.71825
Value after term 8: 2.71828
Value after term 9: 2.71828
```

Gemischte Ausdrücke, Konversion

- Fließkommazahlen sind allgemeiner als ganzzahlige Typen.

Gemischte Ausdrücke, Konversion

- Fließkommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fließkommazahlen konvertiert.

Gemischte Ausdrücke, Konversion

- Fließkommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fließkommazahlen konvertiert.

```
9 * celsius / 5 + 32
```

Gemischte Ausdrücke, Konversion

- Fließkommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fließkommazahlen konvertiert.

9 * celsius / 5 + 32

↑
Typ float, Wert 28

Gemischte Ausdrücke, Konversion

- Fließkommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fließkommazahlen konvertiert.

9 * 28.0f / 5 + 32

Gemischte Ausdrücke, Konversion

- Fließkommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fließkommazahlen konvertiert.

9 * 28.0f / 5 + 32

wird zu `float` konvertiert: 9.0f

Gemischte Ausdrücke, Konversion

- Fließkommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fließkommazahlen konvertiert.

252.0f / 5 + 32

wird zu `float` konvertiert: 5.0f

Gemischte Ausdrücke, Konversion

- Fließkommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fließkommazahlen konvertiert.

50.4f + 32

wird zu `float` konvertiert: 32.0f

Gemischte Ausdrücke, Konversion

- Fließkommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fließkommazahlen konvertiert.

82.4f

Löcher im Wertebereich

```
float n1;  
std::cout << "First number =? ";  
std::cin >> n1;
```

```
float n2;  
std::cout << "Second number =? ";  
std::cin >> n2;
```

```
float d;  
std::cout << "Their difference =? ";  
std::cin >> d;
```

```
std::cout << "Computed difference - input difference = "  
          << n1 - n2 - d << "\n";
```

Löcher im Wertebereich

```
float n1;  
std::cout << "First number =? ";  
std::cin >> n1;
```

Eingabe 1.5

```
float n2;  
std::cout << "Second number =? ";  
std::cin >> n2;
```

Eingabe 1.0

```
float d;  
std::cout << "Their difference =? ";  
std::cin >> d;
```

Eingabe 0.5

```
std::cout << "Computed difference - input difference = "  
          << n1 - n2 - d << "\n";
```

Löcher im Wertebereich

```
float n1;  
std::cout << "First number =? ";  
std::cin >> n1;
```

Eingabe 1.5

```
float n2;  
std::cout << "Second number =? ";  
std::cin >> n2;
```

Eingabe 1.0

```
float d;  
std::cout << "Their difference =? ";  
std::cin >> d;
```

Eingabe 0.5

```
std::cout << "Computed difference - input difference = "  
          << n1 - n2 - d << "\n";
```

Ausgabe 0

Löcher im Wertebereich

```
float n1;  
std::cout << "First number =? ";  
std::cin >> n1;
```

Eingabe 1.1

```
float n2;  
std::cout << "Second number =? ";  
std::cin >> n2;
```

Eingabe 1.0

```
float d;  
std::cout << "Their difference =? ";  
std::cin >> d;
```

Eingabe 0.1

```
std::cout << "Computed difference - input difference = "  
          << n1 - n2 - d << "\n";
```


Löcher im Wertebereich

```
float n1;  
std::cout << "First number =? ";  
std::cin >> n1;
```

Eingabe 1.1

```
float n2;  
std::cout << "Second number =? ";  
std::cin >> n2;
```

Eingabe 1.0

```
float d;  
std::cout << "Their difference =? ";  
std::cin >> d;
```

Eingabe 0.1

```
std::cout << "Computed difference - input difference = "  
          << n1 - n2 - d << "\n";
```

Ausgabe 2.23517e-8

Löcher im Wertebereich

```
float n1;  
std::cout << "First number =? ";  
std::cin >> n1;
```

Eingabe 1.1

```
float n2;  
std::cout << "Second number =? ";  
std::cin >> n2;
```

Eingabe 1.0

```
float d;  
std::cout << "Their difference =? ";  
std::cin >> d;
```

Eingabe 0.1

```
std::cout << "Computed difference - input difference = "  
          << n1 - n2 - d << "\n";
```

Ausgabe 2.23517e-8

Was ist denn hier los?

Wertebereich

Ganzzahlige Typen:

- Über- und Unterlauf häufig, aber ...
- Wertebereich ist zusammenhängend (keine Löcher): \mathbb{Z} ist „diskret“.

Wertebereich

Ganzzahlige Typen:

- Über- und Unterlauf häufig, aber ...
- Wertebereich ist zusammenhängend (keine Löcher): \mathbb{Z} ist „diskret“.

Fließkommatypen:

- Über- und Unterlauf selten, aber ...
- es gibt Löcher: \mathbb{R} ist „kontinuierlich“.

8. Fließkommazahlen II

Fließkommazahlensysteme; IEEE Standard; Grenzen der Fließkommaarithmetik; Fließkomma-Richtlinien; Harmonische Zahlen

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

- $\beta \geq 2$, die Basis,
- $p \geq 1$, die Präzision (Stellenzahl),
- e_{\min} , der kleinste Exponent,
- e_{\max} , der grösste Exponent.

Fließkommazahlensysteme

Ein Fließkommazahlensystem ist durch vier natürliche Zahlen definiert:

- $\beta \geq 2$, die Basis,
- $p \geq 1$, die Präzision (Stellenzahl),
- e_{\min} , der kleinste Exponent,
- e_{\max} , der grösste Exponent.

Bezeichnung:

$$F(\beta, p, e_{\min}, e_{\max})$$

Fließkommazahlensysteme

$F(\beta, p, e_{\min}, e_{\max})$ enthält die Zahlen

$$\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e,$$

$$d_i \in \{0, \dots, \beta - 1\}, \quad e \in \{e_{\min}, \dots, e_{\max}\}.$$

Fließkommazahlensysteme

$F(\beta, p, e_{\min}, e_{\max})$ enthält die Zahlen

$$\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e,$$

$$d_i \in \{0, \dots, \beta - 1\}, \quad e \in \{e_{\min}, \dots, e_{\max}\}.$$

In Basis- β -Darstellung:

$$\pm d_0 \bullet d_1 \dots d_{p-1} \times \beta^e,$$

Fließkommazahlensysteme

Darstellungen der Dezimalzahl 0.1 (mit $\beta = 10$):

$$1.0 \cdot 10^{-1}, \quad 0.1 \cdot 10^0, \quad 0.01 \cdot 10^1, \quad \dots$$

Unterschiedliche Darstellungsmöglichkeiten durch Wahl des Exponenten

Normalisierte Darstellung

Normalisierte Zahl:

$$\pm d_0 \bullet d_1 \dots d_{p-1} \times \beta^e, \quad d_0 \neq 0$$

Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

Normalisierte Darstellung

Normalisierte Zahl:

$$\pm d_0 \bullet d_1 \dots d_{p-1} \times \beta^e, \quad d_0 \neq 0$$

Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

Normalisierte Darstellung

Normalisierte Zahl:

$$\pm d_0 \bullet d_1 \dots d_{p-1} \times \beta^e, \quad d_0 \neq 0$$

Bemerkung 2

Die Zahl 0, sowie alle Zahlen kleiner als $\beta^{e_{\min}}$, haben keine normalisierte Darstellung (greifen wir später wieder auf)

Menge der normalisierten Zahlen

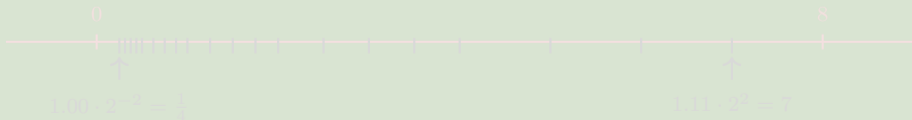
$$F^*(\beta, p, e_{\min}, e_{\max})$$

Normalisierte Darstellung

Beispiel $F^*(2, 3, -2, 2)$

(nur positive Zahlen)

$d_0 \bullet d_1 d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
1.00 ₂	0.25	0.5	1	2	4
1.01 ₂	0.3125	0.625	1.25	2.5	5
1.10 ₂	0.375	0.75	1.5	3	6
1.11 ₂	0.4375	0.875	1.75	3.5	7

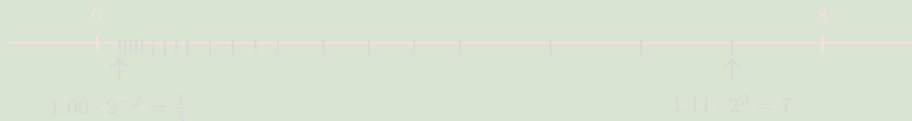


Normalisierte Darstellung

Beispiel $F^*(2, \mathbf{3}, -2, 2)$

(nur positive Zahlen)

$d_0 \bullet d_1 d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
1.00 ₂	0.25	0.5	1	2	4
1.01 ₂	0.3125	0.625	1.25	2.5	5
1.10 ₂	0.375	0.75	1.5	3	6
1.11 ₂	0.4375	0.875	1.75	3.5	7

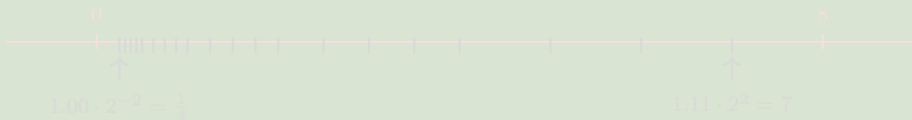


Normalisierte Darstellung

Beispiel $F^*(2, 3, -2, 2)$

(nur positive Zahlen)

$d_0 \bullet d_1 d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
1.00 ₂	0.25	0.5	1	2	4
1.01 ₂	0.3125	0.625	1.25	2.5	5
1.10 ₂	0.375	0.75	1.5	3	6
1.11 ₂	0.4375	0.875	1.75	3.5	7

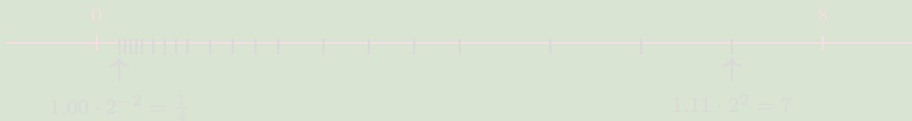


Normalisierte Darstellung

Beispiel $F^*(2, 3, -2, \mathbf{2})$

(nur positive Zahlen)

$d_0 \bullet d_1 d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = \mathbf{2}$
1.00 ₂	0.25	0.5	1	2	4
1.01 ₂	0.3125	0.625	1.25	2.5	5
1.10 ₂	0.375	0.75	1.5	3	6
1.11 ₂	0.4375	0.875	1.75	3.5	7

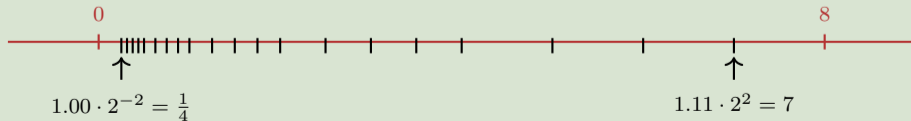


Normalisierte Darstellung

Beispiel $F^*(2, 3, -2, 2)$

(nur positive Zahlen)

$d_0 \bullet d_1 d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
1.00_2	0.25	0.5	1	2	4
1.01_2	0.3125	0.625	1.25	2.5	5
1.10_2	0.375	0.75	1.5	3	6
1.11_2	0.4375	0.875	1.75	3.5	7



Binäre und dezimale Systeme

- Intern rechnet der Computer mit $\beta = 2$
(binäres System)

Binäre und dezimale Systeme

- Intern rechnet der Computer mit $\beta = 2$
(binäres System)
- Literale und Eingaben haben $\beta = 10$
(dezimales System)

Berechnung der *Binärdarstellung*:

$$x = \sum_{i=0}^{\infty} b_i 2^{-i}$$

Berechnung der *Binärdarstellung*:

$$x = b_0.b_1b_2b_3 \dots$$

Berechnung der *Binärdarstellung*:

$$\begin{aligned}x &= b_0 \bullet b_1 b_2 b_3 \dots \\ &= b_0 + 0 \bullet b_1 b_2 b_3 \dots\end{aligned}$$

Berechnung der *Binärdarstellung*:

$$\begin{aligned}x &= b_0 \bullet b_1 b_2 b_3 \dots \\ &= b_0 + 0 \bullet b_1 b_2 b_3 \dots \\ &\implies\end{aligned}$$

Berechnung der *Binärdarstellung*:

$$x = b_0 \bullet b_1 b_2 b_3 \dots$$

$$= b_0 + 0 \bullet b_1 b_2 b_3 \dots$$

$$\implies$$

$$(x - b_0) = 0 \bullet b_1 b_2 b_3 b_4 \dots$$

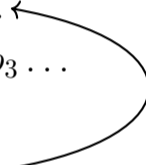
Berechnung der *Binärdarstellung*:

$$\begin{aligned}x &= b_0 \bullet b_1 b_2 b_3 \dots \\ &= b_0 + 0 \bullet b_1 b_2 b_3 \dots\end{aligned}$$

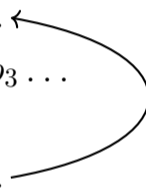
$$\implies$$

$$2 \cdot (x - b_0) = b_1 \bullet b_2 b_3 b_4 \dots$$

Berechnung der *Binärdarstellung*:

$$\begin{aligned}x &= b_0 \bullet b_1 b_2 b_3 \dots \leftarrow \\ &= b_0 + 0 \bullet b_1 b_2 b_3 \dots \\ &\implies \\ 2 \cdot (x - b_0) &= b_1 \bullet b_2 b_3 b_4 \dots\end{aligned}$$


Berechnung der *Binärdarstellung*:

$$\begin{aligned}x &= b_0 \bullet b_1 b_2 b_3 \dots \leftarrow \\ &= b_0 + 0 \bullet b_1 b_2 b_3 \dots \\ &\implies \\ 2 \cdot (x - b_0) &= b_1 \bullet b_2 b_3 b_4 \dots\end{aligned}$$


```
for (int b_0; x != 0; x = 2 * (x - b_0)) {  
    b_0 = (x >= 1);  
    std::cout << b_0;  
}
```

Beispiel (binär)

$$\begin{aligned}x &= 1.01011 \\ &= 1 + 0.01011\end{aligned}$$

\implies

$$2 \cdot (x - 1) = 0.1011$$

Beispiel (binär)

$$\begin{aligned}x &= 1.01011 \\ &= 1 + 0.01011\end{aligned}$$

\implies

$$2 \cdot (x - 1) = 0.1011$$

Beispiel (binär)

$$\begin{aligned}x &= 0.1011 \\ &= 0 + 0.1011\end{aligned}$$

\implies

$$2 \cdot (x - 0) = 1.011$$

Beispiel (binär)

$$\begin{aligned}x &= 0.1011 \\ &= 0 + 0.1011\end{aligned}$$

\implies

$$2 \cdot (x - 0) = 1.011$$

Beispiel (binär)

$$\begin{aligned}x &= 1.011 \\ &= 1 + 0.011\end{aligned}$$

\implies

$$2 \cdot (x - 1) = 0.11$$

Beispiel (binär)

$$\begin{aligned}x &= 1.011 \\ &= 1 + 0.011\end{aligned}$$

\implies

$$2 \cdot (x - 1) = 0.11$$

Beispiel (binär)

$$\begin{aligned}x &= 0.11 \\ &= 0 + 0.11 \\ &\implies \\ 2 \cdot (x - 0) &= 1.1\end{aligned}$$

Beispiel (binär)

$$\begin{aligned}x &= 0.11 \\ &= 0 + 0.11 \\ &\implies \\ 2 \cdot (x - 0) &= 1.1\end{aligned}$$

Beispiel (binär)

$$\begin{aligned}x &= 1.1 \\ &= 1 + 0.1\end{aligned}$$

\implies

$$2 \cdot (x - 1) = 1$$

Beispiel (binär)

$$\begin{aligned}x &= 1.1 \\ &= 1 + 0.1 \\ &\implies\end{aligned}$$

$$2 \cdot (x - 1) = 1$$

Beispiel (binär)

$$\begin{aligned}x &= 1 \\ &= 1 + 0 \\ &\implies\end{aligned}$$

$$2 \cdot (x - 1) = 0$$

Beispiel (binär)

$$\begin{aligned}x &= 1 \\ &= 1 + 0\end{aligned}$$

\implies

$$2 \cdot (x - 1) = 0$$

Binärdarstellung von 1.1_{10}

$$\begin{array}{r} x \quad b_i \quad x - b_i \quad 2(x - b_i) \\ \hline 1.1 \quad b_0 = 1 \end{array}$$

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$		

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$		

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$		

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$		

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2

Binärdarstellung von 1.1_{10}

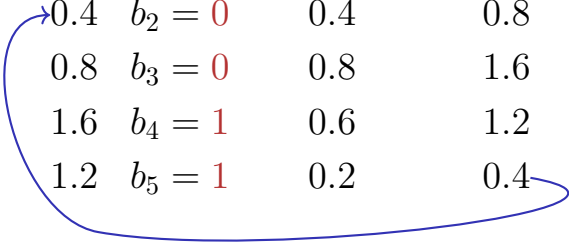
x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2
1.2	$b_5 = 1$		

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2
1.2	$b_5 = 1$	0.2	0.4

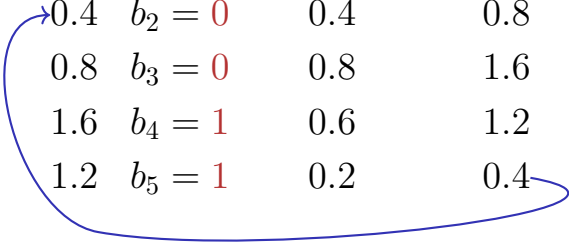
Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2
1.2	$b_5 = 1$	0.2	0.4



Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2
1.2	$b_5 = 1$	0.2	0.4



$\Rightarrow 1.\overline{00011}$, periodisch, *nicht* endlich

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich \Rightarrow Fehler bei der Konversion

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich \Rightarrow Fehler bei der Konversion
- 1.1f und 0.1f: *Approximationen* von 1.1 und 0.1

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich \Rightarrow Fehler bei der Konversion
- `1.1f` und `0.1f`: *Approximationen* von 1.1 und 0.1
- In `diff.cpp`: `1.1 - 1.0 \neq 0.1`

Binärdarstellungen von 1.1 und 0.1

auf meinem Computer:

$$\begin{aligned} \mathbf{1.1} &= \underline{1.100000000000000000}888178\dots \\ \mathbf{1.1f} &= \underline{1.1000000}238418\dots \end{aligned}$$

Rechnen mit Fließkommazahlen

ist fast so einfach wie mit ganzen Zahlen.

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 1.011 \cdot 2^{-1} \end{array}$$

Rechnen mit Fließkommazahlen

Beispiel ($\beta = 2, p = 4$):

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 1.011 \cdot 2^{-1} \end{array}$$

1. Exponenten anpassen durch Denormalisieren einer Zahl

Rechnen mit Fließkommazahlen

Beispiel ($\beta = 2, p = 4$):

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \end{array} \checkmark$$

1. Exponenten anpassen durch Denormalisieren einer Zahl

Rechnen mit Fließkommazahlen

Beispiel ($\beta = 2, p = 4$):

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \\ \hline \end{array}$$

2. Binäre Addition der Signifikanden

Rechnen mit Fließkommazahlen

Beispiel ($\beta = 2, p = 4$):

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \\ \hline = 100.101 \cdot 2^{-2} \checkmark \end{array}$$

2. Binäre Addition der Signifikanden

Rechnen mit Fließkommazahlen

Beispiel ($\beta = 2, p = 4$):

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \\ \hline = 100.101 \cdot 2^{-2} \end{array}$$

3. Renormalisierung

Rechnen mit Fließkommazahlen

Beispiel ($\beta = 2, p = 4$):

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \\ \hline = 1.00101 \cdot 2^0 \checkmark \end{array}$$

3. Renormalisierung

Rechnen mit Fließkommazahlen

Beispiel ($\beta = 2, p = 4$):

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \\ \hline = 1.00101 \cdot 2^0 \end{array}$$

4. Runden auf p signifikante Stellen, falls nötig

Rechnen mit Fließkommazahlen

Beispiel ($\beta = 2, p = 4$):

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \\ \hline = 1.001 \cdot 2^0 \checkmark \end{array}$$

4. Runden auf p signifikante Stellen, falls nötig

Der IEEE Standard 754

legt Fließkommazahlensysteme und deren Rundungsverhalten fest und wird fast überall benutzt

- Single precision (**float**) Zahlen:

$$F^*(2, 24, -126, 127) \text{ (32 bit)}$$

Der IEEE Standard 754

legt Fließkommazahlensysteme und deren Rundungsverhalten fest und wird fast überall benutzt

- Single precision (**float**) Zahlen:

$$F^*(2, 24, -126, 127) \text{ (32 bit)}$$

- Double precision (**double**) Zahlen:

$$F^*(2, 53, -1022, 1023) \text{ (64 bit)}$$

Der IEEE Standard 754

legt Fließkommazahlensysteme und deren Rundungsverhalten fest und wird fast überall benutzt

- Single precision (**float**) Zahlen:

$$F^*(2, 24, -126, 127) \text{ (32 bit)} \quad \text{plus } 0, \infty, \dots$$

- Double precision (**double**) Zahlen:

$$F^*(2, 53, -1022, 1023) \text{ (64 bit)} \quad \text{plus } 0, \infty, \dots$$

- Alle arithmetischen Operationen runden das *exakte* Ergebnis auf die nächste darstellbare Zahl

Beispiel: 32-bit Darstellung einer Fließkommazahl



± Exponent

Mantisse

± $2^{-126}, \dots, 2^{127}$
 $0, \infty, \dots$

1.000000000000000000000000000000
...
1.111111111111111111111111111111

Regel 1

Teste keine gerundeten Fließkommazahlen auf Gleichheit!

Regel 1

Teste keine gerundeten Fließkommazahlen auf Gleichheit!

```
for (float i = 0.1; i != 1.0; i += 0.1)
    std::cout << i << "\n";
```

Regel 1

Teste keine gerundeten Fließkommazahlen auf Gleichheit!

```
for (float i = 0.1; i != 1.0; i += 0.1)
    std::cout << i << "\n";
```

Endlosschleife, weil i niemals exakt 1 ist!

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

$$\begin{array}{r} 1.000 \cdot 2^5 \\ + 1.000 \cdot 2^0 \end{array}$$

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

$$\begin{aligned} & 1.000 \cdot 2^5 \\ & + 1.000 \cdot 2^0 \\ & = 1.00001 \cdot 2^5 \end{aligned}$$

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

$$\begin{aligned} & 1.000 \cdot 2^5 \\ & + 1.000 \cdot 2^0 \\ & = 1.00001 \cdot 2^5 \\ & \text{„=“ } 1.000 \cdot 2^5 \text{ (Rundung auf 4 Stellen)} \end{aligned}$$

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

$$\begin{aligned} & 1.000 \cdot 2^5 \\ & + 1.000 \cdot 2^0 \\ & = 1.00001 \cdot 2^5 \\ & \text{„=“ } 1.000 \cdot 2^5 \text{ (Rundung auf 4 Stellen)} \end{aligned}$$

Addition von 1 hat keinen Effekt!

- Die n -te Harmonische Zahl ist

$$H_n = \sum_{i=1}^n \frac{1}{i}$$

- Die n -te Harmonische Zahl ist

$$H_n = \sum_{i=1}^n \frac{1}{i} \approx \ln n.$$

- Die n -te Harmonische Zahl ist

$$H_n = \sum_{i=1}^n \frac{1}{i} \approx \ln n.$$

- Diese Summe kann vorwärts oder rückwärts berechnet werden, was mathematisch gesehen natürlich äquivalent ist.

```
std::cout << "Compute H_n for n =? ";
unsigned int n;
std::cin >> n;

float fs = 0;
for (unsigned int i = 1; i <= n; ++i)
    fs += 1.0f / i;
std::cout << "Forward sum = " << fs << "\n";

float bs = 0;
for (unsigned int i = n; i >= 1; --i)
    bs += 1.0f / i;
std::cout << "Backward sum = " << bs << "\n";
```

```
std::cout << "Compute H_n for n =? ";  
unsigned int n;  
std::cin >> n;
```

Eingabe: **1000000**

```
float fs = 0;  
for (unsigned int i = 1; i <= n; ++i)  
    fs += 1.0f / i;  
std::cout << "Forward sum = " << fs << "\n";
```

Vorwärts: **15.4037**

```
float bs = 0;  
for (unsigned int i = n; i >= 1; --i)  
    bs += 1.0f / i;  
std::cout << "Backward sum = " << bs << "\n";
```

Rückwärts: **16.686**

```
std::cout << "Compute H_n for n =? ";  
unsigned int n;  
std::cin >> n;
```

Eingabe: **10000000**

```
float fs = 0;  
for (unsigned int i = 1; i <= n; ++i)  
    fs += 1.0f / i;  
std::cout << "Forward sum = " << fs << "\n";
```

Vorwärts: **15.4037**

```
float bs = 0;  
for (unsigned int i = n; i >= 1; --i)  
    bs += 1.0f / i;  
std::cout << "Backward sum = " << bs << "\n";
```

Rückwärts: **18.8079**

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist „richtig“ falsch.

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist „richtig“ falsch.
- Die Rückwärtssumme approximiert H_n gut.

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist „richtig“ falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist „richtig“ falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

- Bei $1 + 1/2 + 1/3 + \dots$ sind späte Terme zu klein, um noch beizutragen.

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist „richtig“ falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

- Bei $1 + 1/2 + 1/3 + \dots$ sind späte Terme zu klein, um noch beizutragen.
- Problematik wie bei $2^5 + 1$ „=“ 2^5

Regel 3

Subtrahiere keine zwei Zahlen sehr ähnlicher Grösse!

Auslöschungsproblematik, siehe Skript.

Literatur

David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic (1991)



Randy Glasbergen, 1996

9. Funktionen I

Funktionsdefinitionen- und Aufrufe, Auswertung von Funktionsaufrufen, Der Typ **void**

Potenzberechnung

```
double a;
int n;
std::cin >> a; // Eingabe a
std::cin >> n; // Eingabe n

double result = 1.0;
if (n < 0) { //  $a^n = (1/a)^{-n}$ 
    a = 1.0/a;
    n = -n;
}
for (int i = 0; i < n; ++i)
    result *= a;

std::cout << a << "^" << n << " = " << result << ".\n";
```

Potenzberechnung

```
double a;  
int n;  
std::cin >> a; // Eingabe a  
std::cin >> n; // Eingabe n
```

```
double result = 1.0;  
if (n < 0) { //  $a^n = (1/a)^{-n}$   
    a = 1.0/a;  
    n = -n;  
}  
for (int i = 0; i < n; ++i)  
    result *= a;
```

```
std::cout << a << "^" << n << " = " << result << ".\n";
```


Potenzberechnung

```
double a;  
int n;  
std::cin >> a; // Eingabe a  
std::cin >> n; // Eingabe n
```

```
double result = 1.0;  
if (n < 0) { //  $a^n = (1/a)^{-n}$   
    a = 1.0/a;  
    n = -n;  
}  
for (int i = 0; i < n; ++i)  
    result *= a;
```

 "Funktion pow"

```
std::cout << a << "^" << n << " = " << pow(a,n) << ".\n";
```

Funktion zur Potenzberechnung

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow(double b, int e)
{
    double result = 1.0;
    if (e < 0) { // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i)
        result *= b;
    return result;
}
```

Funktion zur Potenzberechnung

```
double pow(double b, int e){...}
```

Funktion zur Potenzberechnung

```
// Prog: callpow.cpp
// Define and call a function for computing powers.
#include <iostream>
```

```
double pow(double b, int e){...}
```

```
int main()
{
    std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
    std::cout << pow( 1.5, 2) << "\n"; // outputs 2.25
    std::cout << pow(-2.0, 9) << "\n"; // outputs -512

    return 0;
}
```

Funktionsdefinitionen

T fname (T_1 pname₁, T_2 pname₂, ..., T_N pname_N)
block

Funktionsname



Funktionsdefinitionen

Rückgabety

T fname (T_1 pname₁, T_2 pname₂, ..., T_N pname_N)
block

Funktionsname

Funktionsdefinitionen

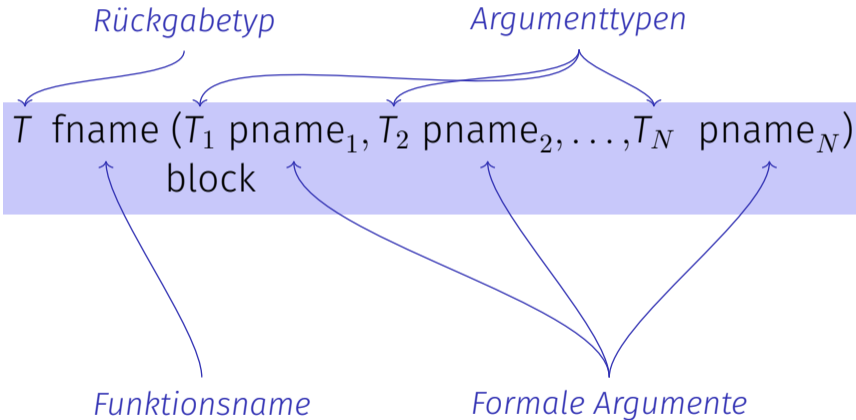
Rückgabebetyp

T fname (T_1 pname₁, T_2 pname₂, ..., T_N pname_N)
block

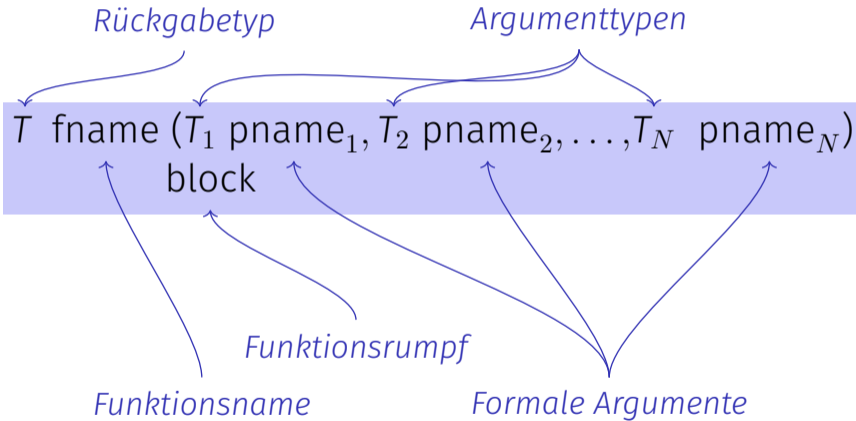
Funktionsname

Formale Argumente

Funktionsdefinitionen



Funktionsdefinitionen



Xor

```
// post: returns l XOR r
bool Xor(bool l, bool r)
{
    return l && !r || !l && r;
}
```

Harmonic

```
// PRE: n >= 0
// POST: returns nth harmonic number
//       computed with backward sum
float Harmonic(int n)
{
    float res = 0;
    for (unsigned int i = n; i >= 1; --i)
        res += 1.0f / i;
    return res;
}
```

min

```
// POST: returns the minimum of a and b
int min(int a, int b)
{
    if (a<b)
        return a;
    else
        return b;
}
```

Funktionsaufrufe

$fname (expression_1, expression_2, \dots, expression_N)$

- Alle Aufrufargumente müssen konvertierbar sein in die entsprechenden Argumenttypen.

Funktionsaufrufe

$fname (expression_1, expression_2, \dots, expression_N)$

- Alle Aufrufargumente müssen konvertierbar sein in die entsprechenden Argumenttypen.
- Der Funktionsaufruf selbst ist ein Ausdruck vom Rückgabotyp.

Funktionsaufrufe

$fname (expression_1, expression_2, \dots, expression_N)$

- Alle Aufrufargumente müssen konvertierbar sein in die entsprechenden Argumenttypen.
- Der Funktionsaufruf selbst ist ein Ausdruck vom Rückgabetyt.

Beispiel: **pow(a,n)**: Ausdruck vom Typ **double**

Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

- Aufrufargumente sind R-Werte
 ↪ *call-by-value* (auch *pass-by-value*), dazu gleich mehr
- Funktionsaufruf selbst ist R-Wert.

Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

- Aufrufargumente sind R-Werte
 \hookrightarrow *call-by-value* (auch *pass-by-value*), dazu gleich mehr
- Funktionsaufruf selbst ist R-Wert.

$fname: R\text{-Wert} \times R\text{-Wert} \times \dots \times R\text{-Wert} \longrightarrow R\text{-Wert}$

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```

...

```
pow (2.0, -2)
```

Auswertung Funktionsaufruf

Aufruf von pow

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){  
    assert (e >= 0 || b != 0);  
    double result = 1.0;  
    if (e<0) {  
        //  $b^e = (1/b)^{-e}$   
        b = 1.0/b;  
        e = -e;  
    }  
    for (int i = 0; i < e ; ++i)  
        result * = b;  
    return result;  
}
```

b=2.0, e=-2

...

```
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){  
    assert (e >= 0 || b != 0);  
    double result = 1.0;  
    if (e<0) {  
        //  $b^e = (1/b)^{-e}$   
        b = 1.0/b;  
        e = -e;  
    }  
    for (int i = 0; i < e ; ++i)  
        result * = b;  
    return result;  
}
```

Diagram illustrating the evaluation of the function call `pow(2.0, -2)`. The arguments `2.0` and `-2` are passed to the parameters `b` and `e` respectively. The assertion `assert (e >= 0 || b != 0);` is satisfied, resulting in `// ok`.

```
...  
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```



result=1.0

...

```
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```



e == -2

...

```
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```



b=0.5

...

```
pow (2.0, -2)
```


Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```



e=2

...

```
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```



i=0

...

```
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```

i=0

result=0.5

...

```
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```

...

```
pow (2.0, -2)
```



i=1

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```

i=1

result=0.25

...

```
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```

...

```
pow (2.0, -2)
```



i=2

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```

→ result=0.25

...

```
pow (2.0, -2)
```

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```

...

pow (2.0, -2)

result=0.25

Rückgabe

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```

...

pow (2.0, -2)

Rückgabe

Wert: 0.25

Auswertung Funktionsaufruf

```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        //  $b^e = (1/b)^{-e}$ 
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}
```

...

pow (2.0, -2)

Wert: 0.25

Gültigkeit formaler Argumente

```
int main(){  
    double b = 2.0;  
    int e = -2;  
    double z = pow(b, e);  
  
    std::cout << z; // 0.25  
    std::cout << b; // 2  
    std::cout << e; // -2  
    return 0;  
}
```

Gültigkeit formaler Argumente

```
double pow(double b, int e){
    double r = 1.0;
    if (e<0) {
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        r * = b;
    return r;
}
```

```
int main(){
    double b = 2.0;
    int e = -2;
    double z = pow(b, e);

    std::cout << z; // 0.25
    std::cout << b; // 2
    std::cout << e; // -2
    return 0;
}
```

Gültigkeit formaler Argumente

```
double pow(double b, int e){  
    double r = 1.0;  
    if (e<0) {  
        b = 1.0/b;  
        e = -e;  
    }  
    for (int i = 0; i < e ; ++i)  
        r * = b;  
    return r;  
}
```

```
int main(){  
    double b = 2.0;  
    int e = -2;  
    double z = pow(b, e);  
  
    std::cout << z; // 0.25  
    std::cout << b; // 2  
    std::cout << e; // -2  
    return 0;  
}
```

Nicht die formalen Argumente **b** und **e** von **pow**, sondern die hier definierten Variablen lokal zum Rumpf von **main**

Der Typ void

```
// POST: "(i, j)" has been written to standard output
???? print_pair(int i, int j) {
    std::cout << "(" << i << ", " << j << ")\n";
}

int main() {
    print_pair(3,4); // outputs (3, 4)
    return 0;
}
```

Der Typ void

```
// POST: "(i, j)" has been written to standard output
void print_pair(int i, int j) {
    std::cout << "(" << i << ", " << j << ")\n";
}

int main() {
    print_pair(3,4); // outputs (3, 4)
    return 0;
}
```

Der Typ `void`

- Fundamentaler Typ mit leerem Wertebereich

Der Typ `void`

- Fundamentaler Typ mit leerem Wertebereich
- Verwendung als Rückgabetyt für Funktionen, die *nur* einen Effekt haben

void-Funktionen

- benötigen kein **return**.
- Ausführung endet, wenn Ende des Funktionsrumpfes erreicht wird oder
- **return;** erreicht wird

Funktionen und return

Falsch:

```
bool compare(float x, float y) {  
    float delta = x - y;  
    if (delta*delta < 0.001f) return true;  
}
```

Funktionen und `return`

Das Verhalten einer Funktion mit Rückgabetyt ungleich `void` ist **undefiniert** wenn das Ende des Funktionsrumpfes ohne `return` Anweisung erreicht wird.

Falsch:

```
bool compare(float x, float y) {  
    float delta = x - y;  
    if (delta*delta < 0.001f) return true;  
}
```

Der Wert von `compare(10,20)` ist hier undefiniert

Funktionen und `return`

Das Verhalten einer Funktion mit Rückgabebetyp ungleich `void` ist **undefiniert** wenn das Ende des Funktionsrumpfes ohne `return` Anweisung erreicht wird.

Besser:

```
bool compare(float x, float y) {  
    float delta = x - y;  
    if (delta*delta < 0.001f)  
        return true;  
    else  
        return false;  
}
```

Alle Ausführungspfade erreichen ein **return**

Funktionen und return

Das Verhalten einer Funktion mit Rückgabetyt ungleich **void** ist **undefiniert** wenn das Ende des Funktionsrumpfes ohne **return** Anweisung erreicht wird.

Noch besser (weil einfacher)

```
bool compare(float x, float y) {  
    float delta = x - y;  
    return delta*delta < 0.001f;  
}
```

10. Funktionen II

Vor- und Nachbedingungen Stepwise Refinement,
Gültigkeitsbereich, Bibliotheken, Standardfunktionen

Vorbedingungen

Vorbedingung (precondition):

- Was muss bei Funktionsaufruf gelten?
- Spezifiziert *Definitionsereich* der Funktion.

Vorbedingungen

Vorbedingung (precondition):

- Was muss bei Funktionsaufruf gelten?
- Spezifiziert *Definitionsereich* der Funktion.

0^e ist für $e < 0$ undefiniert

```
// PRE: e >= 0 || b != 0.0
```

Nachbedingungen

Nachbedingung (postcondition):

- Was gilt nach Funktionsaufruf?
- Spezifiziert *Wert* und *Effekt* des Funktionsaufrufes.

Nachbedingungen

Nachbedingung (postcondition):

- Was gilt nach Funktionsaufruf?
- Spezifiziert *Wert* und *Effekt* des Funktionsaufrufes.

Hier nur Wert, kein Effekt.

```
// POST: return value is  $b^e$ 
```

Vor- und Nachbedingungen

- sollten korrekt sein:

Vor- und Nachbedingungen

- sollten korrekt sein:
- *Wenn* die Vorbedingung beim Funktionsaufruf gilt, *dann* gilt auch die Nachbedingung nach dem Funktionsaufruf.

Vor- und Nachbedingungen

- sollten korrekt sein:
- *Wenn* die Vorbedingung beim Funktionsaufruf gilt, *dann* gilt auch die Nachbedingung nach dem Funktionsaufruf.

Funktion **pow**: funktioniert für alle Basen $b \neq 0$

Fromme Lügen...

```
// PRE: e >= 0 || b != 0.0  
// POST: return value is be
```

ist formal inkorrekt

Fromme Lügen...

```
// PRE: e >= 0 || b != 0.0  
// POST: return value is be
```

ist formal inkorrekt:

- Überlauf, falls e oder b zu gross sind
- b^e vielleicht nicht als double Wert darstellbar (Löcher im Wertebereich)

Fromme Lügen...sind erlaubt.

```
// PRE: e >= 0 || b != 0.0  
// POST: return value is b^e
```

Mathematische Bedingungen als Kompromiss zwischen formaler Korrektheit und lascher Praxis.

Prüfen von Vorbedingungen...

- Vorbedingungen sind nur Kommentare.

Prüfen von Vorbedingungen...

- Vorbedingungen sind nur Kommentare.
- Wie können wir *sicherstellen*, dass sie beim Funktionsaufruf gelten?

...mit Assertions

```
#include <cassert>
...
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow(double b, int e) {
    assert (e >= 0 || b != 0);
    double result = 1.0;
    ...
}
```

Nachbedingungen mit Assertions

- Das Ergebnis „komplizierter“ Berechnungen ist oft einfach zu prüfen.

Nachbedingungen mit Assertions

- Das Ergebnis „komplizierter“ Berechnungen ist oft einfach zu prüfen.
- Dann lohnt sich der Einsatz von assert für die Nachbedingung

Nachbedingungen mit Assertions

- Das Ergebnis „komplizierter“ Berechnungen ist oft einfach zu prüfen.
- Dann lohnt sich der Einsatz von assert für die Nachbedingung

```
// PRE: the discriminant p*p/4 - q is nonnegative
// POST: returns larger root of the polynomial x^2 + p x + q
double root(double p, double q)
{
    assert(p*p/4 >= q); // precondition
    double x1 = - p/2 + sqrt(p*p/4 - q);
    assert(equals(x1*x1+p*x1+q,0)); // postcondition
    return x1;
}
```

Stepwise Refinement

Einfache Programmier- technik zum Lösen komplexer Probleme

Niklaus Wirth. Program development by
stepwise refinement. Commun. ACM 14,
4, 1971

Education P. Wegner
Editor

Program Development by Stepwise Refinement

Niklaus Wirth
Eidgenössische Technische Hochschule
Zürich, Switzerland

The creative activity of programming—to be distinguished from coding—is usually taught by examples serving to exhibit certain techniques. It is here considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures. The process of successive refinement of specifications is illustrated by a short but nontrivial example, from which a number of conclusions are drawn regarding the art and the instruction of programming.

Key Words and Phrases: education in programming, programming techniques, stepwise program construction

CR Categories: 1.50, 4.0

1. Introduction

Programming is usually taught by examples. Experience shows that the success of a programming course critically depends on the choice of these examples. Unfortunately, they are too often selected with the prime intent to demonstrate what a computer can do. Instead, a main criterion for selection should be their suitability to exhibit certain widely applicable techniques. Furthermore, examples of programs are commonly presented as finished "products" followed by explanations of their purpose and their linguistic details. But active programming consists of the design of new programs, rather than contemplation of old programs. As a consequence of these teaching methods, the student obtains the impression that programming consists mainly of mastering a language (with all the peculiarities and intricacies so abundant in modern PL's) and relying on one's intuition to somehow transform ideas into finished programs. Clearly, programming courses should teach methods of design and construction, and the selected examples should be such that a gradual development can be nicely demonstrated.

This paper deals with a single example chosen with

these two purposes in mind. Some well-known techniques are briefly demonstrated and motivated (strategy of preselection, stepwise construction of trial solutions, introduction of auxiliary data, recursion), and the program is gradually developed in a sequence of refinement steps.

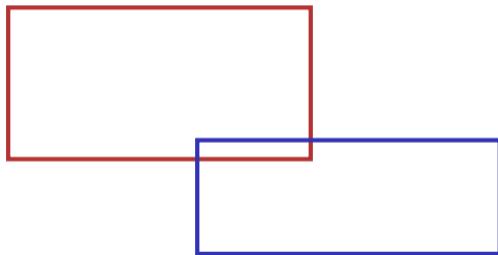
In each step, one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of an underlying computer or programming language, and must therefore be guided by the facilities available on that computer or language. The result of the execution of a program is expressed in terms of data, and it may be necessary to introduce further data for communication between the obtained subtasks or instructions. As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine program and data specifications in parallel.

Every refinement step implies some design decisions. It is important that these decisions be made explicit, and that the programmer be aware of the underlying criteria and of the existence of alternative solutions. The possible solutions to a given problem emerge as the leaves of a tree, each node representing a point of deliberation and decision. Subtrees may be considered as families of solutions with certain common characteristics and structures. The notion of such a tree may be particularly helpful in the situation of changing purpose and environment to which a program may sometime have to be adapted.

A guideline in the process of stepwise refinement should be the principle to decompose decisions as much as possible, to untangle aspects which are only seemingly interdependent, and to defer those decisions which concern details of representation as long as possible. This

Beispielproblem

Finde heraus, ob sich zwei Rechtecke schneiden!



Top-Down Ansatz

- Formulierung einer groben Lösung mit Hilfe von
 - Kommentaren
 - fiktiven Funktionen

Top-Down Ansatz

- Formulierung einer groben Lösung mit Hilfe von
 - Kommentaren
 - fiktiven Funktionen
- Wiederholte Verfeinerung:
 - Kommentare \rightarrow Programmtext
 - fiktive Funktionen \rightarrow Funktionsdefinitionen

Grobe Lösung

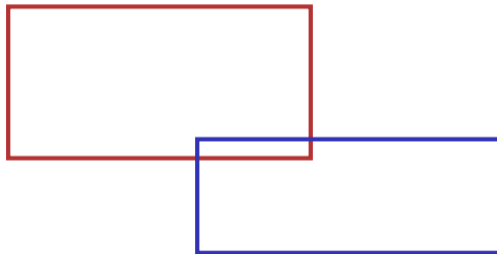
```
int main()
{
    // Eingabe Rechtecke

    // Schnitt?

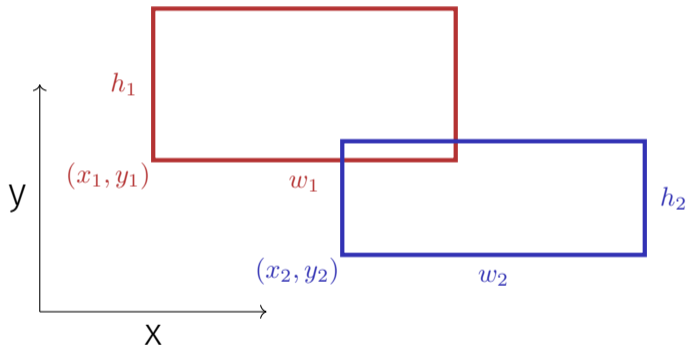
    // Ausgabe der Loesung

    return 0;
}
```

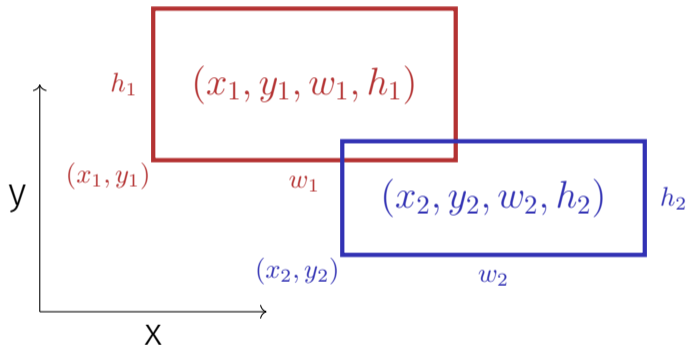
Verfeinerung 1: Eingabe Rechtecke



Verfeinerung 1: Eingabe Rechtecke

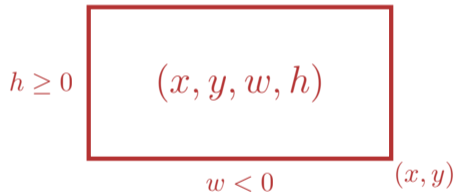


Verfeinerung 1: Eingabe Rechtecke



Verfeinerung 1: Eingabe Rechtecke

Breite w und/oder Höhe h dürfen negativ sein!



Verfeinerung 1: Eingabe Rechtecke

```
int main()
{
    std::cout << "Enter two rectangles [x y w h each] \n";
    int x1, y1, w1, h1;
    std::cin >> x1 >> y1 >> w1 >> h1;
    int x2, y2, w2, h2;
    std::cin >> x2 >> y2 >> w2 >> h2;

    // Schnitt?

    // Ausgabe der Loesung

    return 0;
}
```

Verfeinerung 2: Schnitt? und Ausgabe

```
int main()
{
    Eingabe Rectecke ✓

    bool clash = rectangles_intersect(x1,y1,w1,h1,x2,y2,w2,h2);

    if (clash)
        std::cout << "intersection!\n";
    else
        std::cout << "no intersection!\n";

    return 0;
}
```

Verfeinerung 3: Schnittfunktion...

```
bool rectangles_intersect(int x1, int y1, int w1, int h1,  
                          int x2, int y2, int w2, int h2)  
{  
    return false; // todo  
}
```

```
int main() {
```

Eingabe Rechtecke ✓

Schnitt? ✓

Ausgabe der Loesung ✓

```
    return 0;
```

```
}
```

Verfeinerung 3: Schnittfunktion...

```
bool rectangles_intersect(int x1, int y1, int w1, int h1,  
                          int x2, int y2, int w2, int h2)  
{  
    return false; // todo  
}
```

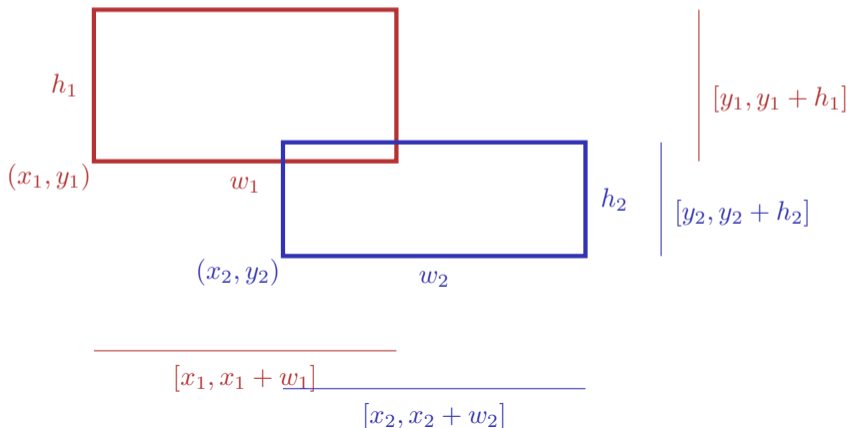
Funktion main ✓

Verfeinerung 3: ...mit PRE und POST!

```
// PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles,  
//       where w1, h1, w2, h2 may be negative.  
// POST: returns true if (x1, y1, w1, h1) and  
//       (x2, y2, w2, h2) intersect  
bool rectangles_intersect(int x1, int y1, int w1, int h1,  
                          int x2, int y2, int w2, int h2)  
{  
    return false; // todo  
}
```

Verfeinerung 4: Intervallschnitt

Zwei Rechtecke schneiden sich genau dann, wenn sich ihre x - und y -Intervalle schneiden.



Verfeinerung 4: Intervallschnitte

```
// PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles, where
//       w1, h1, w2, h2 may be negative.
// POST: returns true if (x1, y1, w1, h1), (x2, y2, w2, h2) intersect
bool rectangles_intersect(int x1, int y1, int w1, int h1,
                          int x2, int y2, int w2, int h2)
{
    return intervals_intersect(x1, x1 + w1, x2, x2 + w2)
        && intervals_intersect(y1, y1 + h1, y2, y2 + h2);
}
```

Verfeinerung 4: Intervallschnitte

```
// PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles, where
//       w1, h1, w2, h2 may be negative.
// POST: returns true if (x1, y1, w1, h1), (x2, y2, w2, h2) intersect
bool rectangles_intersect(int x1, int y1, int w1, int h1,
                          int x2, int y2, int w2, int h2)
{
    return intervals_intersect(x1, x1 + w1, x2, x2 + w2)
        && intervals_intersect(y1, y1 + h1, y2, y2 + h2); ✓
}
```


Verfeinerung 4: Intervallschnitte

```
// PRE: [a1, b1], [a2, b2] are (generalized) intervals,  
//       with [a,b] := [b,a] if a>b  
// POST: returns true if [a1, b1],[a2, b2] intersect  
bool intervals_intersect(int a1, int b1, int a2, int b2)  
{  
    return false; // todo  
}
```

Funktion rectangles_intersect ✓

Funktion main ✓

Verfeinerung 5: Min und Max

```
// PRE: [a1, b1], [a2, b2] are (generalized) intervals,  
//       with [a,b] := [b,a] if a>b  
// POST: returns true if [a1, b1],[a2, b2] intersect  
bool intervals_intersect(int a1, int b1, int a2, int b2)  
{  
    return max(a1, b1) >= min(a2, b2)  
        && min(a1, b1) <= max(a2, b2);  
}
```

Verfeinerung 5: Min und Max

```
// PRE: [a1, b1], [a2, b2] are (generalized) intervals,  
//       with [a,b] := [b,a] if a>b  
// POST: returns true if [a1, b1],[a2, b2] intersect  
bool intervals_intersect(int a1, int b1, int a2, int b2)  
{  
    return max(a1, b1) >= min(a2, b2)  
        && min(a1, b1) <= max(a2, b2); ✓  
}
```

Verfeinerung 5: Min und Max

```
// POST: the maximum of x and y is returned
int max(int x, int y){
    if (x>y) return x; else return y;
}
```

```
// POST: the minimum of x and y is returned
int min(int x, int y){
    if (x<y) return x; else return y;
}
```

Funktion intervals_intersect ✓

Funktion rectangles_intersect ✓

Funktion main ✓

Verfeinerung 5: Min und Max

```
// POST: the maximum of x and y is returned
int max(int x, int y){
    if (x>y) return x; else return y;
}
```

gibt es schon in der Standardbibliothek

```
// POST: the minimum of x and y is returned
int min(int x, int y){
    if (x<y) return x; else return y;
}
```

Funktion intervals_intersect ✓

Funktion rectangles_intersect ✓

Funktion main ✓

Nochmal zurück zu Intervallen

```
// PRE: [a1, b1], [a2, h2] are (generalized) intervals,  
//       with [a,b] := [b,a] if a>b  
// POST: returns true if [a1, b1],[a2, b2] intersect  
bool intervals_intersect(int a1, int b1, int a2, int b2)  
{  
    return std::max(a1, b1) >= std::min(a2, b2)  
        && std::min(a1, b1) <= std::max(a2, b2); ✓  
}
```

Das haben wir schrittweise erreicht!

```
#include <iostream>
#include <algorithm>

// PRE: [a1, b1], [a2, h2] are (generalized) intervals,
//      with [a,b] := [b,a] if a>b
// POST: returns true if [a1, b1],[a2, b2] intersect
bool intervals_intersect(int a1, int b1, int a2, int b2)
{
    return std::max(a1, b1) >= std::min(a2, b2)
        && std::min(a1, b1) <= std::max(a2, b2);
}

// PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles, where
//      w1, h1, w2, h2 may be negative.
// POST: returns true if (x1, y1, w1, h1),(x2, y2, w2, h2) intersect
bool rectangles_intersect(int x1, int y1, int w1, int h1,
                          int x2, int y2, int w2, int h2)
{
    return intervals_intersect(x1, x1 + w1, x2, x2 + w2)
        && intervals_intersect(y1, y1 + h1, y2, y2 + h2);
}
```

```
int main ()
{
    std::cout << "Enter two rectangles [x y w h each]\n";
    int x1, y1, w1, h1;
    std::cin >> x1 >> y1 >> w1 >> h1;
    int x2, y2, w2, h2;
    std::cin >> x2 >> y2 >> w2 >> h2;
    bool clash = rectangles_intersect(x1,y1,w1,h1,x2,y2,w2,h2);
    if (clash)
        std::cout << "intersection!\n";
    else
        std::cout << "no intersection!\n";
    return 0;
}
```

Ergebnis

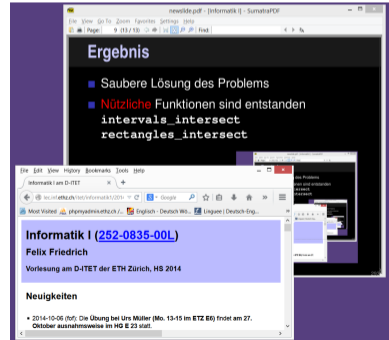
- Saubere Lösung des Problems
- Nützliche Funktionen sind entstanden

`intervals_intersect`

`rectangles_intersect`

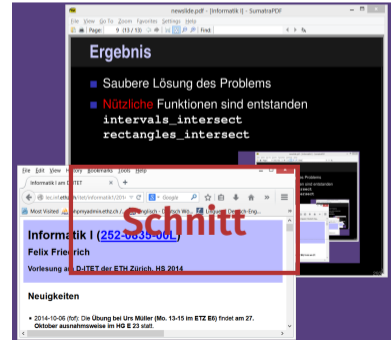
Ergebnis

- Saubere Lösung des Problems
- **Nützliche** Funktionen sind entstanden
`intervals_intersect`
`rectangles_intersect`



Ergebnis

- Saubere Lösung des Problems
- **Nützliche** Funktionen sind entstanden
`intervals_intersect`
`rectangles_intersect`



Wo darf man eine Funktion benutzen?

```
#include <iostream>
```

```
int main()  
{  
    std::cout << f(1); // Fehler: f undeklariert  
    return 0;  
}
```

Gültigkeit f

```
int f(int i) // Gültigkeitsbereich von f ab hier  
{  
    return i;  
}
```

Gültigkeitsbereich einer Funktion

- ist der Teil des Programmes, in dem die Funktion aufgerufen werden kann

Gültigkeitsbereich einer Funktion

- ist der Teil des Programmes, in dem die Funktion aufgerufen werden kann

Erweiterung durch **Deklaration** einer Funktion: wie Definition aber ohne {...}.

```
double pow(double b, int e);
```

So geht's also nicht...

```
#include <iostream>
```

```
int main()
{
    std::cout << f(1); // Fehler: f undeklariert
    return 0;
}
```

Gültigkeit f

```
int f(int i) // Gültigkeitsbereich von f ab hier
{
    return i;
}
```

...aber so!

```
#include <iostream>
int f(int i); // Gultigkeitsbereich von f ab hier

int main()
{
    std::cout << f(1);
    return 0;
}

int f(int i)
{
    return i;
}
```

Forward Declarations, wozu?

Funktionen, die sich gegenseitig aufrufen:

```
int f(...) // f ab hier gültig
{
    g(...) // g ist undeklariert
}
int g(...) // g ab hier gültig!
{
    f(...) // ok
}
```

The diagram illustrates the validity of functions f and g. A blue arrow labeled "Gültigkeit g" points downwards from the definition of g to the call of g in f. A red arrow labeled "Gültigkeit f" points downwards from the definition of f to the call of f in g.

Forward Declarations, wozu?

Funktionen, die sich gegenseitig aufrufen:

```
int g(...); // g ab hier gültig

int f(...) // f ab hier gültig
{
    g(...) // ok
}

int g(...)
{
    f(...) // ok
}
```

Wiederverwendbarkeit

- Funktionen wie `rectangles_intersect` und `pow` sind in vielen Programmen nützlich.

Wiederverwendbarkeit

- Funktionen wie `rectangles_intersect` und `pow` sind in vielen Programmen nützlich.
- „Lösung:“ Funktion einfach ins Hauptprogramm hineinkopieren, wenn wir sie brauchen!

Level 1: Auslagern der Funktion

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow(double b, int e)
{
    double result = 1.0;
    if (e < 0) { // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i)
        result *= b;
    return result;
}
```

Level 1: Auslagern der Funktion

```
double pow(double b, int e); in  
separater Datei mymath.cpp
```

Level 1: Inkludieren der Funktion

```
// Prog: callpow2.cpp  
// Call a function for computing powers.
```

```
#include <iostream>  
#include "mymath.cpp"
```

```
int main()  
{  
    std::cout << pow( 2.0, -2) << "\n";  
    std::cout << pow( 1.5, 2) << "\n";  
    std::cout << pow( 5.0, 1) << "\n";  
    std::cout << pow(-2.0, 9) << "\n";  
  
    return 0;  
}
```

Level 1: Inkludieren der Funktion

```
// Prog: callpow2.cpp  
// Call a function for computing powers.
```

```
#include <iostream>  
#include "mymath.cpp" ← im Arbeitsverzeichnis
```

```
int main()  
{  
    std::cout << pow( 2.0, -2) << "\n";  
    std::cout << pow( 1.5, 2) << "\n";  
    std::cout << pow( 5.0, 1) << "\n";  
    std::cout << pow(-2.0, 9) << "\n";  
  
    return 0;  
}
```

Nachteil des Inkludierens

- `#include` kopiert die Datei (`mymath.cpp`) in das Hauptprogramm (`callpow2.cpp`).

Nachteil des Inkludierens

- `#include` kopiert die Datei (`mymath.cpp`) in das Hauptprogramm (`callpow2.cpp`).
- Der Compiler muss die Funktionsdefinition für jedes Programm neu übersetzen.



```
Terminal — tcsh8.5 — 80x24
Shabdas-iMac:~ admin$ sudo port install amarok
--> Fetching pkgconfig
--> Attempting to fetch pkg-config-0.25.tar.gz from http://aarnet.au.distfiles
.macports.org/pub/macports/mpdistfiles/pkgconfig
--> Verifying checksum(s) for pkgconfig
--> Extracting pkgconfig
--> Applying patches to pkgconfig
--> Configuring pkgconfig
--> Building pkgconfig
--> Staging pkgconfig into destroot
--> Installing pkgconfig @0.25_1
--> Deactivating pkgconfig @0.23_1
--> Activating pkgconfig @0.25_1
--> Cleaning pkgconfig
--> Computing dependencies for openssl
--> Fetching openssl
--> Attempting to fetch openssl-1.0.0c.tar.gz from http://aarnet.au.distfiles.
macports.org/pub/macports/mpdistfiles/openssl
--> Verifying checksum(s) for openssl
--> Extracting openssl
--> Applying patches to openssl
--> Configuring openssl
--> Building openssl
--> Staging openssl into destroot
```

Level 2: Getrennte Übersetzung

```
double pow(double b,  
           int e)  
{  
    ...  
}
```

mymath.cpp

g++ -c mymath.cpp

```
001110101100101010  
000101110101000111  
000101110101000111  
111100001101010001  
111111101000111010  
010101101011010001  
100101111100101010
```

mymath.o

Level 2: Getrennte Übersetzung

```
// PRE: e >= 0 || b != 0.0  
// POST: return value is be  
double pow(double b, int e);
```

mymath.h

Level 2: Getrennte Übersetzung

```
#include <iostream>
#include "mymath.h"
int main()
{
    std::cout << pow(2,-2) << "\n";
    return 0;
}
```

callpow3.cpp



```
001110101100101010
000101110101000111
000101100001111111
Funktion main
111100001101010001
010101101011010001
1011111101000111010
rufe pow auf! 1010
111111101000111010
```

callpow3.o

Der Linker vereint...

```
001110101100101010
000101110101000111
000101 Funktion pow
111100001101010001
111111101000111010
010101101011010001
100101111100101010
```

mymath.o

+

```
001110101100101010
000101110101000111
000101 Funktion main
111100001101010001
010101101011010001
1001 rufe pow auf! 1010
111111101000111010
```

callpow3.o

... was zusammengehört

```
001110101100101010
000101110101000111
000101 Funktion pow
111100001101010001
111111101000111010
010101101011010001
100101111100101010
```

mymath.o

+

```
001110101100101010
000101110101000111
000101 Funktion main
111100001101010001
010101101011010001
100101 rufe pow auf!
111111101000111010
```

callpow3.o

=

```
001110101100101010
000101110101000111
000101 Funktion pow
111100001101010001
111111101000111010
010101101011010001
100101111100101010
001110101100101010
000101110101000111
000101 Funktion main
111100001101010001
010101101011010001
100101 rufe addr auf!
111111101000111010
```

Ausführbare Datei callpow3

Verfügbarkeit von Quellcode?

Beobachtung

mymath.cpp (Quellcode) wird nach dem Erzeugen von **mymath.o** (Object Code) nicht mehr gebraucht.

Verfügbarkeit von Quellcode?

Beobachtung

mymath.cpp (Quellcode) wird nach dem Erzeugen von **mymath.o** (Object Code) nicht mehr gebraucht.

Viele Anbieter von Funktionsbibliotheken liefern dem Benutzer keinen Quellcode.

Verfügbarkeit von Quellcode?

Beobachtung

`mymath.cpp` (Quellcode) wird nach dem Erzeugen von `mymath.o` (Object Code) nicht mehr gebraucht.

Viele Anbieter von Funktionsbibliotheken liefern dem Benutzer keinen Quellcode.
Header-Dateien sind dann die *einzigsten* lesbaren Informationen.

Open-Source-Software

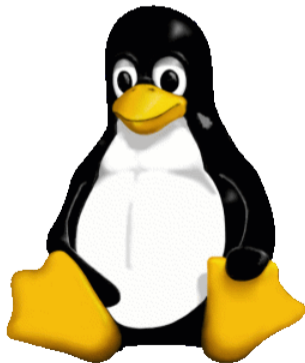
- Alle Quellcodes sind verfügbar.

Open-Source-Software

- Alle Quellcodes sind verfügbar.
- Nur das erlaubt die Weiterentwicklung durch Benutzer und engagierte „Hacker“.

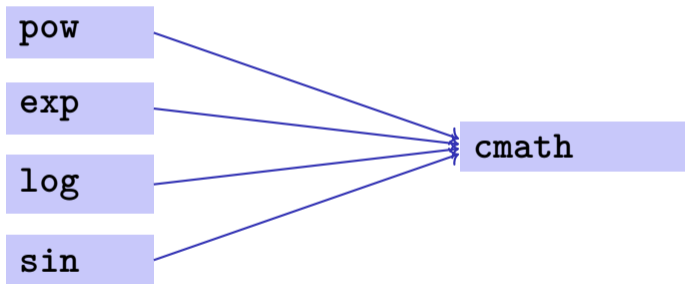
Open-Source-Software

- Alle Quellcodes sind verfügbar.



Bibliotheken

- Logische Gruppierung ähnlicher Funktionen



Namensräume...

```
// cmath
namespace std {

    double pow(double b, int e);

    ....
    double exp(double x);
    ...
}
```

...vermeiden Namenskonflikte

```
#include <cmath>
#include "mymath.h"

int main()
{
    double x = std::pow(2.0, -2); // <cmath>
    double y = pow(2.0, -2); // mymath.h
}
```

Funktionen aus der Standardbibliothek

- vermeiden die Neuerfindung des Rades (wie bei `std::pow`);
- führen auf einfache Weise zu interessanten und effizienten Programmen;

Funktionen aus der Standardbibliothek

- vermeiden die Neuerfindung des Rades (wie bei `std::pow`);
- führen auf einfache Weise zu interessanten und effizienten Programmen;
- garantieren einen Qualitäts-Standard, der mit selbstgeschriebenen Funktionen kaum erreicht werden kann.

Beispiel: Primzahltest mit sqrt

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \dots, n - 1\}$ ein Teiler von n ist.

```
unsigned int d;  
for (d=2; n % d != 0; ++d);
```

Primzahltest mit sqrt

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \dots, \lfloor \sqrt{n} \rfloor\}$ ein Teiler von n ist.

```
unsigned int bound = std::sqrt(n);  
unsigned int d;  
for (d = 2; d <= bound && n % d != 0; ++d);
```

Primzahltest mit sqrt

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \dots, \lfloor \sqrt{n} \rfloor\}$ ein Teiler von n ist.

```
unsigned int bound = std::sqrt(n);  
unsigned int d;  
for (d = 2; d <= bound && n % d != 0; ++d);
```

- Das funktioniert, weil `std::sqrt` auf die nächste darstellbare `double`-Zahl rundet (IEEE Standard 754).

Funktionen sollten mehr können! Swap ?

```
void swap(int x, int y) {  
    int t = x;  
    x = y;  
    y = t;  
}  
  
int main(){  
    int a = 2;  
    int b = 1;  
    swap(a, b);  
    assert(a==1 && b==2);  
}
```

Funktionen sollten mehr können! Swap ?

```
void swap(int x, int y) {  
    int t = x;  
    x = y;  
    y = t;  
}  
  
int main(){  
    int a = 2;  
    int b = 1;  
    swap(a, b);  
    assert(a==1 && b==2); // fail! 😞  
}
```

Funktionen sollten mehr können! Swap ?

```
// POST: values of x and y are exchanged
void swap(int& x, int& y) {
    int t = x;
    x = y;
    y = t;
}
int main(){
    int a = 2;
    int b = 1;
    swap(a, b);
    assert(a==1 && b==2);
}
```

Funktionen sollten mehr können! Swap ?

```
// POST: values of x and y are exchanged
void swap(int& x, int& y) {
    int t = x;
    x = y;
    y = t;
}
int main(){
    int a = 2;
    int b = 1;
    swap(a, b);
    assert(a==1 && b==2); // ok! 😊
}
```


Sneak Preview: Referenztypen

- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern!

Sneak Preview: Referenztypen

- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern!
- Kein neues Konzept auf der Funktionenseite, sondern eine neue Klasse von Typen

Sneak Preview: Referenztypen

- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern!
- Kein neues Konzept auf der Funktionenseite, sondern eine neue Klasse von Typen



Referenztypen (z.B. `int&`)

11. Referenztypen

Referenztypen: Definition und Initialisierung, Pass by Value, Pass by Reference, temporäre Objekte, Const-Referenzen

Swap!

```
// POST: values of x and y have been exchanged
```

```
void swap(int& x, int& y) {  
    int t = x;  
    x = y;  
    y = t;  
}
```

```
int main() {  
    int a = 2;  
    int b = 1;  
    swap(a, b);  
    assert(a == 1 && b == 2); // ok! 😊  
}
```

Referenztypen

- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern

Referenztypen

- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern
- Kein neues Konzept auf der Funktionenseite, sondern eine neue Klasse von Typen: *Referenztypen*

Referenztypen: Definition

T&

Gelesen als „*T*-Referenz“



Zugrundeliegender Typ

Referenztypen: Definition

$T\&$

Gelesen als „ T -Referenz“



Zugrundeliegender Typ

- $T\&$ hat den gleichen Wertebereich und gleiche Funktionalität wie T ...

Referenztypen: Definition

$T\&$

Gelesen als „ T -Referenz“



Zugrundeliegender Typ

- $T\&$ hat den gleichen Wertebereich und gleiche Funktionalität wie T ...
- ...aber Initialisierung und Zuweisung funktionieren anders

Anakin Skywalker alias Darth Vader



Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;  
int& darth_vader = anakin_skywalker; // Alias  
darth_vader = 22;  
  
std::cout << anakin_skywalker;
```

Anakin Skywalker alias Darth Vader

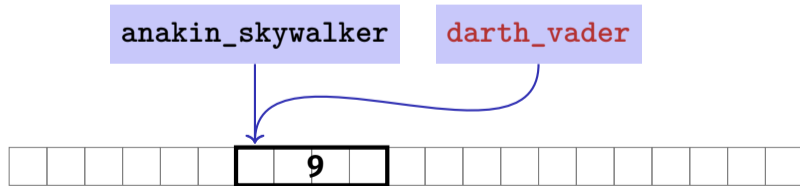
```
int anakin_skywalker = 9;  
int& darth_vader = anakin_skywalker; // Alias  
darth_vader = 22;  
  
std::cout << anakin_skywalker;
```



Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;  
int& darth_vader = anakin_skywalker; // Alias  
darth_vader = 22;
```

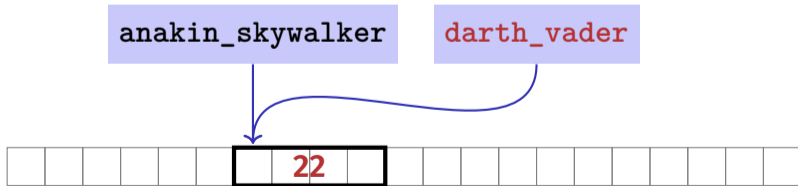
```
std::cout << anakin_skywalker;
```



Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;  
int& darth_vader = anakin_skywalker; // Alias  
darth_vader = 22;
```

```
std::cout << anakin_skywalker;
```



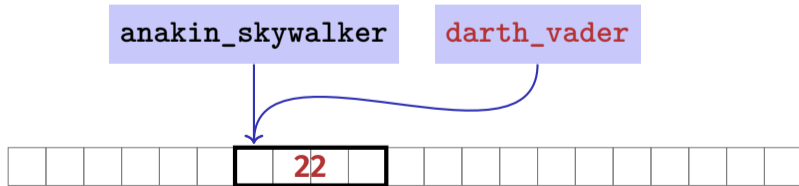
Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;  
int& darth_vader = anakin_skywalker; // Alias
```

```
darth_vader = 22;
```

Zuweisung an den L-Wert hinter dem Alias

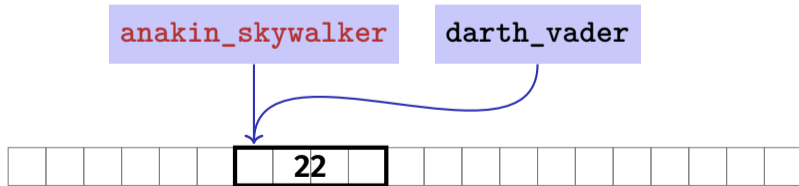
```
std::cout << anakin_skywalker;
```



Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;  
int& darth_vader = anakin_skywalker; // Alias  
darth_vader = 22;
```

```
std::cout << anakin_skywalker; // 22
```



Referenztypen: Initialisierung & Zuweisung

```
int& darth_vader = anakin_skywalker;
```

- Eine Variable mit Referenztyp (eine *Referenz*) muss mit einem L-Wert initialisiert werden

Referenztypen: Initialisierung & Zuweisung

```
int& darth_vader = anakin_skywalker;
```

- Eine Variable mit **Referenztyp** (eine *Referenz*) muss mit einem **L-Wert** initialisiert werden
- Die Variable wird dabei ein *Alias* des **L-Werts** (ein anderer Name für das referenzierte Objekt)

Referenztypen: Initialisierung & Zuweisung

```
int& darth_vader = anakin_skywalker;  
darth_vader = 22; // Effekt: anakin_skywalker = 22
```

- Eine Variable mit **Referenztyp** (eine *Referenz*) muss mit einem **L-Wert** initialisiert werden
- Die Variable wird dabei ein *Alias* des **L-Werts** (ein anderer Name für das referenzierte Objekt)
- Zuweisung an die Referenz erfolgt an das Objekt *hinter* dem Alias

Referenztypen: Realisierung

Intern wird ein Wert vom Typ $T\&$ durch die Adresse eines Objekts vom Typ T repräsentiert.

```
int& j; // Fehler: j muss Alias von irgendetwas sein
```

Referenztypen: Realisierung

Intern wird ein Wert vom Typ $T\&$ durch die Adresse eines Objekts vom Typ T repräsentiert.

```
int& j; // Fehler: j muss Alias von irgendetwas sein
```

```
int& k = 5; // Fehler: Literal 5 hat keine Adresse
```

Pass by Reference

```
void increment (int& i) {  
    ++i;  
}  
  
...  
int j = 5;  
increment (j);  
std::cout << j;
```

Pass by Reference

```
void increment (int& i) {  
    ++i;  
}  
...  
int j = 5;  
increment (j);  
std::cout << j;
```

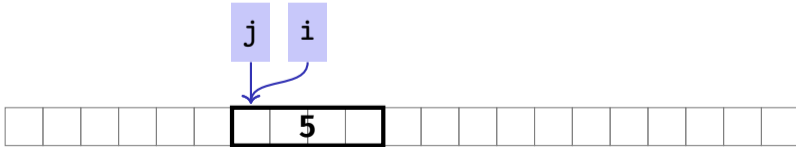


Pass by Reference

```
void increment (int& i) {  
    ++i;  
}
```

Initialisierung der formalen Argumente: `i` wird Alias des Aufrufarguments `j`

```
...  
int j = 5;  
increment (j);  
std::cout << j;
```

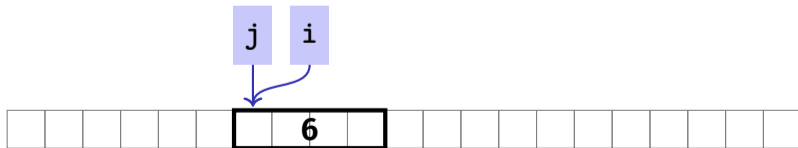


Pass by Reference

```
void increment (int& i) {  
    ++i;  
}
```

...

```
int j = 5;  
increment (j);  
std::cout << j;
```



Pass by Reference

```
void increment (int& i) {  
    ++i;  
}  
...  
int j = 5;  
increment (j);  
std::cout << j; // 6
```



Pass by Reference

Formales Argument *hat* Referenztyp:

⇒ *Pass by Reference*

Formales Argument wird (intern) mit der *Adresse* des Aufrufarguments (L-Wert) initialisiert und wird damit zu einem *Alias*.

Pass by Value

Formales Argument *hat keinen* Referenztyp:

⇒ *Pass by Value*

Formales Argument wird mit dem *Wert* des Aufrufarguments (R-Wert) initialisiert und wird damit zu einer *Kopie*.

Referenzen im Kontext von intervals_intersect

```
// PRE: [a1, b1], [a2, b2] are (generalized) intervals,  
// POST: returns true if [a1, b1], [a2, b2] intersect, in which case  
//       [l, h] contains the intersection of [a1, b1], [a2, b2]
```

```
bool intervals_intersect(int& l, int& h,  
                        int a1, int b1, int a2, int b2) {  
    sort(a1, b1);  
    sort(a2, b2);  
    l = std::max(a1, a2); // Zuweisungen  
    h = std::min(b1, b2); // via Referenzen  
    return l <= h;  
}
```



```
...
```

```
int lo = 0; int hi = 0;
```

```
if (intervals_intersect(lo, hi, 0, 2, 1, 3)) // Initialisierung  
    std::cout << "[" << lo << "," << hi << "]" << "\n"; // [1,2]
```

Referenzen im Kontext von intervals intersect

```
// POST: a <= b
void sort(int& a, int& b) {
    if (a > b)
        std::swap(a, b); // Initialisierung ("Durchreichen" von a, b)
}
```

```
bool intervals_intersect(int& l, int& h,
                        int a1, int b1, int a2, int b2) {
    sort(a1, b1); // Initialisierung
    sort(a2, b2); // Initialisierung
    l = std::max(a1, a2);
    h = std::min(b1, b2);
    return l <= h;
}
```

Return by Reference

- Auch der Rückgabebetyp einer Funktion kann ein Referenztyp sein: *Return by Reference*

Return by Reference

- Auch der Rückgabotyp einer Funktion kann ein Referenztyp sein: *Return by Reference*

```
int& inc(int& i) {  
    return ++i;  
}
```

Return by Reference

- Auch der Rückgabotyp einer Funktion kann ein Referenztyp sein: *Return by Reference*

```
int& inc(int& i) {  
    return ++i;  
}
```

- Aufruf `inc(x)`, für eine `int`-Variable `x`, hat exakt die Semantik des Prä-Inkrementes `++x`

Return by Reference

- Auch der Rückgabebetyp einer Funktion kann ein Referenztyp sein: *Return by Reference*

```
int& inc(int& i) {  
    return ++i;  
}
```

- Aufruf `inc(x)`, für eine `int`-Variable `x`, hat exakt die Semantik des Prä-Inkrementes `++x`
- Funktionsaufruf *selbst* ist nun ein L-Wert

Return by Reference

- Auch der Rückgabebetyp einer Funktion kann ein Referenztyp sein: *Return by Reference*

```
int& inc(int& i) {  
    return ++i;  
}
```

- Aufruf **inc(x)**, für eine **int**-Variable **x**, hat exakt die Semantik des Prä-Inkrement **++x**
- Funktionsaufruf *selbst* ist nun ein L-Wert
- Daher möglich: **inc(inc(x))** oder **++(inc(x))**

Temporäre Objekte

Was ist hier falsch?

```
int& foo(int i) {  
    return i;  
}
```

Temporäre Objekte

Was ist hier falsch?

```
int& foo(int i) {  
    return i;  
}
```

```
int k = 3;  
int& j = foo(k); // j ist Alias einer "Leiche"  
std::cout << j; // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

```
int& foo(int i) {  
    return i;  
}
```



```
// main()
```

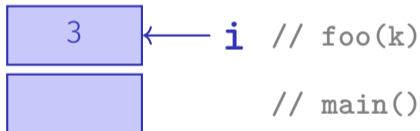
```
int k = 3;  
int& j = foo(k); // j ist Alias einer "Leiche"  
std::cout << j; // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

Wert des Aufrufarguments kommt
auf den *Aufrufstapel*

```
int& foo(int i) {  
    return i;  
}
```



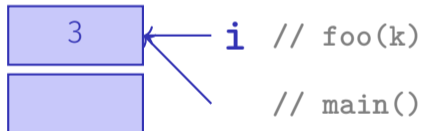
```
int k = 3;  
int& j = foo(k); // j ist Alias einer "Leiche"  
std::cout << j; // undefined behavior
```


Temporäre Objekte

Was ist hier falsch?

i wird als Referenz zurückgegeben

```
int& foo(int i) {  
    return i;  
}
```



```
int k = 3;  
int& j = foo(k); // j ist Alias einer "Leiche"  
std::cout << j; // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

```
int& foo(int i) {  
    return i;  
}
```

```
int k = 3;  
int& j = foo(k); // j ist Alias einer "Leiche"  
std::cout << j; // undefined behavior
```

...und verschwindet vom Aufrufstapel



// main()

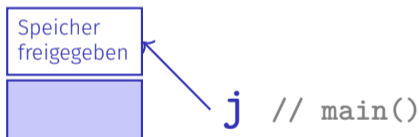
Temporäre Objekte

Was ist hier falsch?

```
int& foo(int i) {  
    return i;  
}
```

```
int k = 3;  
int& j = foo(k); // j ist Alias einer "Leiche"  
std::cout << j; // undefined behavior
```

j wird Alias des freigegebenen Speichers



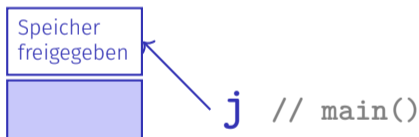
Temporäre Objekte

Was ist hier falsch?

```
int& foo(int i) {  
    return i;  
}
```

```
int k = 3;  
int& j = foo(k); // j ist Alias einer "Leiche"  
std::cout << j; // undefined behavior
```

Zugriff auf `j` ist undefiniertes Verhalten!



Die Referenz-Richtlinie

Referenz-Richtlinie

Wenn man eine Referenz erzeugt, muss das Objekt, auf das sie verweist, mindestens so lange „leben“ wie die Referenz selbst.

Const-Referenzen

- haben Typ `const T &`
- Typ kann verstanden werden als „`(const T) &`“
- können auch mit R-Werten initialisiert werden (Compiler erzeugt temporäres Objekt ausreichender Lebensdauer)

Const-Referenzen

- haben Typ `const T &`
- Typ kann verstanden werden als „`(const T) &`“
- können auch mit R-Werten initialisiert werden (Compiler erzeugt temporäres Objekt ausreichender Lebensdauer)

```
const T& r = lvalue;
```

`r` wird mit der Adresse von `lvalue` initialisiert (effizient)

Const-Referenzen

- haben Typ `const T &`
- Typ kann verstanden werden als „`(const T) &`“
- können auch mit R-Werten initialisiert werden (Compiler erzeugt temporäres Objekt ausreichender Lebensdauer)

```
const T& r = rvalue;
```

`r` wird mit der Adresse eines temporären Objektes vom Wert des `rvalue` initialisiert (pragmatisch)

Was genau ist konstant?

Betrachte L-Wert vom Typ `const T`. **Fall: 1** *T ist kein Referenztyp.*

⇒ Dann ist der *L-Wert eine Konstante*

```
const int n = 5;  
int& a = n;  
a = 6;
```

Was genau ist konstant?

Betrachte L-Wert vom Typ `const T`. **Fall: 1** *T ist kein Referenztyp.*

⇒ Dann ist der *L-Wert eine Konstante*

```
const int n = 5;  
int& a = n; // Compilerfehler: const-qualification discarded  
a = 6;
```

Unser *Schummelversuch* wird vom Compiler erkannt

Was genau ist konstant?

Betrachte L-Wert vom Typ **const** *T*. **Fall 2:** *T* ist Referenztyp.

⇒ Dann ist der *L-Wert ein Lese-Alias*, durch den der L-Wert *dahinter* nicht verändert werden darf.

Was genau ist konstant?

Betrachte L-Wert vom Typ `const T`. **Fall 2:** *T* ist Referenztyp.

⇒ Dann ist der *L-Wert ein Lese-Alias*, durch den der L-Wert *dahinter* nicht verändert werden darf.

```
int n = 5;

const int& r = n; // r ist Lese-Alias von n
r = 6;           // Compilerfehler: read-only reference
```

Was genau ist konstant?

Betrachte L-Wert vom Typ `const T`. **Fall 2:** *T* ist Referenztyp.

⇒ Dann ist der L-Wert ein *Lese-Alias*, durch den der L-Wert *dahinter* nicht verändert werden darf.

```
int n = 5;

const int& r = n; // r ist Lese-Alias von n
r = 6;           // Compilerfehler: read-only reference

int& rw = n;     // rw ist Lese-Schreib-Alias
rw = 6;         // OK
```

Wann `const T&` verwenden?

```
void f_1(T& arg);
```

```
void f_2(const T& arg);
```

- Argumenttypen sind Referenzen; Aufrufargumente werden daher nicht kopiert, was effizient ist
- Aber nur `f_2` „verspricht“ Argument nicht zu verändern

Wann `const T&` verwenden?

```
void f_1(T& arg);
```

```
void f_2(const T& arg);
```

- Argumenttypen sind Referenzen; Aufrufargumente werden daher nicht kopiert, was effizient ist
- Aber nur `f_2` „verspricht“ Argument nicht zu verändern

Regel

Funktionsargumenttypen, falls möglich, als `const T&` (*pass by read-only reference*) deklarieren: effizient *und* sicher.

Wann `const T&` verwenden?

```
void f_1(T& arg);
```

```
void f_2(const T& arg);
```

- Argumenttypen sind Referenzen; Aufrufargumente werden daher nicht kopiert, was effizient ist
- Aber nur `f_2` „verspricht“ Argument nicht zu verändern

Regel

Funktionsargumenttypen, falls möglich, als `const T&` (*pass by read-only reference*) deklarieren: effizient *und* sicher.

Lohnt sich i.d.R. nicht für fundamentale Typen (`int`, `double`, ...). Wir lernen später in der Vorlesung Typen kennen, die mehr Speicherplatz benötigen.

12. Vektoren I

Vektoren, Sieb des Eratosthenes, Speicherlayout, Iteration

Vektoren: Motivation

- Wir können jetzt über Zahlen iterieren

```
for (int i=0; i<n ; ++i) {...}
```

Vektoren: Motivation

- Wir können jetzt über Zahlen iterieren

```
for (int i=0; i<n ; ++i) {...}
```

- .. aber noch nicht über Daten!

Vektoren: Motivation

- Wir können jetzt über Zahlen iterieren

```
for (int i=0; i<n ; ++i) {...}
```

- .. aber noch nicht über Daten!
- Vektoren speichern *gleichartige* Daten.

Vektoren: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$

Vektoren: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Vektoren: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Streiche alle echten Vielfachen von 2 ...

Vektoren: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	--------------	---	--------------	---	--------------	---	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----

Streiche alle echten Vielfachen von 2 ...

Vektoren: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	--------------	---	--------------	---	--------------	---	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----

... und gehe zur nächsten Zahl

Vektoren: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	--------------	---	--------------	---	--------------	---	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----

Streiche alle echten Vielfachen von 3 ...

Vektoren: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	--------------	---	--------------	---	--------------	--------------	---------------	----	---------------	----	---------------	---------------	---------------	----	---------------	----	---------------	---------------	---------------	----

Streiche alle echten Vielfachen von 3 ...

Vektoren: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	--------------	---	--------------	---	--------------	--------------	---------------	----	---------------	----	---------------	---------------	---------------	----	---------------	----	---------------	---------------	---------------	----

... und gehe zur nächsten Zahl

Vektoren: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	--------------	---	--------------	---	--------------	--------------	---------------	----	---------------	----	---------------	---------------	---------------	----	---------------	----	---------------	---------------	---------------	----

Am Ende des Streichungsprozesses bleiben nur die Primzahlen übrig.

Vektoren: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	--------------	---	--------------	---	--------------	--------------	---------------	----	---------------	----	---------------	---------------	---------------	----	---------------	----	---------------	---------------	---------------	----

- Frage: wie streichen wir Zahlen aus?

Vektoren: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	--------------	---	--------------	---	--------------	--------------	---------------	----	---------------	----	---------------	---------------	---------------	----	---------------	----	---------------	---------------	---------------	----

- Frage: wie streichen wir Zahlen aus?
- Antwort: mit einem *Vektor*.

Eratosthenes mit Vektoren: Initialisierung


...

```
#include <vector>
```

...

```
std::vector<bool> crossed_out(n, false);
```

Initialisierung mit **n** Elementen
Initialwert **false**.



↑
Elementtyp, in spitzen Klammern

Eratosthenes mit Vektoren: Berechnung

```
for (unsigned int i = 2; i < crossed_out.size(); ++i)
    if (!crossed_out[i]) { // i is prime
        std::cout << i << " ";

        // cross out all proper multiples of i
        for (unsigned int m = 2*i; m < crossed_out.size(); m += i)
            crossed_out[m] = true;
    }
```

Speicherlayout eines Vektors

Ein Vektor belegt einen *zusammenhängenden* Speicherbereich

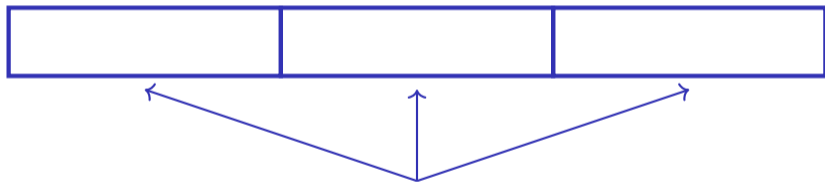
Beispiel: Ein Vektor mit 3 Elementen vom Type **T**



Speicherlayout eines Vektors

Ein Vektor belegt einen *zusammenhängenden* Speicherbereich

Beispiel: Ein Vektor mit 3 Elementen vom Type **T**

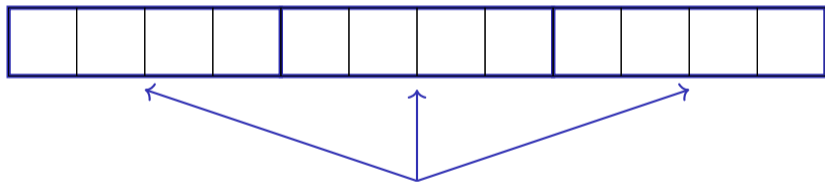


Speichersegmente für jeweils einen Wert vom Typ **T**

Speicherlayout eines Vektors

Ein Vektor belegt einen *zusammenhängenden* Speicherbereich

Beispiel: Ein Vektor mit 3 Elementen vom Type **T**



Speichersegmente für jeweils einen Wert vom Typ **T**
(**T** belegt z.B. 4 Bytes)

Wahlfreier Zugriff (Random Access)

Gegeben

- Vektor **vec** mit **T**-Elementen
- **int**-Ausdruck **exp** mit Wert $i \geq 0$

Wahlfreier Zugriff (Random Access)

Gegeben

- Vektor **vec** mit **T**-Elementen
- **int**-Ausdruck **exp** mit Wert $i \geq 0$

Dann ist der Ausdruck

vec [exp]

- ein *L-Wert* vom Type **T**

Wahlfreier Zugriff (Random Access)

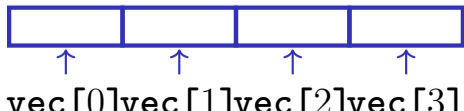
Gegeben

- Vektor **vec** mit **T**-Elementen
- **int**-Ausdruck **exp** mit Wert $i \geq 0$

Dann ist der Ausdruck

vec [exp]

- ein *L*-Wert vom Type **T**
- der sich auf das *i*-te Element von **vec** bezieht (Zählung ab 0!)



Wahlfreier Zugriff (Random Access)

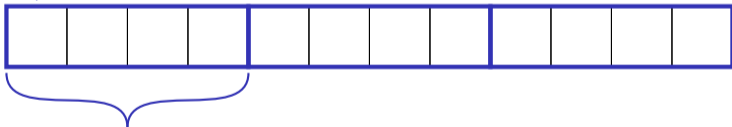
`vec [exp]`

- Der Wert i von `exp` heisst *Index*
- `[]` ist der *Index-Operator* (auch *Subskript-Operator*)

Wahlfreier Zugriff (Random Access)

Wahlfreier Zugriff ist sehr effizient:

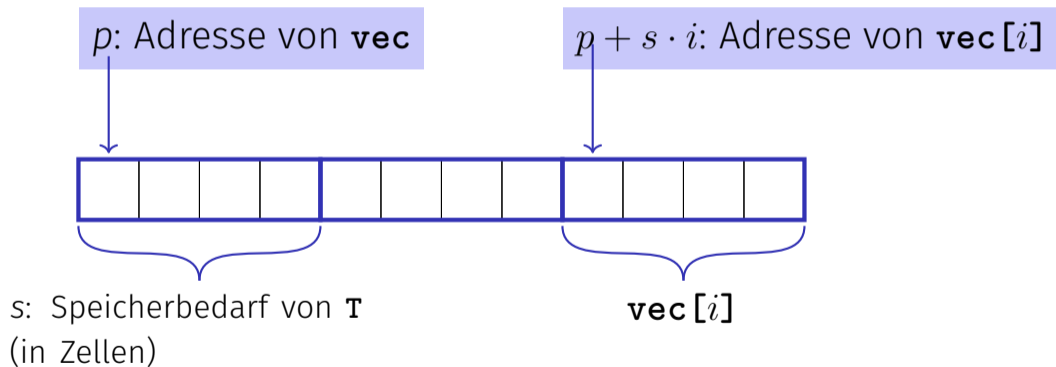
p : Adresse von **vec**, d.h. Adresse der ersten Speicherzelle



s : Speicherbedarf von **T**
(in Zellen)

Wahlfreier Zugriff (Random Access)

Wahlfreier Zugriff ist sehr effizient:



Vektorinitialisierung

■ `std::vector<int> vec(5);`

Die 5 Elemente von `vec` werden mit Nullen initialisiert

Vektorinitialisierung

- `std::vector<int> vec(5);`

Die 5 Elemente von `vec` werden mit Nullen initialisiert

- `std::vector<int> vec(5, 2);`

Die 5 Elemente von `vec` werden mit 2 initialisiert

Vektorinitialisierung

- `std::vector<int> vec(5);`

Die 5 Elemente von `vec` werden mit Nullen initialisiert

- `std::vector<int> vec(5, 2);`

Die 5 Elemente von `vec` werden mit 2 initialisiert

- `std::vector<int> vec{4, 3, 5, 2, 1};`

Der Vektor wird mit einer *Initialisierungsliste* initialisiert

Vektorinitialisierung

- `std::vector<int> vec(5);`

Die 5 Elemente von `vec` werden mit Nullen initialisiert

- `std::vector<int> vec(5, 2);`

Die 5 Elemente von `vec` werden mit 2 initialisiert

- `std::vector<int> vec{4, 3, 5, 2, 1};`

Der Vektor wird mit einer *Initialisierungsliste* initialisiert

- `std::vector<int> vec;`

Ein leerer Vektor wird initialisiert

Achtung

Der Zugriff auf Elemente ausserhalb der gültigen Grenzen eines Vektors führt zu *undefiniertem Verhalten*

```
std::vector vec(10);  
for (unsigned int i = 0; i <= 10; ++i)  
    vec[i] = 30;
```

Achtung

Der Zugriff auf Elemente ausserhalb der gültigen Grenzen eines Vektors führt zu *undefiniertem Verhalten*


```
std::vector vec(10);  
for (unsigned int i = 0; i <= 10; ++i)  
    vec[i] = 30; // Laufzeit-Fehler: Zugriff auf vec[10]
```


Achtung

Prüfung der Indexgrenzen

Bei Verwendung des Indexoperators auf einem Vektor ist es die alleinige **Verantwortung des Programmierers**, die Gültigkeit aller Elementzugriffe zu prüfen.

Konsequenzen illegaler Index-Zugriffe

 
[Alle](#) [Videos](#) [Bilder](#) [News](#) [Shopping](#) [Mehr](#) [Einstellungen](#) [Tools](#)

Ungefähr 127'000 Ergebnisse (0.30 Sekunden)

CWE - CWE-125: Out-of-bounds Read (3.0)

<https://cwe.mitre.org> › [CWE List](#) ▼ [Diese Seite übersetzen](#)

However, this method only verifies that the given array index is less than the maximum length of the array but does not check for the minimum value (CWE-839). This will allow a negative value to be accepted as the input array index, which will result in a **out of bounds** read (CWE-125) and may allow access to sensitive ...

CWE - CWE-787: Out-of-bounds Write (3.0)

<https://cwe.mitre.org> › [CWE List](#) ▼ [Diese Seite übersetzen](#)

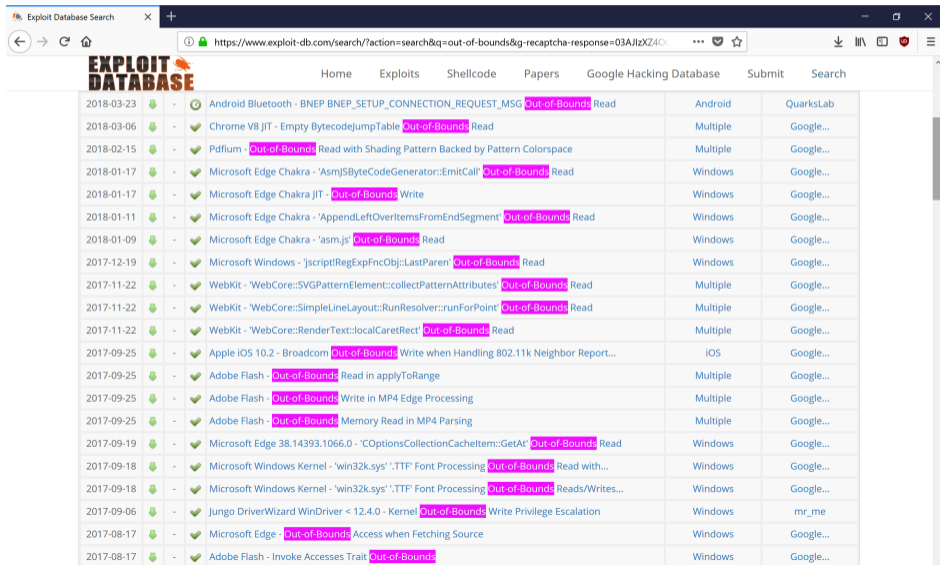
This typically occurs when the pointer or its index is incremented or decremented to a position beyond the bounds of the buffer or when pointer arithmetic results in a position outside of the valid memory location to name a few. This may result in corruption of sensitive information, a crash, or code execution among other ...

c - How dangerous is it to access an array out of bounds? - Stack ...

<https://stackoverflow.com/.../how-dangerous-is-it-to-access-an-arr...> ▼ [Diese Seite übersetzen](#)

As far as the ISO C standard (the official definition of the language) is concerned, accessing an array outside its bounds has "undefined behavior". The literal meaning of this is: behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no ...

Konsequenzen illegaler Index-Zugriffe



The screenshot shows a web browser window displaying the Exploit Database search results for the query 'out-of-bounds'. The page features a navigation bar with links for Home, Exploits, Shellcode, Papers, Google Hacking Database, Submit, and Search. The main content is a table of search results, each with a date, a status icon, a title, a type of access (e.g., Read, Write), a platform, and a source.

Date	Status	Title	Access Type	Platform	Source
2018-03-23	🟢	Android Bluetooth - BNEP BNEP_SETUP_CONNECTION_REQUEST_MSG	Out-of-Bounds Read	Android	QuarksLab
2018-03-06	🟢	Chrome V8 JIT - Empty BytecodeJumpTable	Out-of-Bounds Read	Multiple	Google...
2018-02-15	🟢	Pdfium - Out-of-Bounds Read with Shading Pattern Backed by Pattern Colorspace	Out-of-Bounds Read	Multiple	Google...
2018-01-17	🟢	Microsoft Edge Chakra - 'AsmJSByteCodeGenerator::EmitCall'	Out-of-Bounds Read	Windows	Google...
2018-01-17	🟢	Microsoft Edge Chakra JIT -	Out-of-Bounds Write	Windows	Google...
2018-01-11	🟢	Microsoft Edge Chakra - 'AppendLeftOverItemsFromEndSegment'	Out-of-Bounds Read	Windows	Google...
2018-01-09	🟢	Microsoft Edge Chakra - 'asm.js'	Out-of-Bounds Read	Windows	Google...
2017-12-19	🟢	Microsoft Windows - 'jscript!RegExpFuncObj::LastParen'	Out-of-Bounds Read	Windows	Google...
2017-11-22	🟢	WebKit - 'WebCore::SVGPatternElement::collectPatternAttributes'	Out-of-Bounds Read	Multiple	Google...
2017-11-22	🟢	WebKit - 'WebCore::SimpleLineLayout::RunResolver::runForPoint'	Out-of-Bounds Read	Multiple	Google...
2017-11-22	🟢	WebKit - 'WebCore::RenderText::localCaretRect'	Out-of-Bounds Read	Multiple	Google...
2017-09-25	🟢	Apple iOS 10.2 - Broadcom	Out-of-Bounds Write when Handling 802.11k Neighbor Report...	iOS	Google...
2017-09-25	🟢	Adobe Flash -	Out-of-Bounds Read in applyToRange	Multiple	Google...
2017-09-25	🟢	Adobe Flash -	Out-of-Bounds Write in MP4 Edge Processing	Multiple	Google...
2017-09-25	🟢	Adobe Flash -	Out-of-Bounds Memory Read in MP4 Parsing	Multiple	Google...
2017-09-19	🟢	Microsoft Edge 38.14393.1066.0 - 'COptionsCollectionCacheItem::GetAt'	Out-of-Bounds Read	Windows	Google...
2017-09-18	🟢	Microsoft Windows Kernel - 'win32k.sys' '.TTF' Font Processing	Out-of-Bounds Read with...	Windows	Google...
2017-09-18	🟢	Microsoft Windows Kernel - 'win32k.sys' '.TTF' Font Processing	Out-of-Bounds Reads/Writes...	Windows	Google...
2017-09-06	🟢	Jungo DriverWizard WinDriver < 12.4.0 - Kernel	Out-of-Bounds Write Privilege Escalation	Windows	mr_me
2017-08-17	🟢	Microsoft Edge -	Out-of-Bounds Access when Fetching Source	Windows	Google...
2017-08-17	🟢	Adobe Flash - Invoke Accesses Trait	Out-of-Bounds	Windows	Google...

Vektoren bieten viel Funktionalität

Hier ein paar Beispielfunktionen, weitere folgen später in der Vorlesung.

Vektoren bieten viel Funktionalität

Hier ein paar Beispielfunktionen, weitere folgen später in der Vorlesung.

```
std::vector<int> v(10);  
std::cout << v.at(10);  
    // Zugriff mit Index-Check → Laufzeitfehler  
    // Ideal für Hausaufgaben
```

Vektoren bieten viel Funktionalität

Hier ein paar Beispielfunktionen, weitere folgen später in der Vorlesung.

```
std::vector<int> v(10);  
std::cout << v.at(10);  
    // Zugriff mit Index-Check → Laufzeitfehler  
    // Ideal für Hausaufgaben  
  
v.push_back(-1); // -1 is appended (added at end)  
std::cout << v.size(); // outputs 11  
std::cout << v.at(10); // outputs -1
```

13. Zeichen und Texte I

Zeichen und Texte, ASCII, UTF-8, Caesar-Code

Zeichen und Texte

- Texte haben wir schon gesehen:

```
std::cout << "Prime numbers in {2,...,999}:\n";
```

String-Literal

Zeichen und Texte

- Texte haben wir schon gesehen:

```
std::cout << "Prime numbers in {2,...,999}:\n";
```

String-Literal

- Können wir auch „richtig“ mit Texten arbeiten?

Zeichen und Texte

- Texte haben wir schon gesehen:

```
std::cout << "Prime numbers in {2,...,999}:\n";
```

String-Literal

- Können wir auch „richtig“ mit Texten arbeiten? Ja!

Zeichen: Wert des fundamentalen Typs **char**

Text: **std::string** \approx Vektor von **char** Elementen

Der Typ `char` („character“)

Repräsentiert druckbare Zeichen (z.B. `'a'`) und *Steuerzeichen* (z.B. `'\n'`)

Der Typ `char` („character“)

Repräsentiert druckbare Zeichen (z.B. `'a'`) und *Steuerzeichen* (z.B. `'\n'`)

```
char c = 'a';
```

Deklariert und initialisiert Variable `c` vom Typ `char` mit Wert `'a'`

Der Typ `char` („character“)

Repräsentiert druckbare Zeichen (z.B. `'a'`) und *Steuerzeichen* (z.B. `'\n'`)

```
char c = 'a';
```

Deklariert und initialisiert Variable `c` vom Typ `char` mit Wert `'a'`

Literal vom Typ `char`

Der Typ `char` („character“)

Ist formal ein ganzzahliger Typ

- Werte konvertierbar nach `int` / `unsigned int`

Der Typ `char` („character“)

Ist formal ein ganzzahliger Typ

- Werte konvertierbar nach `int` / `unsigned int`
- Werte belegen meistens 8 Bit

Wertebereich:

$\{-128, \dots, 127\}$ oder $\{0, \dots, 255\}$

Der ASCII-Code

- Definiert konkrete Konversionsregeln **char** \longrightarrow **(unsigned) int**

Zeichen \longrightarrow $\{0, \dots, 127\}$

'A', 'B', ... , 'Z' \longrightarrow 65, 66, ..., 90

'a', 'b', ... , 'z' \longrightarrow 97, 98, ..., 122

'0', '1', ... , '9' \longrightarrow 48, 49, ..., 57

Der ASCII-Code

- Definiert konkrete Konversionsregeln **char** \longrightarrow **(unsigned) int**

Zeichen \longrightarrow $\{0, \dots, 127\}$

'A', 'B', ... , 'Z' \longrightarrow 65, 66, ..., 90

'a', 'b', ... , 'z' \longrightarrow 97, 98, ..., 122

'0', '1', ... , '9' \longrightarrow 48, 49, ..., 57

- Wird von allen gängigen Computersystemen unterstützt

Der ASCII-Code

- Definiert konkrete Konversionsregeln **char** \longrightarrow **(unsigned) int**

Zeichen \longrightarrow $\{0, \dots, 127\}$

'A', 'B', ... , 'Z' \longrightarrow 65, 66, ..., 90

'a', 'b', ... , 'z' \longrightarrow 97, 98, ..., 122

'0', '1', ... , '9' \longrightarrow 48, 49, ..., 57

- Wird von allen gängigen Computersystemen unterstützt
- Erlaubt Arithmetik über Zeichen

```
for (char c = 'a'; c <= 'z'; ++c)
    std::cout << c; // abcdefghijklmnopqrstuvwxyz
```

Erweiterung von ASCII: Unicode, UTF-8

- Internationalisierung von Software \Rightarrow grosse Zeichensätze nötig. Heute daher üblich:
 - Zeichensatz *Unicode*: 150 Schriftsysteme, ca. 137'000 Zeichen
 - Codierungsstandard *UTF-8*: Abbildung Zeichen \leftrightarrow Zahlen

Erweiterung von ASCII: Unicode, UTF-8

- Internationalisierung von Software \Rightarrow grosse Zeichensätze nötig. Heute daher üblich:
 - Zeichensatz *Unicode*: 150 Schriftsysteme, ca. 137'000 Zeichen
 - Codierungsstandard *UTF-8*: Abbildung Zeichen \leftrightarrow Zahlen
- UTF-8 ist eine *Multibyte-Codierung*: Häufig genutzte Zeichen (z.B. lat. Alphabet) belegen nur ein Byte, andere Zeichen bis zu vier

Erweiterung von ASCII: Unicode, UTF-8

- Internationalisierung von Software \Rightarrow grosse Zeichensätze nötig. Heute daher üblich:
 - Zeichensatz *Unicode*: 150 Schriftsysteme, ca. 137'000 Zeichen
 - Codierungsstandard *UTF-8*: Abbildung Zeichen \leftrightarrow Zahlen
- UTF-8 ist eine *Multibyte-Codierung*: Häufig genutzte Zeichen (z.B. lat. Alphabet) belegen nur ein Byte, andere Zeichen bis zu vier
- Länge einer Zeichen-Bytefolge wird dabei durch Bitmuster codiert

Nutzbare Bits	Bitmuster
7	0xxxxxxx
11	110xxxxx 10xxxxxx

Einige Unicode-Zeichen in UTF-8

Symbol	Codierung (jeweils 16 Bit)
ئى	11101111 10101111 10111001

Einige Unicode-Zeichen in UTF-8

Symbol	Codierung (jeweils 16 Bit)
س	11101111 10101111 10111001
☠	11100010 10011000 10100000
☎	11100010 10011000 10000011
☯	11100010 10011000 10011001

Einige Unicode-Zeichen in UTF-8

Symbol	Codierung (jeweils 16 Bit)
س	11101111 10101111 10111001
☠	11100010 10011000 10100000
☎	11100010 10011000 10000011
☪	11100010 10011000 10011001
A	01000001

Caesar-Code

Ersetze jedes druckbare Zeichen in einem Text durch seinen Vor-Vor-Vorgänger.

' ' (32) → '|' (124)

'!' (33) → '}' (125)

...

'D' (68) → 'A' (65)

'E' (69) → 'B' (66)

...

~ (126) → '{' (123)



```
// PRE: divisor > 0
// POST: return the remainder of dividend / divisor
//        with 0 <= result < divisor
int mod(int dividend, int divisor);

// POST: if c is one of the 95 printable ASCII characters, c is
//        cyclically shifted s printable characters to the right
char shift(char c, int s) {
    if (c >= 32 && c <= 126) { // c is printable
        c = 32 + mod(c - 32 + s, 95);
    }

    return c;
}
```

```
// PRE:  divisor > 0
// POST: return the remainder of dividend / divisor
//       with 0 <= result < divisor
int mod(int dividend, int divisor);

// POST: if c is one of the 95 printable ASCII characters, c is
//       cyclically shifted s printable characters to the right
char shift(char c, int s) {
    if (c >= 32 && c <= 126) { // c is printable
        c = 32 + mod(c - 32 + s, 95);
    }

    return c;
}
```

"- 32" transforms interval [32, 126] to [0, 94]
"mod(x, 95)" computes $x \bmod 95$ in [0, 94]

```
// POST: Each character read from std::cin was shifted cyclically
//       by s characters and afterwards written to std::cout
void caesar(int s) {
    std::cin >> std::noskipws; // #include <ios>

    char next;
    while (std::cin >> next) {
        std::cout << shift(next, s);
    }
}
```

Leerzeichen und Zeilen-
umbrüche sollen *nicht* ignori-
ert werden

```
// POST: Each character read from std::cin was shifted cyclically
//       by s characters and afterwards written to std::cout
void caesar(int s) {
    std::cin >> std::noskipws; // #include <ios>

    char next;
    while (std::cin >> next) {
        std::cout << shift(next, s);
    }
}
```

Konversion nach **bool**: liefert *false* genau dann, wenn die Eingabe leer ist

```
// POST: Each character read from std::cin was shifted cyclically
//       by s characters and afterwards written to std::cout
void caesar(int s) {
    std::cin >> std::noskipws; // #include <ios>

    char next;
    while (std::cin >> next) {
        std::cout << shift(next, s);
    }
}
```

Verschiebung druckbarer Zeichen
um **s**

```
int main() {  
    int s;  
    std::cin >> s;  
  
    // Shift input by s  
    caesar(s);  
  
    return 0;  
}
```

Verschlüsseln: Verschiebung um n (hier: 3)

```
3.  
Hello World, my password is 1234.  
Khoor#Zruog/#p|#sdvvzrug#lv#45671
```

Entschlüsseln: Verschiebung um $-n$ (hier: -3)

```
-3.  
Khoor#Zruog/#p|#sdvvzrug#lv#45671  
Hello World, my password is 1234.
```

Caesar-Code: Generalisierung

```
void caesar(int s) {  
    std::cin >> std::noskipws;  
  
    char next;  
    while (std::cin >> next) {  
        std::cout << shift(next, s);  
    }  
}
```

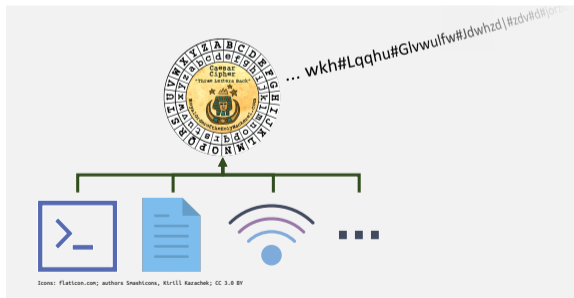
- Momentan nur von `std::cin` nach `std::cout`

Caesar-Code: Generalisierung

```
void caesar(int s) {  
    std::cin >> std::noskipws;  
  
    char next;  
    while (std::cin >> next) {  
        std::cout << shift(next, s);  
    }  
}
```

- Momentan nur von `std::cin` nach `std::cout`

- Besser: von beliebiger Zeichenquelle (Konsole, Datei, ...) zu beliebiger Zeichensenke (Konsole, ...)



14. Zeichen und Texte II

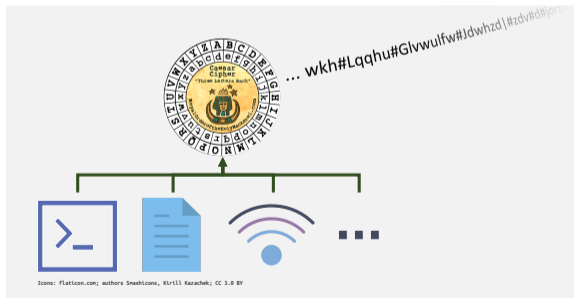
Caesar-Code mit Streams, Texte als Strings,
String-Operationen

Caesar-Code: Generalisierung

```
void caesar(int s) {  
    std::cin >> std::noskipws;  
  
    char next;  
    while (std::cin >> next) {  
        std::cout << shift(next, s);  
    }  
}
```

- Momentan nur von `std::cin` nach `std::cout`

- Besser: von beliebiger Zeichenquelle (Konsole, Datei, ...) zu beliebiger Zeichensenke (Konsole, ...)



Einschub: Abstrakte vs. konkrete Typen

DestroyBox



(abstrakt,
generisch)

Einschub: Abstrakte vs. konkrete Typen

DestroyBox



(abstrakt,
generisch)



(ist eine)



(konkret,
spezifisch)

FireBox

Einschub: Abstrakte vs. konkrete Typen

DestroyBox



(abstrakt,
generisch)

(ist eine)



(konkret,
spezifisch)

ShredBox

FireBox

Einschub: Abstrakte vs. konkrete Typen

DestroyBox



(abstrakt,
generisch)

(ist eine)



(konkret,
spezifisch)

ShredBox

FireBox

Einschub: Abstrakte vs. konkrete Typen

DestroyBox



(abstrakt,
generisch)

(ist eine)



ShredBox

FireBox

(konkret,
spezifisch)

```
void move_house(DestroyBox& db) {  
    // any destroy box will do  
    db.dispose(old_ikea_couch);  
    db.dispose(cheap_wine);  
    ...  
}
```


Einschub: Abstrakte vs. konkrete Typen

DestroyBox



(abstrakt,)
(generisch)

(ist eine)



ShredBox

FireBox

(konkret,)
(spezifisch)

```
void move_house(DestroyBox& db) {  
    // any destroy box will do  
    db.dispose(old_ikea_couch);  
    db.dispose(cheap_wine);  
    ...  
}
```

```
FireBox fb(5000°C);  
move_house(fb);
```

```
ShredBox sb;  
move_house(sb);
```

Abstrakte und konkrete Zeichenströme

DestroyBox



(abstrakt,
generisch)

(ist eine)



ShredBox



FireBox

(konkret,
spezifisch)

`std::ostream`



Abstrakte und konkrete Zeichenströme

DestroyBox



(abstrakt,
generisch)

(ist eine)



ShredBox



FireBox

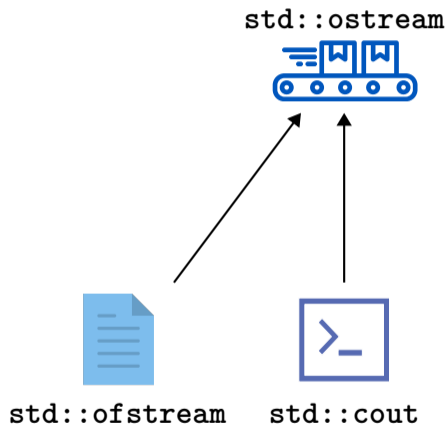
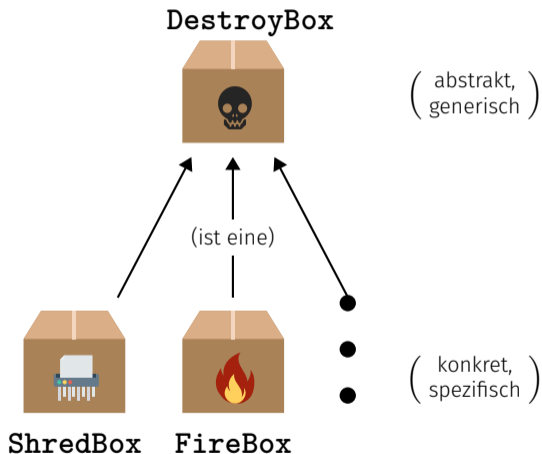
(konkret,
spezifisch)

std::ostream

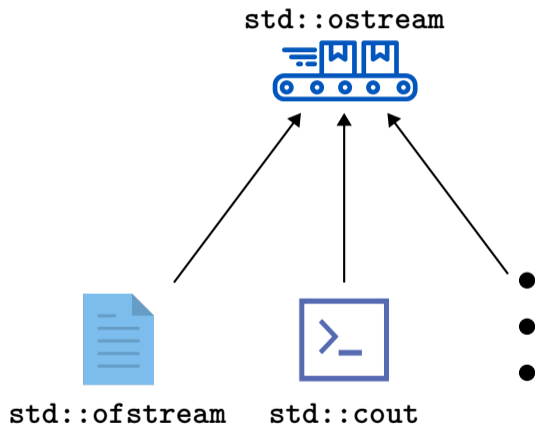
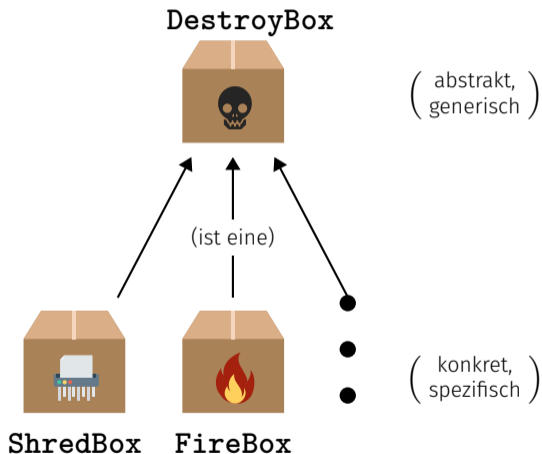


std::cout

Abstrakte und konkrete Zeichenströme



Abstrakte und konkrete Zeichenströme



Caesar-Code: Generalisierung

```
void caesar(std::istream& in,
            std::ostream& out,
            int s) {

    in >> std::noskipws;

    char next;
    while (in >> next) {
        out << shift(next, s);
    }
}
```

■ `std::istream/std::ostream`
ist ein *abstrakter Eingabe-/*
Ausgabestrom an `chars`

Caesar-Code: Generalisierung

```
void caesar(std::istream& in,
            std::ostream& out,
            int s) {

    in >> std::noskipws;

    char next;
    while (in >> next) {
        out << shift(next, s);
    }
}
```

- `std::istream/std::ostream` ist ein *abstrakter Eingabe-/Ausgabestrom* an `chars`
- Aufruf der Funktion erfolgt mit *konkreten* Strömen, z.B.:
 - Konsole: `std::cin/cout`
 - Dateien: `std::ifstream/ofstream`

Caesar-Code: Generalisierung, Beispiel 1

```
#include <iostream>
```

```
...
```

```
// in void main():
```

```
caesar(std::cin, std::cout, s);
```

Aufruf der generischen **caesar**-Funktion: Von **std::cin** nach **std::cout**

Caesar-Code: Generalisierung, Beispiel 2

```
#include <iostream>
#include <fstream>
...

// in void main():
std::string to_file_name = ...; // Name of file to write to
std::ofstream to(to_file_name); // Output file stream

caesar(std::cin, to, s);
```

Aufruf der generischen **caesar**-Funktion: Von **std::cin** zu Datei

Caesar-Code: Generalisierung, Beispiel 3

```
#include <iostream>
#include <fstream>
...

// in void main():
std::string from_file_name = ...; // Name of file to read from
std::string to_file_name = ...; // Name of file to write to
std::ifstream from(from_file_name); // Input file stream
std::ofstream to(to_file_name); // Output file stream

caesar(from, to, s);
```

Aufruf der generischen **caesar**-Funktion: Von Datei zu Datei

Ströme: Abschluss

Hinweis: Sie müssen Ströme nur *anwenden* können

- *Anwenderwissen*, auf dem Niveau der vorherigen Folien, reicht für Hausaufgaben und Prüfung aus
- D.h. Sie müssen nicht wissen, wie Ströme intern funktionieren
- Wie sie selbst *abstrakte* und dazu passende *konkrete Typen* erstellen können, erfahren Sie ganz am Ende dieses Kurses

- Text „**Sein oder nicht sein**“ könnte als `vector<char>` repräsentiert werden

- Text „**Sein oder nicht sein**“ könnte als `vector<char>` repräsentiert werden
- Texte sind jedoch allgegenwärtig, daher existiert in der Standardbibliothek ein eigener Typ für sie: `std::string` (Zeichenkette)
- Benutzung benötigt `#include <string>`

Benutzung von `std::string`

- Deklaration und Initialisierung mittels Literal:

```
std::string text = "Essen ist fertig!"
```

Benutzung von `std::string`

- Deklaration und Initialisierung mittels Literal:

```
std::string text = "Essen ist fertig!"
```

- Mit variabler Länge initialisieren:

```
std::string text(n, 'a')
```

Benutzung von `std::string`

- Deklaration und Initialisierung mittels Literal:

```
std::string text = "Essen ist fertig!"
```

- Mit variabler Länge initialisieren:

```
std::string text(n, 'a')
```

- Texte vergleichen:

```
if (text1 == text2) ...
```


Benutzung von `std::string`

- Grösse auslesen:

```
for (unsigned int i = 0; i < text.size(); ++i) ...
```

Benutzung von `std::string`

- Grösse auslesen:

```
for (unsigned int i = 0; i < text.size(); ++i) ...
```

- Einzelne Zeichen lesen:

```
if (text[0] == 'a') ... // or text.at(0)
```

Benutzung von `std::string`

- Grösse auslesen:

```
for (unsigned int i = 0; i < text.size(); ++i) ...
```

- Einzelne Zeichen lesen:

```
if (text[0] == 'a') ... // or text.at(0)
```

- Einzelne Zeichen schreiben:

```
text[0] = 'b'; // or text.at(0)
```

Benutzung von `std::string`

- Strings konkatenieren (zusammensetzen):

```
text = ":-";  
text += ")";  
assert(text == ":-)");
```

- Viele weitere Operationen, bei Interesse siehe <https://en.cppreference.com/w/cpp/string>

15. Vektoren II

Mehrdimensionale Vektoren/Vektoren von Vektoren, Kürzeste Wege, Vektoren als Funktionsargumente

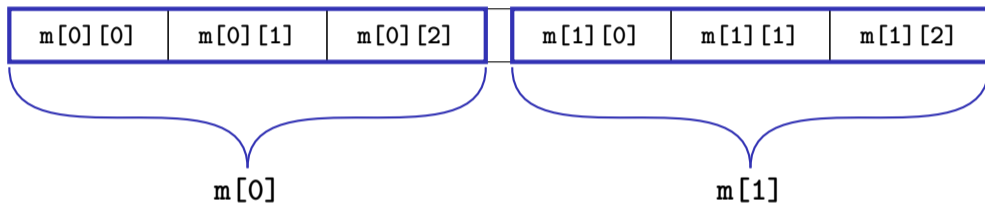
Mehrdimensionale Vektoren

- Zum Speichern von mehrdimensionalen Strukturen wie Tabellen, Matrizen, ...
- ...können *Vektoren von Vektoren* verwendet werden:

```
std::vector<std::vector<int>> m; // An empty matrix
```

Mehrdimensionale Vektoren

Im Speicher: flach



Mittels Initialisierungslisten:

```
// A 3-by-5 matrix
std::vector<std::vector<std::string>> m = {
    {"ZH", "BE", "LU", "BS", "GE"},
    {"FR", "VD", "VS", "NE", "JU"},
    {"AR", "AI", "OW", "IW", "ZG"}
};

assert(m[1][2] == "VS");
```

Auf bestimmte Größe füllen:

```
unsigned int a = ...;
unsigned int b = ...;

// An a-by-b matrix with all ones
std::vector<std::vector<int>>
    m(a, std::vector<int>(b, 1));
```

Auf bestimmte Grösse füllen:

```
unsigned int a = ...;  
unsigned int b = ...;  
  
// An a-by-b matrix with all ones  
std::vector<std::vector<int>>  
  m(a, std::vector<int>(b, 1));
```

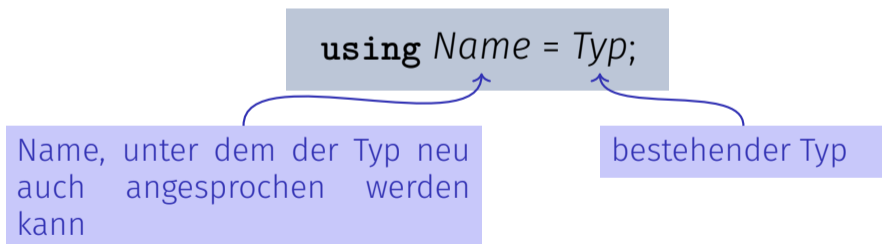
(Es gibt noch viele weitere Wege, Vektoren zu initialisieren)

Mehrdimensionale Vektoren und Typ- Aliasse

- Auch möglich: Vektoren von Vektoren von Vektoren von ...:
`std::vector<std::vector<std::vector<...>>>`
- Typnamen können offensichtlich laaaaaaaang werden

Mehrdimensionale Vektoren und Typ-Aliasse

- Auch möglich: Vektoren von Vektoren von Vektoren von ...:
`std::vector<std::vector<std::vector<...>>>`
- Typnamen können offensichtlich laaaaaaaang werden
- Dann hilft die Deklaration eines *Typ-Alias*:



Typ-Alias: Beispiel

```
#include <iostream>
#include <vector>
using imatrix = std::vector<std::vector<int>>;

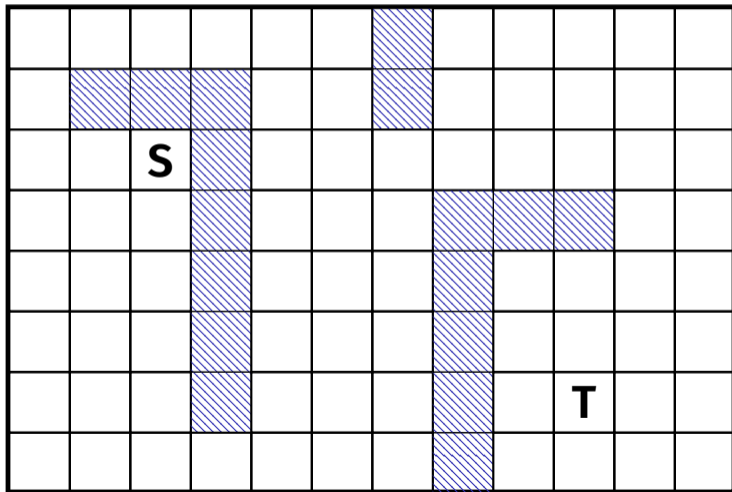
// POST: Matrix 'm' was output to stream 'out'
void print(const imatrix& m, std::ostream& out);

int main() {
    imatrix m = ...;
    print(m, std::cout);
}
```

Erinnerung: **const**-Referenz für Effizienz (keine Kopie) und Sicherheit (unveränderlich)

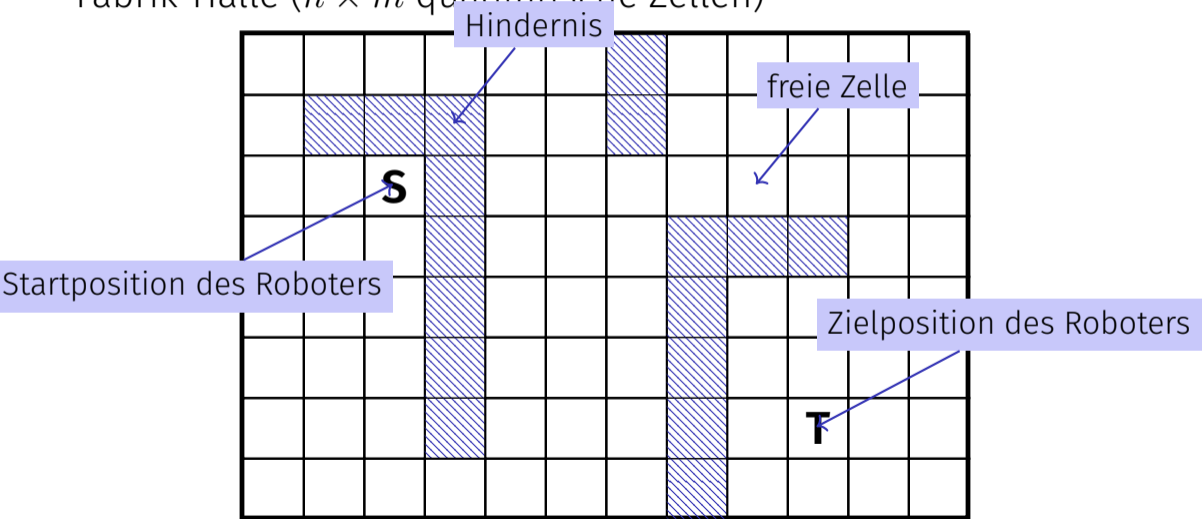
Anwendung: Kürzeste Wege

Fabrik-Halle ($n \times m$ quadratische Zellen)



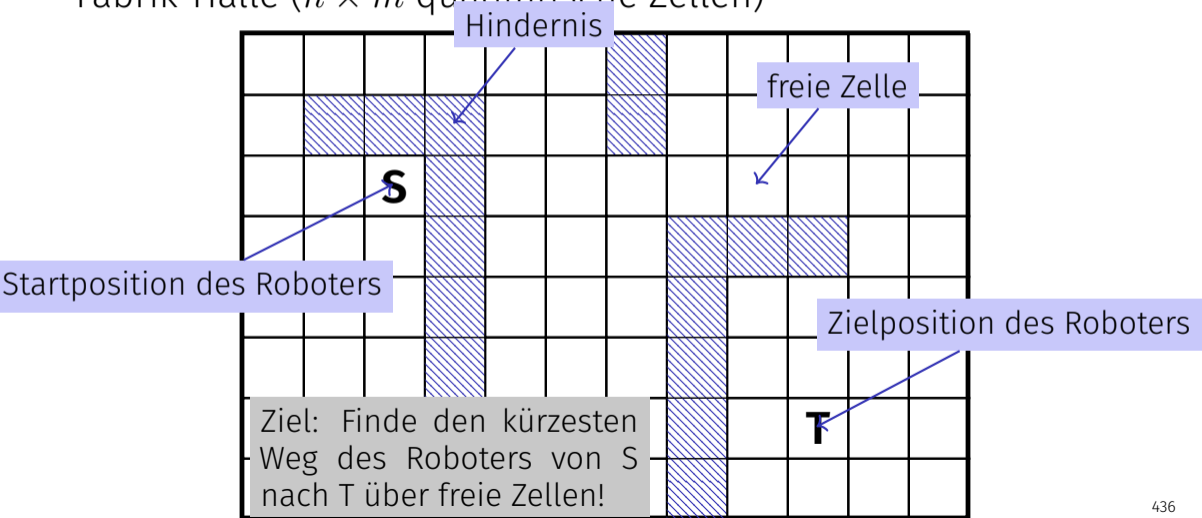
Anwendung: Kürzeste Wege

Fabrik-Halle ($n \times m$ quadratische Zellen)



Anwendung: Kürzeste Wege

Fabrik-Halle ($n \times m$ quadratische Zellen)



Ein (scheinbar) anderes Problem

Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen

4	5	6	7	8	9		15	16	17	18	19
3				9	10		14	15	16	17	18
2	1	0		10	11	12	13	14	15	16	17
3	2	1		11	12	13				17	18
4	3	2		10	11	12		20	19	18	19
5	4	3		9	10	11		21	20	19	20
6	5	4		8	9	10		22	21	20	21
7	6	5	6	7	8	9		23	22	21	22

Ein (scheinbar) anderes Problem

Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen

4	5	6	7	8	9		15	16	17	18	19
3				9	10		14	15	16	17	18
2	1	0		10	11	12	13	14	15	16	17
3	2	1		11	12	13				17	18
4	3	2		10	11	12		20	19	18	19
5	4	3		9	10	11		21	20	19	20
								22	21	20	21
								23	22	21	22

Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen

4	5	6	7	8	9		15	16	17	18	19
3				9	10		14	15	16	17	18
2	1	0		10	11	12	13	14	15	16	17
3	2	1								17	18
								20	19	18	19
5	4	3		9	10	11		21	20	19	20
								22	21	20	21
								23	22	21	22

Zielposition.
Kürzester Weg:
Länge 21

Startposition

Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen

4	5	6	7	8	9		15	16	17	18	19
3				9	10		14	15	16	17	18
2	1	0		10	11	12	13	14	15	16	17
3	2	1								17	18
								20	19	18	19
5	4	3		9	10	11		21	20	19	20
								22	21	20	21
								23	22	21	22

Startposition

Zielposition.
Kürzester Weg:
Länge 21

Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



Ein (scheinbar) anderes Problem

Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



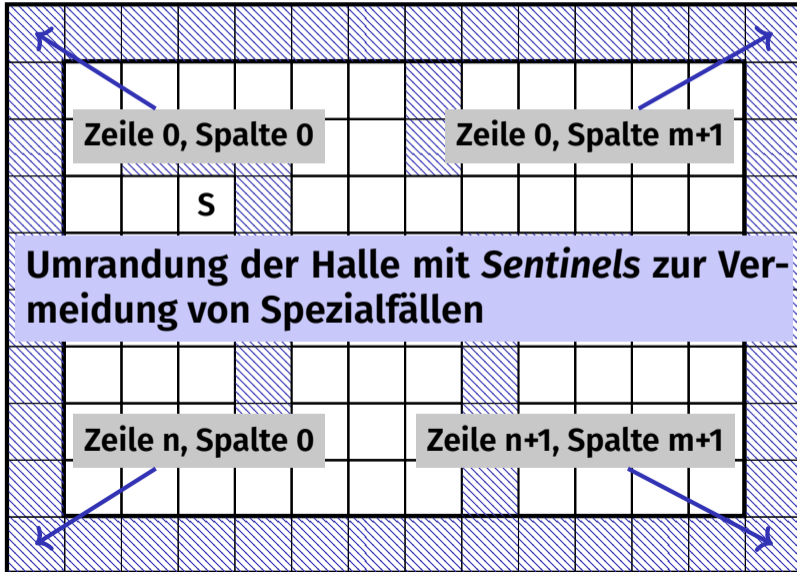
Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

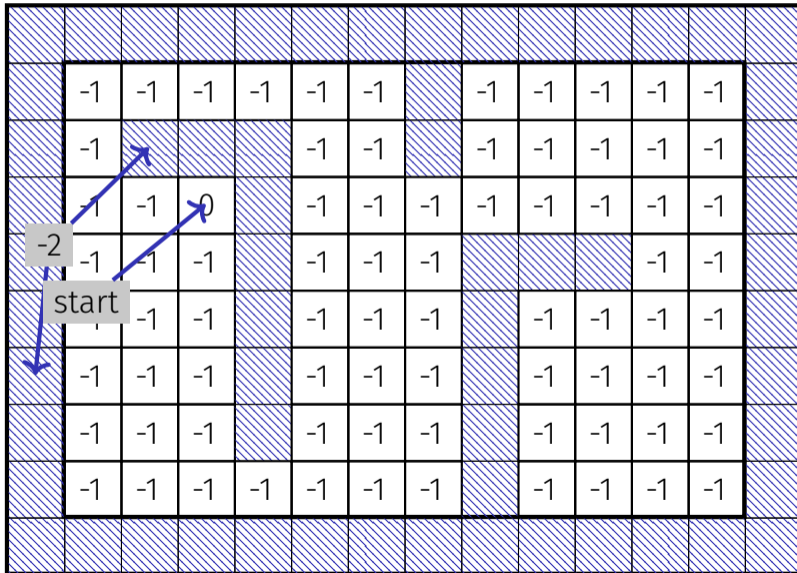
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen

4	5	6	7	8	9		15	16	17	18	19
3				9	10		14	15	16	17	18
2	1	0		10	11	12	13	14	15	16	17
3	2	1		11	12	13				17	18
4	3	2		10	11	12		20	19	18	19
5	4	3		9	10	11		21	20	19	20
6	5	4		8	9	10		22	21	20	21
7	6	5	6	7	8	9		23	22	21	22

Vorbereitung: Wächter (*Sentinels*)

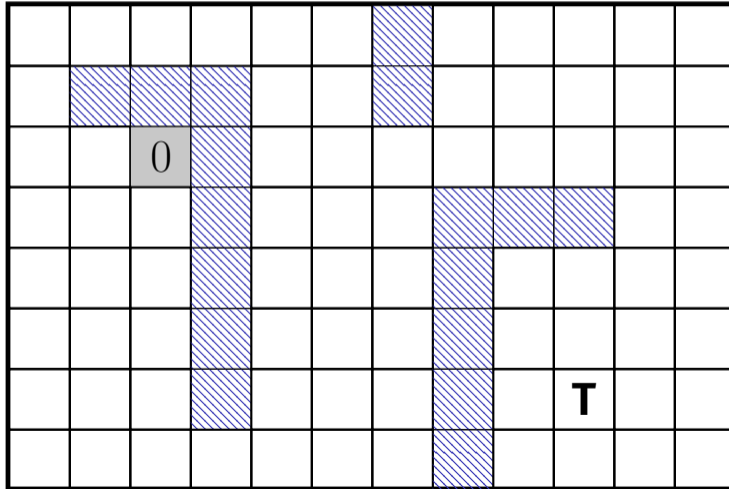


Vorbereitung: Initiale Markierung



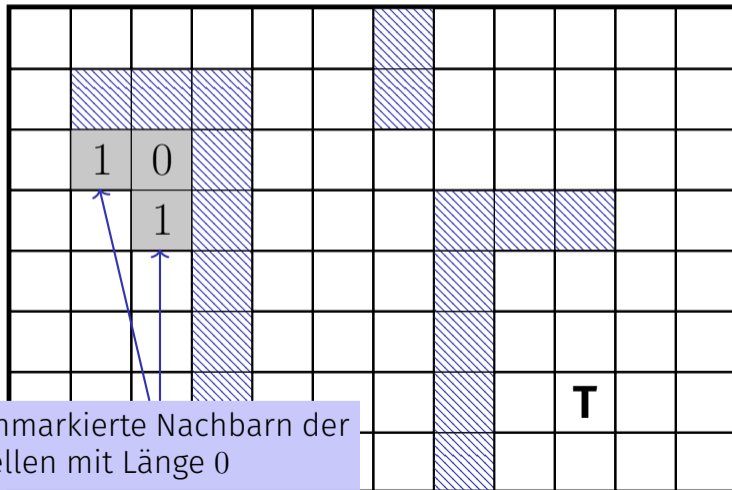
Markierung aller Zellen mit ihren Weglängen

Schritt 0: Alle Zellen mit Weglänge 0



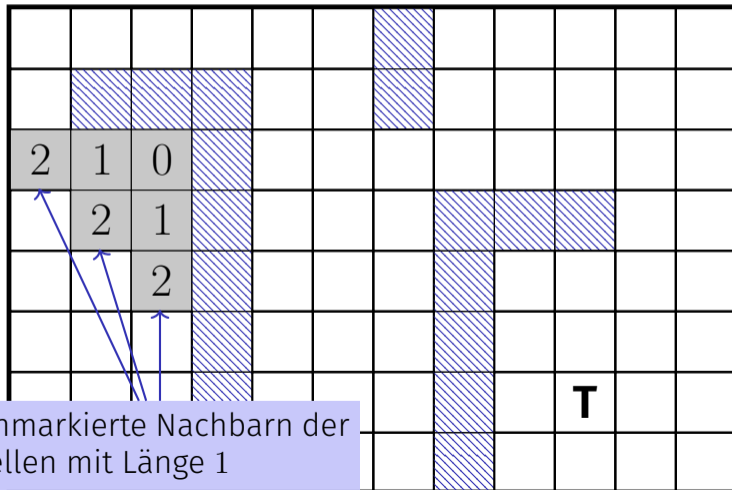
Markierung aller Zellen mit ihren Weglängen

Schritt 1: Alle Zellen mit Weglänge 1



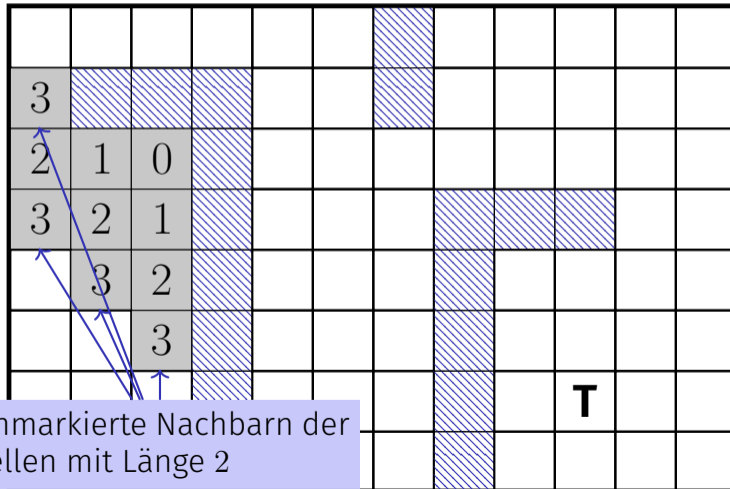
Markierung aller Zellen mit ihren Weglängen

Schritt 2: Alle Zellen mit Weglänge 2



Markierung aller Zellen mit ihren Weglängen

Schritt 3: Alle Zellen mit Weglänge 3



Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3, \dots$

```
for (int i=1;; ++i) {
    bool progress = false;
    for (int r=1; r<n+1; ++r)
        for (int c=1; c<m+1; ++c) {
            if (floor[r][c] != -1) continue;
            if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||
                floor[r][c-1] == i-1 || floor[r][c+1] == i-1 ) {
                floor[r][c] = i; // label cell with i
                progress = true;
            }
        }
    if (!progress) break;
}
```


Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3, \dots$

```
for (int i=1;; ++i) {  
    bool progress = false; ← zeigt an, ob in einem Durchlauf durch  
    for (int r=1; r<n+1; ++r) alle Zellen Fortschritt gemacht wurde  
        for (int c=1; c<m+1; ++c) {  
            if (floor[r][c] != -1) continue;  
            if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||  
                floor[r][c-1] == i-1 || floor[r][c+1] == i-1 ) {  
                floor[r][c] = i; // label cell with i  
                progress = true;  
            }  
        }  
    if (!progress) break;  
}
```

Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3, \dots$

```
for (int i=1;; ++i) {
    bool progress = false;
    for (int r=1; r<n+1; ++r) ← Gehe über alle Zellen
        for (int c=1; c<m+1; ++c) {
            if (floor[r][c] != -1) continue;
            if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||
                floor[r][c-1] == i-1 || floor[r][c+1] == i-1 ) {
                floor[r][c] = i; // label cell with i
                progress = true;
            }
        }
    if (!progress) break;
}
```

Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3, \dots$

```
for (int i=1;; ++i) {
    bool progress = false;
    for (int r=1; r<n+1; ++r)
        for (int c=1; c<m+1; ++c) {
            if (floor[r][c] != -1) continue;
            if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||
                floor[r][c-1] == i-1 || floor[r][c+1] == i-1 ) {
                floor[r][c] = i; // label cell with i
                progress = true;
            }
        }
    if (!progress) break;
}
```

Zelle schon markiert oder Hindernis

Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3, \dots$

```
for (int i=1;; ++i) {  
    bool progress = false;  
    for (int r=1; r<n+1; ++r)  
        for (int c=1; c<m+1; ++c) {  
            if (floor[r][c] != -1) continue;  
            if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||  
                floor[r][c-1] == i-1 || floor[r][c+1] == i-1 ) {  
                floor[r][c] = i; // label cell with i  
                progress = true;  
            }  
        }  
    if (!progress) break;  
}
```

Ein Nachbar hat Weglänge $i - 1$. Die Wächter garantieren immer 4 Nachbarn.

Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3, \dots$

```
for (int i=1;; ++i) {
    bool progress = false;
    for (int r=1; r<n+1; ++r)
        for (int c=1; c<m+1; ++c) {
            if (floor[r][c] != -1) continue;
            if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||
                floor[r][c-1] == i-1 || floor[r][c+1] == i-1 ) {
                floor[r][c] = i; // label cell with i
                progress = true;
            }
        }
    if (!progress) break;
}
```

Kein Fortschritt, alle erreichbaren Zellen markiert; fertig.

Das Kürzeste-Wege-Programm

- Algorithmus: *Breitensuche* (Breiten- vs. *Tiefensuche* wird typischerweise in Algorithmen-Vorlesungen diskutiert)

Das Kürzeste-Wege-Programm

- Algorithmus: *Breitensuche* (Breiten- vs. *Tiefensuche* wird typischerweise in Algorithmen-Vorlesungen diskutiert)
- Das Programm kann recht langsam sein, weil für jedes i alle Zellen durchlaufen werden

Das Kürzeste-Wege-Programm

- Algorithmus: *Breitensuche* (Breiten- vs. *Tiefensuche* wird typischerweise in Algorithmen-Vorlesungen diskutiert)
- Das Programm kann recht langsam sein, weil für jedes i alle Zellen durchlaufen werden
- Verbesserung: Für Markierung i , durchlaufe nur die Nachbarn der Zellen mit Markierung $i - 1$
- Verbesserung: Stoppe, sobald das Ziel erreicht wurde

16. Rekursion 1

Mathematische Rekursion, Terminierung, der Aufrufstapel, Beispiele, Rekursion vs. Iteration, n-Damen Problem

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich *rekursiv* definierbar

Mathematische Rekursion

- Viele mathematische Funktionen sind sehr natürlich *rekursiv* definierbar
- Das heisst, die Funktion erscheint in ihrer eigenen Definition

$$n! = \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot (n - 1)!, & \text{andernfalls} \end{cases}$$

Rekursion in C++: Genauso!

$$n! = \begin{cases} 1, & \text{falls } n \leq 1 \\ n \cdot (n - 1)!, & \text{andernfalls} \end{cases}$$

```
// POST: return value is n!  
unsigned int fac(unsigned int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife ...

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife ...
- ...nur noch schlechter: „verbrennt“ Zeit *und* Speicher

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife ...
- ...nur noch schlechter: „verbrennt“ Zeit *und* Speicher

```
void f() {  
    f() // f() → f() → ... → stack overflow  
}
```

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife ...
- ...nur noch schlechter: „verbrennt“ Zeit *und* Speicher

```
void f() {  
    f() // f() → f() → ... → stack overflow  
}
```

Ein Euro ist ein Euro.

Wim Duisenberg, erster Präsident der EZB

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir **garantierten Fortschritt**
Richtung einer Abbruchbedingung (\approx Basisfall)

Beispiel `fac(n)`:

- Rekursion endet falls $n \leq 1$
- Rekursiver Aufruf mit neuem Argument $< n$
- Abbruchbedingung wird daher garantiert erreicht

```
unsigned int fac(  
    unsigned int n) {  
  
    if (n <= 1)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

Rekursive Funktionen: Auswertung

```
int fac(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return n * fac(n-1);  
}  
  
...  
std::cout << fac(4);
```

Rekursive Funktionen: Auswertung

```
int fac(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return n * fac(n-1);  
}  
  
...  
std::cout << fac(4);
```

fac(4)

Aufruf von **fac(4)**

Rekursive Funktionen: Auswertung

```
int fac(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return n * fac(n-1);  
}  
  
...  
std::cout << fac(4);
```

`fac(4)` \rightsquigarrow `int n = 4`

Aufruf von `fac(4)` \rightsquigarrow Initialisierung des formalen Arguments **n**

Rekursive Funktionen: Auswertung

```
int fac(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return n * fac(n-1);  
}  
  
...  
std::cout << fac(4);
```

`fac(4) \rightsquigarrow int n = 4`

Auswertung des Rückgabeausdrucks

Rekursive Funktionen: Auswertung

```
int fac(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return n * fac(n-1);  
}  
  
...  
std::cout << fac(4);
```

$\text{fac}(4) \rightsquigarrow \text{int } n = 4$
 $\hookrightarrow \text{fac}(n - 1)$

Rekursiver Aufruf mit Argument $n - 1 = 4 - 1 = 3$

Rekursive Funktionen: Auswertung

```
int fac(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return n * fac(n-1);  
}  
  
...  
std::cout << fac(4);
```

$\text{fac}(4) \rightsquigarrow \text{int } n = 4$

$\hookrightarrow \text{fac}(n - 1) \rightsquigarrow \text{int } n = 3$

Initialisierung des formalen Arguments **n**

Rekursive Funktionen: Auswertung

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

...

```
std::cout << fac(4);
```

$\text{fac}(4) \rightsquigarrow \text{int } n = 4$

$\hookrightarrow \text{fac}(n - 1) \rightsquigarrow \text{int } n = 3$

:

*Jeder Aufruf von **fac** arbeitet mit seinem eigenen **n***

Der Aufrufstapel

```
std::cout << fac(4)
```

Der Aufrufstapel

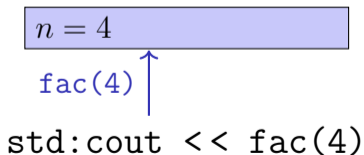
Bei jedem Funktionsaufruf:

```
    fac(4) ↑  
std::cout << fac(4)
```

Der Aufrufstapel

Bei jedem Funktionsaufruf:

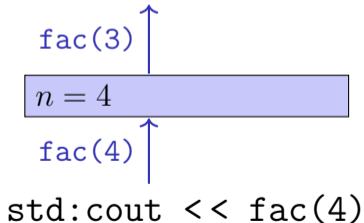
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

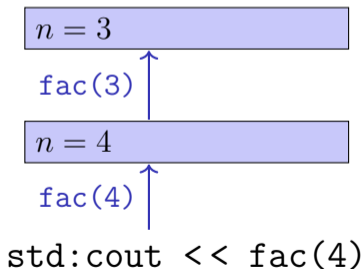
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

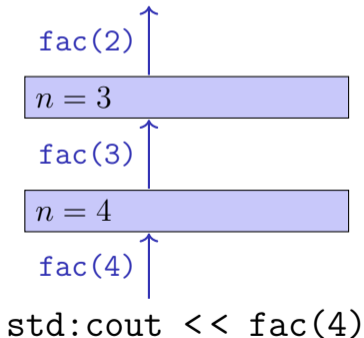
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

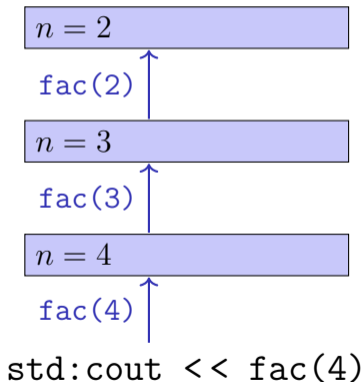
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

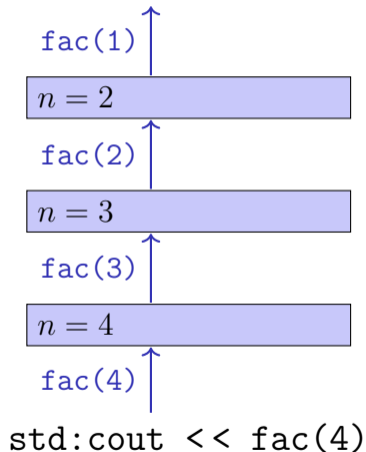
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

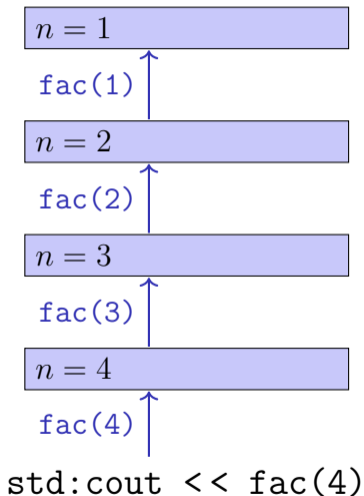
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

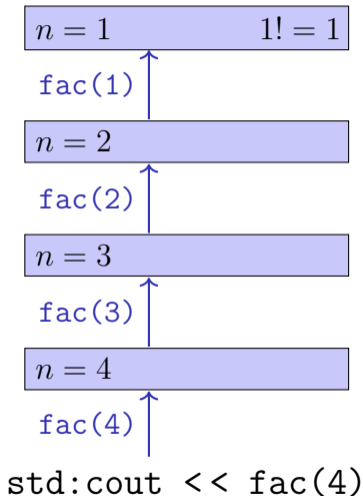
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

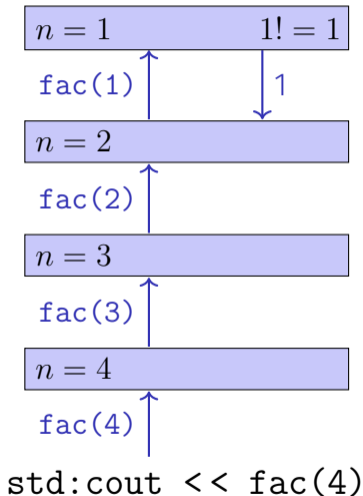
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet



Der Aufrufstapel

Bei jedem Funktionsaufruf:

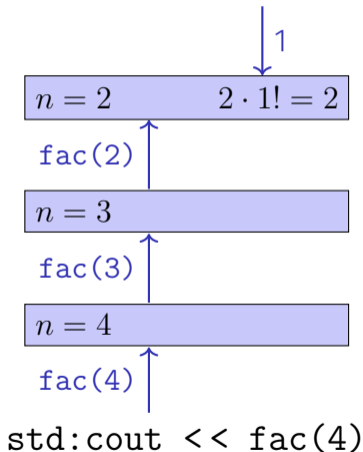
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

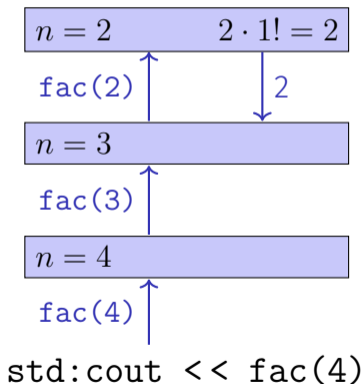
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

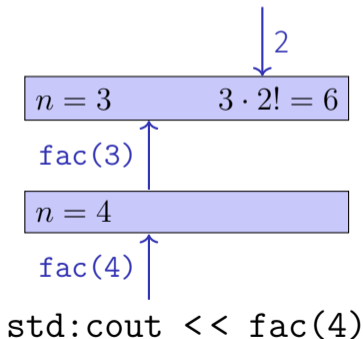
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

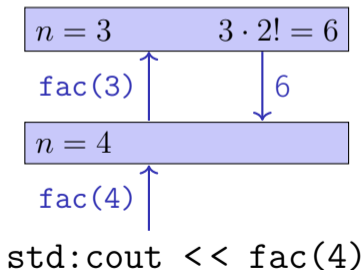
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

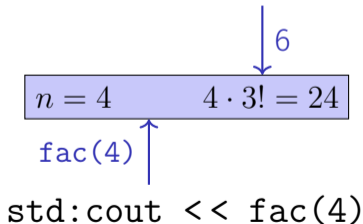
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

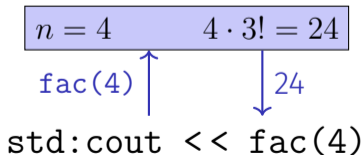
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht




Der Aufrufstapel

Bei jedem Funktionsaufruf:

- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht

`std::cout << fac(4)`



Fibonacci-Zahlen

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

Fibonacci-Zahlen

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...

Fibonacci-Zahlen in Zürich



Fibonacci-Zahlen in C++

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

```
unsigned int fib(unsigned int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return fib(n-1) + fib(n-2); // n > 1  
}
```

Fibonacci-Zahlen in C++

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

```
unsigned int fib(unsigned int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return fib(n-1) + fib(n-2); // n > 1  
}
```

Fibonacci-Zahlen in C++

Laufzeit

`fib(50)` dauert „ewig“, denn es berechnet
 F_{48} 2-mal, F_{47} 3-mal, F_{46} 5-mal, F_{45} 8-mal, F_{44} 13-mal,
 F_{43} 21-mal ... F_1 ca. 10^9 mal (!)

```
unsigned int fib(unsigned int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return fib(n-1) + fib(n-2); // n > 1  
}
```


Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \dots, F_n$

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \dots, F_n$
- Speichere jeweils die zwei letzten berechneten Fibonacci-Zahlen (Variablen **a** und **b**)

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \dots, F_n$
- Speichere jeweils die zwei letzten berechneten Fibonacci-Zahlen (Variablen **a** und **b**)
- Berechne die nächste Zahl als Summe von **a** und **b**

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \dots, F_n$
- Speichere jeweils die zwei letzten berechneten Fibonacci-Zahlen (Variablen **a** und **b**)
- Berechne die nächste Zahl als Summe von **a** und **b**

Kann rekursiv und iterativ implementiert werden, letzteres ist einfacher/direkter

Schnelle Fibonacci-Zahlen in C++

```
unsigned int fib(unsigned int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
  
    unsigned int a = 0; // F_0  
    unsigned int b = 1; // F_1  
  
    for (unsigned int i = 2; i <= n; ++i) {  
        unsigned int a_old = a; //  $F_{i-2}$   
        a = b; // a becomes  $F_{i-1}$   
        b += a_old; // b becomes  $F_{i-1} + F_{i-2}$ , i.e.  $F_i$   
    }  
    return b;  
}
```

$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$

Schnelle Fibonacci-Zahlen in C++

```
unsigned int fib(unsigned int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
  
    unsigned int a = 0; // F_0  
    unsigned int b = 1; // F_1  
  
    for (unsigned int i = 2; i <= n; ++i) {  
        unsigned int a_old = a; //  $F_{i-2}$   
        a = b; // a becomes  $F_{i-1}$   
        b += a_old; // b becomes  $F_{i-1} + F_{i-2}$ , i.e.  $F_i$   
    }  
    return b;  
}
```

$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$

Schnelle Fibonacci-Zahlen in C++

```
unsigned int fib(unsigned int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
  
    unsigned int a = 0; // F_0  
    unsigned int b = 1; // F_1  
  
    for (unsigned int i = 2; i <= n; ++i) {  
        unsigned int a_old = a; //  $F_{i-2}$   
        a = b; // a becomes  $F_{i-1}$   
        b += a_old; // b becomes  $F_{i-1} + F_{i-2}$ , i.e.  $F_i$   
    }  
    return b;  
}
```

$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$

Schnelle Fibonacci-Zahlen in C++

```
unsigned int fib(unsigned int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
  
    unsigned int a = 0; // F_0  
    unsigned int b = 1; // F_1  
  
    for (unsigned int i = 2; i <= n; ++i) {  
        unsigned int a_old = a; // F_{i-2}  
        a = b; // a becomes F_{i-1}  
        b += a_old; // b becomes F_{i-1} + F_{i-2}, i.e. F_i  
    }  
    return b;  
}
```

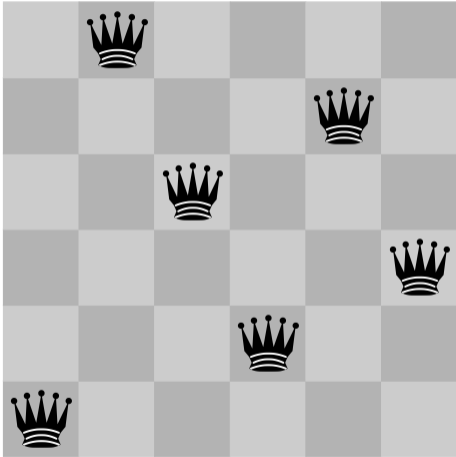
sehr schnell auch bei `fib(50)`

$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$

Die Macht der Rekursion

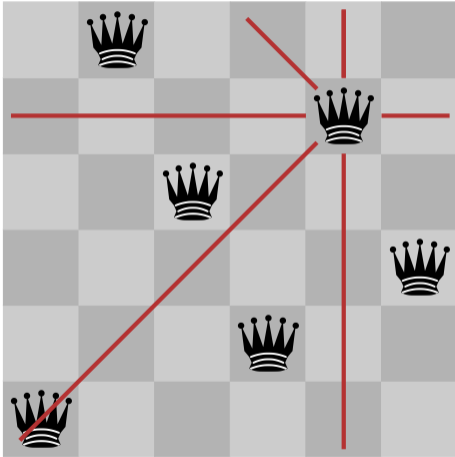
- Einige Probleme scheinen ohne Rekursion kaum lösbar zu sein. Mit Rekursion werden sie plötzlich deutlich einfacher lösbar.
- Beispiele: *das n -Damen-Problem*, Die Türme von Hanoi, Parsen von Ausdrücken, Sudoku-Löser, Umgekehrte Aus- oder Eingabe, Suchen in Bäumen, Divide-And-Conquer (z.B. Sortieren) , ...
- ...sowie die 2. Bonusaufgabe: Nonogramme

Das n -Damen Problem



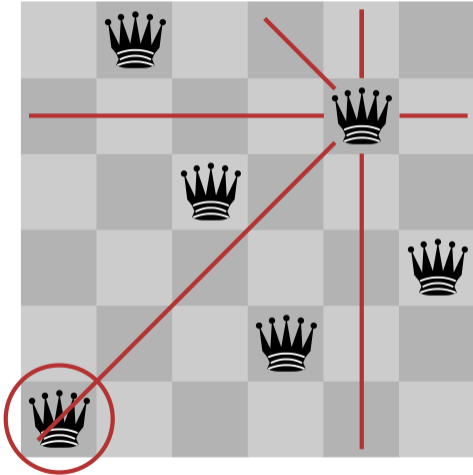
- Gegeben sei ein $n \times n$ Schachbrett
- Zum Beispiel $n = 6$
- Frage: ist es möglich n Damen so zu platzieren, dass keine zwei Damen sich bedrohen?

Das n -Damen Problem



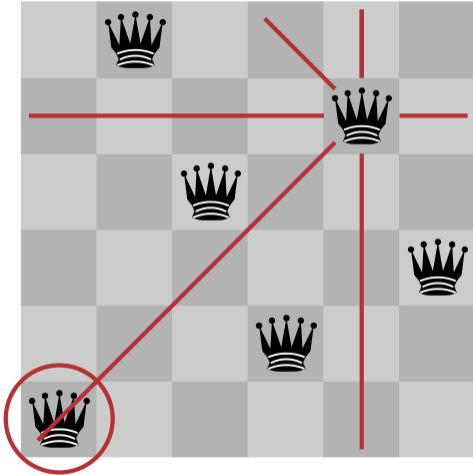
- Gegeben sei ein $n \times n$ Schachbrett
- Zum Beispiel $n = 6$
- Frage: ist es möglich n Damen so zu platzieren, dass keine zwei Damen sich bedrohen?

Das n -Damen Problem



- Gegeben sei ein $n \times n$ Schachbrett
- Zum Beispiel $n = 6$
- Frage: ist es möglich n Damen so zu platzieren, dass keine zwei Damen sich bedrohen?

Das n -Damen Problem



- Gegeben sei ein $n \times n$ Schachbrett
- Zum Beispiel $n = 6$
- Frage: ist es möglich n Damen so zu platzieren, dass keine zwei Damen sich bedrohen?
- Falls ja, wie viele Lösungen gibt es?

Lösung?

- Durchprobieren aller Möglichkeiten?

Lösung?

- Durchprobieren aller Möglichkeiten?
- $\binom{n^2}{n}$ Möglichkeiten. Zu viele!

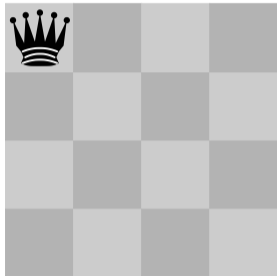
Lösung?

- Durchprobieren aller Möglichkeiten?
- $\binom{n^2}{n}$ Möglichkeiten. Zu viele!
- Nur eine Dame pro Zeile: n^n Möglichkeiten. Besser – aber auch noch zu viele.

Lösung?

- Durchprobieren aller Möglichkeiten?
- $\binom{n^2}{n}$ Möglichkeiten. Zu viele!
- Nur eine Dame pro Zeile: n^n Möglichkeiten. Besser – aber auch noch zu viele.
- Idee: Unsinnige Versuche gar nicht erst weiterverfolgen, stattdessen falsche Züge zurücknehmen \Rightarrow *Backtracking*

Lösung mit Backtracking

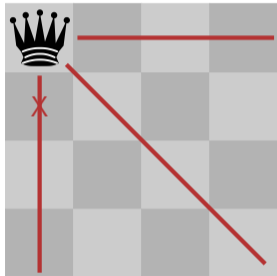


Erste Dame

queens

0
0
0
0

Lösung mit Backtracking

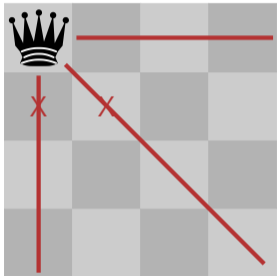


Verbotene Felder:
hier dürfen keine
anderen Damen
stehen.

queens

0
0
0
0

Lösung mit Backtracking

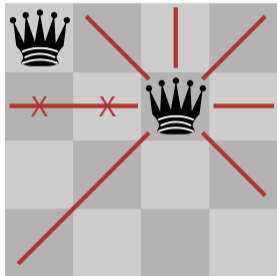


Verbotene Felder:
hier dürfen keine
anderen Damen
stehen.

queens

0
1
0
0

Lösung mit Backtracking

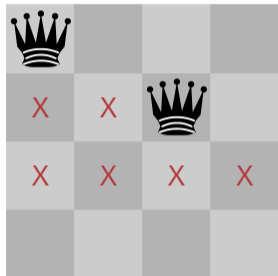


Nächste Dame in
nächster Zeile
(keine Kollision)

queens

0
2
0
0

Lösung mit Backtracking

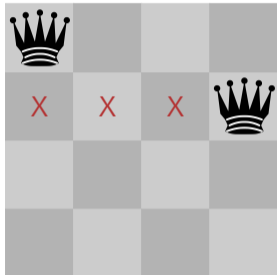


Alle Felder in nächster Zeile verboten. Zurück! (Backtracking!)

queens

0
2
4
0

Lösung mit Backtracking

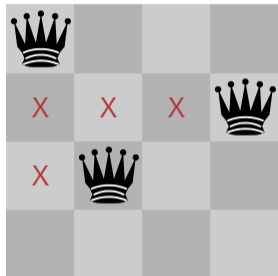


Dame eins weiter
setzen und wieder
versuchen

queens

0
3
0
0

Lösung mit Backtracking

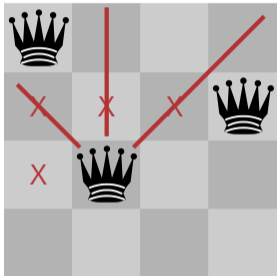


Nächste Zeile

queens

0
3
1
0

Lösung mit Backtracking

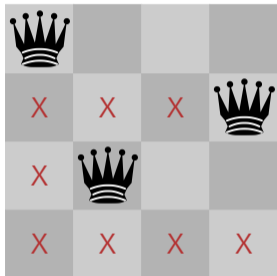


Ok (nur bereits
gesetzte Damen
müssen getestet
werden)

queens

0
3
1
0

Lösung mit Backtracking



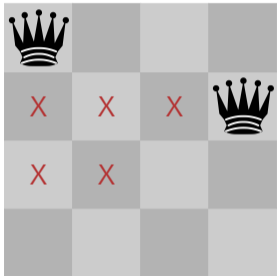
Alle
der
Zeile
Zurück.

Felder
nächsten
verboten.

queens

0
3
1
4

Lösung mit Backtracking

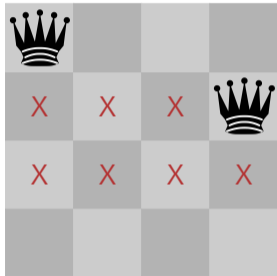


Weiter in der
vorigen Zeile

queens

0
3
1
0

Lösung mit Backtracking

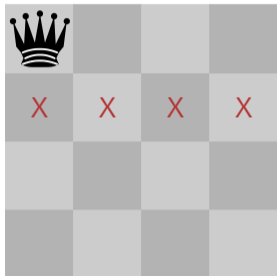


Alle restlichen
Felder auch ver-
boten. Weiter
zurück (back-
tracking)

queens

0
3
4
0

Lösung mit Backtracking

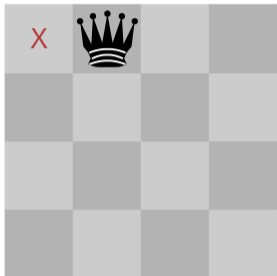


Alle Felder dieser Zeile führten zu keiner Lösung. Weiter zurück (back-tracking)

queens

0
4
0
0

Lösung mit Backtracking

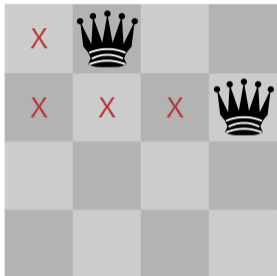


Setze Dame wieder
eins weiter.

queens

1
0
0
0

Lösung mit Backtracking

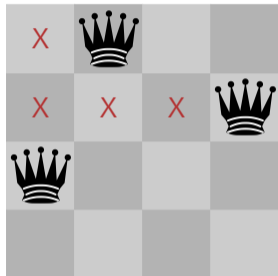


nächste Zeile

queens

1
3
0
0

Lösung mit Backtracking

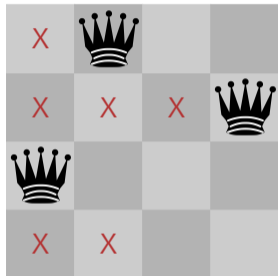


nächste Zeile

queens

1
3
0
0

Lösung mit Backtracking

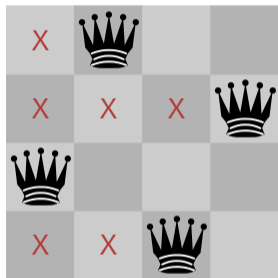


nächste Zeile

queens

1
3
0
1

Lösung mit Backtracking

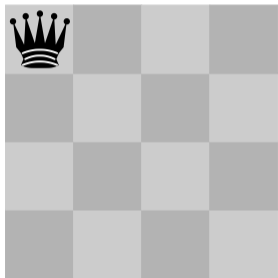


Lösung gefunden

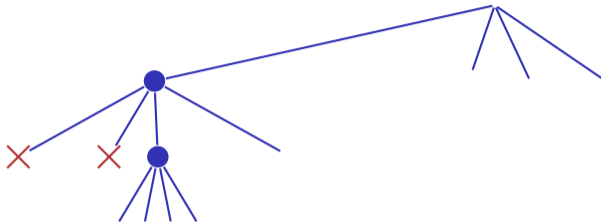
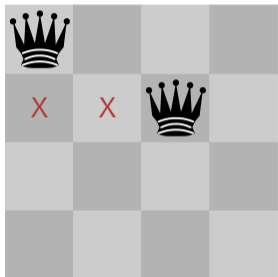
queens

1
3
0
2

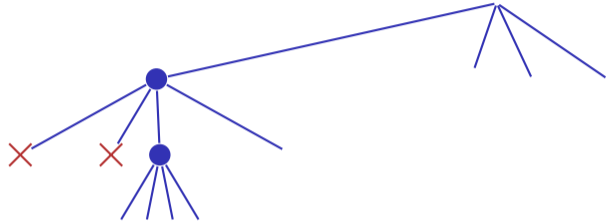
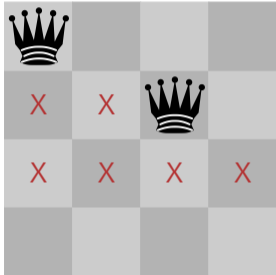
Suchstrategie als Baum visualisiert



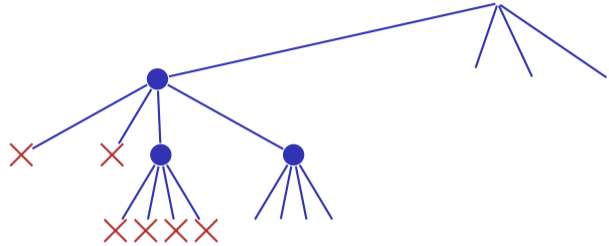
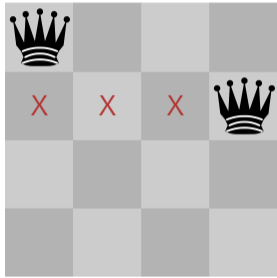
Suchstrategie als Baum visualisiert



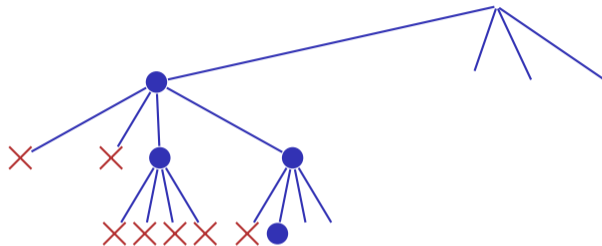
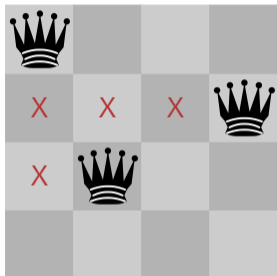
Suchstrategie als Baum visualisiert



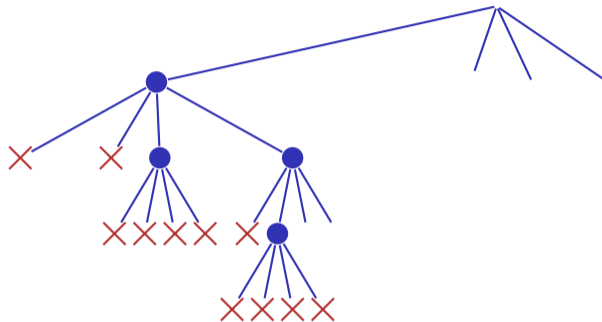
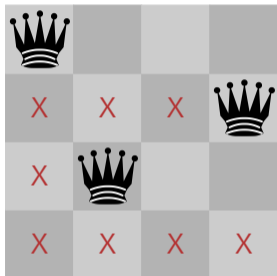
Suchstrategie als Baum visualisiert



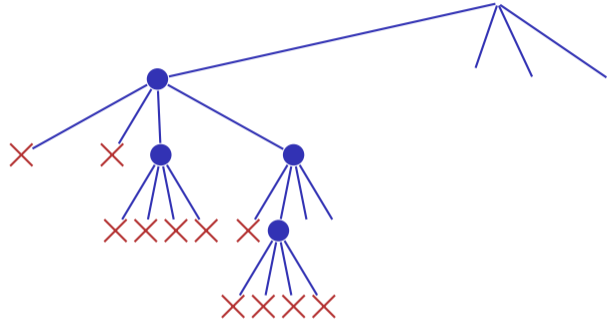
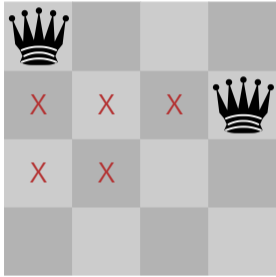
Suchstrategie als Baum visualisiert



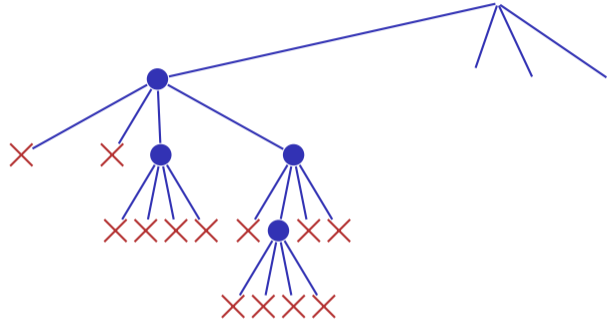
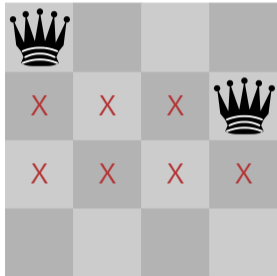
Suchstrategie als Baum visualisiert



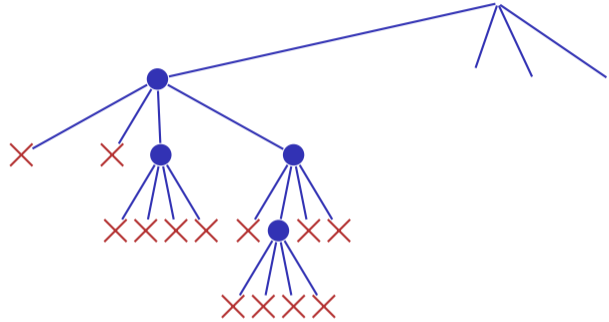
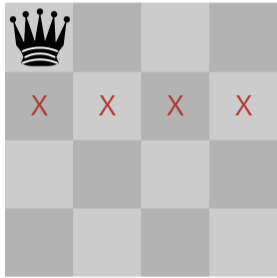
Suchstrategie als Baum visualisiert



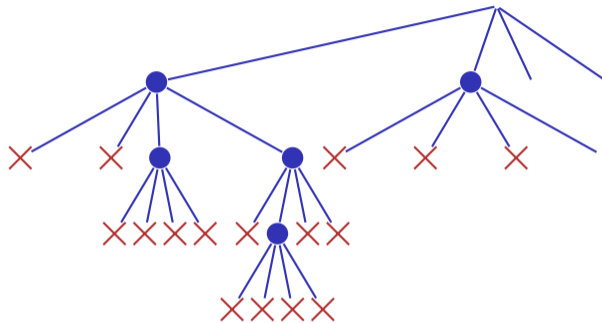
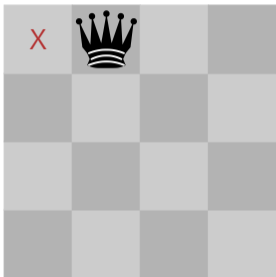
Suchstrategie als Baum visualisiert



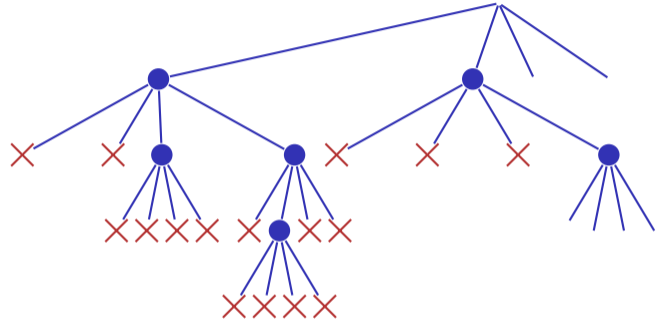
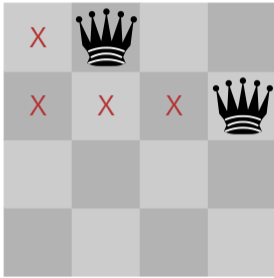
Suchstrategie als Baum visualisiert



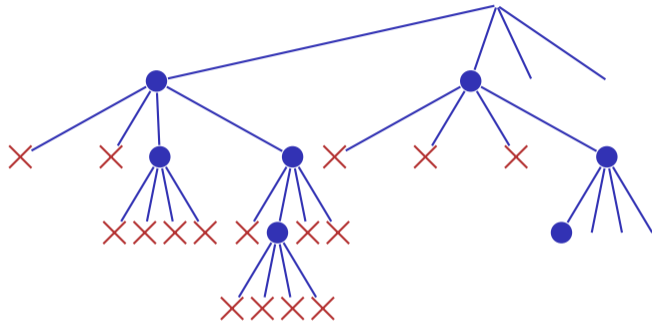
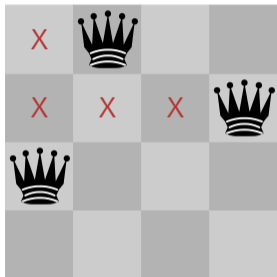
Suchstrategie als Baum visualisiert



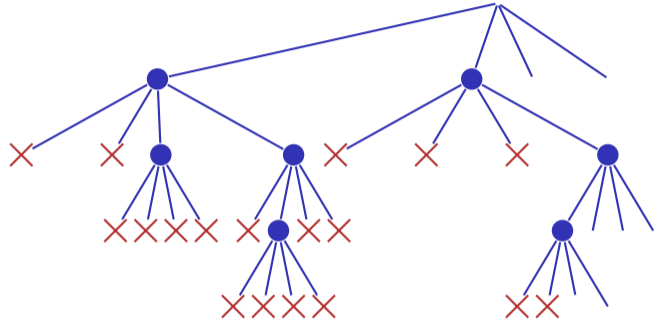
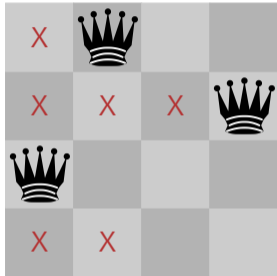
Suchstrategie als Baum visualisiert



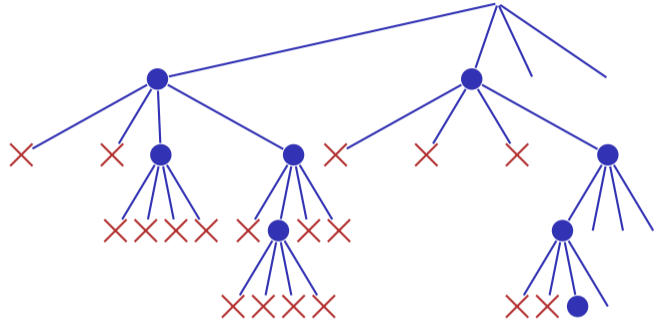
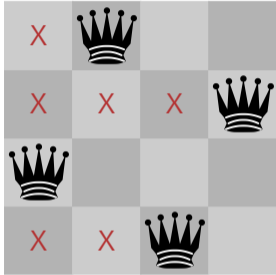
Suchstrategie als Baum visualisiert



Suchstrategie als Baum visualisiert



Suchstrategie als Baum visualisiert



Prüfe Dame

```
using Queens = std::vector<unsigned int>;

// post: returns if queen in the given row is valid, i.e.
//       does not share a common row, column or diagonal
//       with any of the queens on rows 0 to row-1
bool valid(const Queens& queens, unsigned int row) {
    unsigned int col = queens[row];
    for (unsigned int r = 0; r != row; ++r) {
        unsigned int c = queens[r];
        if (col == c || col - row == c - r || col + row == c + r)
            return false; // same column or diagonal
    }
    return true; // no shared column or diagonal
}
```

Rekursion: Finde eine Lösung

```
// pre: all queens from row 0 to row-1 are valid,  
//       i.e. do not share any common row, column or diagonal  
// post: returns if there is a valid position for queens on  
//       row .. queens.size(). if true is returned then the  
//       queens vector contains a valid configuration.  
bool solve(Queens& queens, unsigned int row) {  
    if (row == queens.size())  
        return true;  
    for (unsigned int col = 0; col != queens.size(); ++col) {  
        queens[row] = col;  
        if (valid(queens, row) && solve(queens, row+1))  
            return true; // (else check next position)  
    }  
    return false; // no valid configuration found  
}
```

Rekursion: Zähle alle Lösungen

```
// pre: all queens from row 0 to row-1 are valid,  
//   i.e. do not share any common row, column or diagonal  
// post: returns the number of valid configurations of the  
//   remaining queens on rows row ... queens.size()  
int nSolutions(Queens& queens, unsigned int row) {  
    if (row == queens.size())  
        return 1;  
    int count = 0;  
    for (unsigned int col = 0; col != queens.size(); ++col) {  
        queens[row] = col;  
        if (valid(queens, row))  
            count += nSolutions(queens, row+1);  
    }  
    return count;  
}
```

Hauptprogramm

```
// pre: positions of the queens in vector queens
// post: output of the positions of the queens in a graphical way
void print(const Queens& queens);

int main() {
    int n;
    std::cin >> n;
    Queens queens(n);
    if (solve(queens,0)) {
        print(queens);
        std::cout << "# solutions:" << nSolutions(queens,0) << std::endl;
    } else
        std::cout << "no solution" << std::endl;
    return 0;
}
```

17. Rekursion 2

Bau eines Taschenrechners, Formale Grammatiken, Extended Backus Naur Form (EBNF), Parsen von Ausdrücken

Motivation: Taschenrechner

Eingabe: 3 + 5

Ausgabe: 8

- Binäre Operatoren +, -, *, / und Zahlen

Motivation: Taschenrechner

Eingabe: 3 / 5

Ausgabe: 0.6

- Binäre Operatoren +, -, *, / und Zahlen
- Fließkommarithmetik

Motivation: Taschenrechner

Eingabe: $3 + 5 * 20$

Ausgabe: 103

- Binäre Operatoren $+$, $-$, $*$, $/$ und Zahlen
- Fließkommaarithmetik
- Präzedenzen und Assoziativitäten wie in C++

Motivation: Taschenrechner

Eingabe: $(3 + 5) * 20$

Ausgabe: 160

- Binäre Operatoren $+$, $-$, $*$, $/$ und Zahlen
- Fließkommaarithmetik
- Präzedenzen und Assoziativitäten wie in C++
- Klammerung

Motivation: Taschenrechner

Eingabe: $-(3 + 5) + 20$

Ausgabe: 12

- Binäre Operatoren $+$, $-$, $*$, $/$ und Zahlen
- Fließkommaarithmetik
- Präzedenzen und Assoziativitäten wie in C++
- Klammerung
- Unärer Operator $-$

Naiver Versuch (ohne Klammern)

```
double lval;
std::cin >> lval;

char op;
while (std::cin >> op && op != '=') {
    double rval;
    std::cin >> rval;

    if (op == '+')
        lval += rval;
    else if (op == '*')
        lval *= rval;
    else ...
}
std::cout << "Ergebnis " << lval << "\n";
```

Scheint zu klappen...

```
double lval;
std::cin >> lval;

char op;
while (std::cin >> op && op != '=') {
    double rval;
    std::cin >> rval;

    if (op == '+')
        lval += rval;
    else if (op == '*')
        lval *= rval;
    else ...
}

std::cout << "Ergebnis " << lval << "\n";
```

```
Eingabe 1 * 2 * 3 * 4 =
Ergebnis 24
```

Oops, Strich- vor Punktrechnung...

```
double lval;
std::cin >> lval;

char op;
while (std::cin >> op && op != '=') {
    double rval;
    std::cin >> rval;

    if (op == '+')
        lval += rval;
    else if (op == '*')
        lval *= rval;
    else ...
}
std::cout << "Ergebnis " << lval << "\n";
```

Eingabe 2 + 3 * 3 =
Ergebnis 15

Analyse des Problems

Eingabe:

13 + ...

Analyse des Problems

Eingabe:

$$13 + 4 * \dots$$

Analyse des Problems

Eingabe:

$$13 + 4 * (15 - ...$$


Analyse des Problems

Eingabe:

$$13 + 4 * (15 - 7 * ...$$

Analyse des Problems

Eingabe:

$$13 + 4 * (15 - 7 * 3) =$$


Muss gespeichert bleiben,
damit jetzt ausgewertet
werden kann!

Analyse des Problems

Ergebnis:

$$13 + 4*(15 - 21)$$

Analyse des Problems

Ergebnis:

$$13 + 4 * (-6)$$

Analyse des Problems

Ergebnis:

$$13 + (-24)$$

Analyse des Problems

Ergebnis:

-11

Analyse des Problems

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Diese

Analyse des Problems

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Diese Vorlesung

Analyse des Problems

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Diese Vorlesung ist

Analyse des Problems

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Diese Vorlesung ist insgesamt

Analyse des Problems

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Diese Vorlesung ist insgesamt recht

Analyse des Problems

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Diese Vorlesung ist insgesamt recht rekursiv.

Analyse des Problems

$$13 + 4 * (15 - 7 * 3)$$

Analyse des Problems

$$13 + 4 * (15 - 7 * 3)$$

Das „Verstehen“ eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Analyse des Problems

$$13 + 4 * (15 - 7 * 3)$$

Das „Verstehen“ eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Wir werden die Symbole elegant mittels Rekursion zwischenspeichern.

Analyse des Problems

$$13 + 4 * (15 - 7 * 3)$$

Das „Verstehen“ eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Wir werden die Symbole elegant mittels Rekursion zwischenspeichern.

Wir brauchen ein neues formales (von C++ unabhängiges) Handwerkszeug.

Formale Grammatiken

- Alphabet: endliche Menge von Symbolen
- Sätze: endlichen Folgen von Symbolen

Formale Grammatiken

- Alphabet: endliche Menge von Symbolen
- Sätze: endlichen Folgen von Symbolen

Eine formale Grammatik definiert, welche Sätze gültig sind.

Formale Grammatiken

- Alphabet: endliche Menge von Symbolen
- Sätze: endlichen Folgen von Symbolen

Eine formale Grammatik definiert, welche Sätze gültig sind.

Zur Beschreibung der Grammatik verwenden wir:

Extended Backus Naur Form (EBNF)

What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?

Niklaus Wirth
Federal Institute of Technology (ETH), Zürich, and
Xerox Palo Alto Research Center

Key Words and Phrases: syntactic description
language, extended BNF
CR Categories: 4.20

The population of programming languages is steadily growing, and there is no end of this growth in sight. Many language definitions appear in journals, many are found in technical reports, and perhaps an even greater number remains confined to proprietary circles. After frequent exposure to these definitions, one cannot fail to notice the lack of "common denominators." The only widely accepted fact is that the language structure is defined by a syntax. But even notation for syntactic description eludes any commonly agreed standard form, although the underlying ancestor is invariably the Backus-Naur Form of the Algol 60 report. As variations are often only slight, they become annoying for their very lack of an apparent motivation.

Out of sympathy with the troubled reader who is weary of adapting to a new variant of BNF each time another language definition appears, and without any claim for originality, I venture to submit a simple notation that has proven valuable and satisfactory in use. It has the following properties to recommend it:

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's present address: Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

1. The notation distinguishes clearly between meta-, terminal, and nonterminal symbols.
2. It does not exclude characters used as metasympols from use as symbols of the language (as e.g. "|" in BNF).
3. It contains an explicit iteration construct, and thereby avoids the heavy use of recursion for expressing simple repetition.
4. It avoids the use of an explicit symbol for the empty string (such as (empty) or ϵ).
5. It is based on the ASCII character set.

This meta language can therefore conveniently be used to define its own syntax, which may serve here as an example of its use. The word *identifier* is used to denote *nonterminal symbol*, and *literal* stands for *terminal symbol*. For brevity, *identifier* and *character* are not defined in further detail.

```
syntax      = {production}.
production = identifier "=" expression " ".
expression  = term {"|" term}.
term        = factor {factor}.
factor      = identifier | literal | "(" expression ")" |
              "[" expression "]" | "{" expression "}".
literal     = " " " " character {character} " " " " .
```

Repetition is denoted by curly brackets, i.e. {a} stands for ϵ | a | aa | aaa | Optionality is expressed by square brackets, i.e. [a] stands for ϵ | a. Parentheses merely serve for grouping, e.g. (a|b|c stands for ac | bc. Terminal symbols, i.e. literals, are enclosed in quote marks (and, if a quote mark appears as a literal itself, it is written twice), which is consistent with common practice in programming languages.

Zahl

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

Zahl

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

- eine Ziffer 2

Zahl

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

■ eine Ziffer , oder 2

■ eine Ziffer gefolgt von einer Folge von Ziffern 2 0 1 9

Zahl

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

■ eine Ziffer , oder 2

■ eine Ziffer gefolgt von einer Folge von Ziffern 2 0 1 9

`unsigned_integer = digits .`

`digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .`

`digits = digit | digit digits .`

Zahl

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

■ eine Ziffer , oder 2

■ eine Ziffer gefolgt von einer Folge von Ziffern 2 0 1 9

`unsigned_integer = digits .`

`digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .`

`digits = digit | digit digits .`

Alternative

Terminales Symbol

Nicht terminales Symbol

Zahl (nicht-rekursiv)

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

- eine Ziffer, oder 2
- eine Ziffer gefolgt von **beliebig vielen Ziffern** 2 0 1 9

`unsigned_integer = digits .`

`digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .`

`digits = digit { digit } .`

Zahl (nicht-rekursiv)

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

- eine Ziffer, oder 2
- eine Ziffer gefolgt von **beliebig vielen Ziffern** 2 0 1 9

`unsigned_integer = digits .`

`digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .`

`digits = digit { digit } .`

Optionale Repetition

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

Ausdrücke

$$- (\underline{3} - (\underline{4} - \underline{5})) * (\underline{3} + \underline{4} * \underline{5}) / \underline{6}$$

Was benötigen wir in einer Grammatik?

- Zahl

Ausdrücke

$$- \underline{(3 - (4 - 5))} * \underline{(3 + 4 * 5)} / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, (?)

$$\underline{-} (3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, (?)
- -Zahl, -(?)

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, (?)
-Zahl, -(?)
- ? * ?, ? / ?, ...

$$-(3-(4-5)) * (3+4*5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, (?)
-Zahl, -(?)
- ? * ?, ? / ?, ...
- ? - ?, ? + ?, ...

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, (?)
-Zahl, -(?)
- ? * ?, ? / ?, ...
- ? - ?, ? + ?, ...

Faktor

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, (?)
-Zahl, -(?)
- Faktor * Faktor,
Faktor / Faktor, ...
- ? - ?, ? + ?, ...

Faktor

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, (?)
-Zahl, -(?)
- Faktor * Faktor,
Faktor / Faktor, ...
- ? - ?, ? + ?, ...

Faktor

Term

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, (?)
-Zahl, -(?)
- Faktor * Faktor, Faktor
Faktor / Faktor, ...
- ? - ?, ? + ?, ...

Faktor

Term

Ausdrücke

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, (?)
-Zahl, -(?)
- Faktor * Faktor, Faktor
Faktor / Faktor, ...
- Term + Term,
Term - Term, ...

Faktor

Term

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, (?)
-Zahl, -(?)
- Faktor * Faktor, Faktor
Faktor / Faktor, ...
- Term + Term,
Term - Term, ...

Faktor

Term

Ausdruck

$$-(3-(4-5))*(3+4*5)/6$$

Was benötigen wir in einer Grammatik?

- Zahl, (?)
-Zahl, -(?)
- Faktor * Faktor, Faktor
Faktor / Faktor, ...
- Term + Term, Term
Term - Term, ...

Faktor

Term

Ausdruck

Ausdrücke

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

Was benötigen wir in einer Grammatik?

- Zahl, (Ausdruck)
-Zahl, -(Ausdruck)
- Faktor * Faktor, Faktor
Faktor / Faktor, ...
- Term + Term, Term
Term - Term, ...

Faktor

Term

Ausdruck

Die EBNF für Ausdrücke

Ein Faktor ist

- eine Zahl,

Die EBNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck

Die EBNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck

Die EBNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck oder
- ein negierter Faktor.

```
factor      = unsigned_number  
            | "(" expression ")"  
            | "-" factor.
```

Die EBNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck oder
- ein negierter Faktor.

factor = unsigned_number
| "(" expression ")"
| "-" factor .

Nicht-terminales Symbol

Terminales Symbol

Alternative

Die EBNF für Ausdrücke

factor = unsigned_number
 | "(" expression ")"
 | "-" factor .

Implikation: Ein Faktor beginnt mit

- einer Ziffer, oder
- mit „(“ , oder
- mit „-“.

Die EBNF für Ausdrücke

Ein Term ist

- Faktor,

Die EBNF für Ausdrücke

Ein Term ist

- Faktor,
- Faktor * Faktor, Faktor / Faktor,

Die EBNF für Ausdrücke

Ein Term ist

- Faktor,
- Faktor * Faktor, Faktor / Faktor,
- Faktor * Faktor * Faktor, Faktor / Faktor * Faktor, ...
- ...

Die EBNF für Ausdrücke

Ein Term ist

- Faktor,
- Faktor * Faktor, Faktor / Faktor,
- Faktor * Faktor * Faktor, Faktor / Faktor * Faktor, ...
- ...

$\text{term} = \text{factor} \{ "*" \text{ factor} \mid "/" \text{ factor} \}.$

Die EBNF für Ausdrücke

Ein Term ist

- Faktor,
- Faktor * Faktor, Faktor / Faktor,
- Faktor * Faktor * Faktor, Faktor / Faktor * Faktor, ...
- ...

term = factor { "*" factor | "/" factor } .

Optionale Repetition

Die EBNF für Ausdrücke

factor = unsigned_number
| "(" expression ")"
| "-" factor.

term = factor { "*" factor | "/" factor }.

expression = term { "+" term | "-" term }.

- **Parsen:** Feststellen, ob ein Satz nach der EBNF gültig ist.

Parsen

- **Parsen:** Feststellen, ob ein Satz nach der EBNF gültig ist.
- **Parser:** Programm zum Parsen

Parsen

- **Parsen:** Feststellen, ob ein Satz nach der EBNF gültig ist.
- **Parser:** Programm zum Parsen
- **Praktisch:** Aus der EBNF kann automatisch ein Parser generiert werden:
 - Regeln werden zu Funktionen
 - Alternativen und Optionen werden zu **if**-Anweisungen
 - Nichtterminale Symbole auf der rechten Seite werden zu Funktionsaufrufen
 - Optionale Repetitionen werden zu **while**-Anweisungen

Regeln

factor = unsigned_number
| "(" expression ")"
| "-" factor.

term = factor { "*" factor | "/" factor }.

expression = term { "+" term | "-" term }.

Ausdruck wird aus einem [Eingabestrom](#) gelesen.

```
// POST: returns true if and only if in_stream = factor ...  
//       and in this case extracts factor from in_stream  
bool factor (std::istream& in_stream);
```

```
// POST: returns true if and only if in_stream = term ...,  
//       and in this case extracts all factors from in_stream  
bool term (std::istream& in_stream);
```

```
// POST: returns true if and only if in_stream = expression ...,  
//       and in this case extracts all terms from in_stream  
bool expression (std::istream& in_stream);
```

Ausdruck wird aus einem [Eingabestrom](#) gelesen.

```
// POST: extracts a factor from in_stream
//       and returns its value
double factor (std::istream& in_stream);
```

```
// POST: extracts a term from in_stream
//       and returns its value
double term (std::istream& in_stream);
```

```
// POST: extracts an expression from in_stream
//       and returns its value
double expression (std::istream& in_stream);
```

Vorausschau von einem Zeichen...

...um jeweils die richtige Alternative zu finden.

```
// POST: the next character at the stream is returned  
//       without being consumed. returns 0 if stream ends.
```

```
char peek (std::istream& input){  
    if (input.eof()) return 0; // end of stream  
    return input.peek(); // next character in input  
}
```

```
// POST: leading whitespace characters are extracted from input  
//       and the first non-whitespace character on input returned
```

```
char lookahead (std::istream& input) {  
    input >> std::ws; // skip whitespaces  
    return peek(input);  
}
```

Zahlen parsen

```
bool isDigit(char ch){
    return ch >= '0' && ch <= '9';
}
// POST: returns an unsigned integer consumed from the stream
// number = digit {digit}.
unsigned int unsigned_number (std::istream& input){
    char ch = lookahead(input);
    assert(isDigit(ch));
    unsigned int num = 0;
    while(isDigit(ch) && input >> ch){ // read remaining digits
        num = num * 10 + ch - '0';
        ch = peek(input);
    }
    unsigned_number = digit { digit }.
    digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8'
}
return num;
}
```

Rosinenpickerei

...um jeweils nur das gewünschte Zeichen zu extrahieren.

```
// POST: if expected matches the next lookahead then consume it
//       and return true; return false otherwise
bool consume (std::istream& in_stream, char expected)
{
    if (lookahead(in_stream) == expected){
        in_stream >> expected; // consume one character
        return true;
    }
    return false;
}
```

Faktoren auswerten

```
double factor (std::istream& in_stream)
{
    double value;
    if (consume(in_stream, '(')) {
        value = expression (in_stream);
        consume(in_stream, ')');
    } else if (consume(in_stream, '-')) {
        value = -factor (in_stream);
    } else {
        value = unsigned_number(in_stream);
    }
    return value;
}
```

```
factor = "(" expression ")"
| "-" factor
| unsigned_number.
```

Terme auswerten

```
double term (std::istream& in_stream)
{
    double value = factor (in_stream);
    while(true){
        if (consume(in_stream, '*'))
            value *= factor(in_stream);
        else if (consume(in_stream, '/'))
            value /= factor(in_stream)
        else
            return value;
    }
}
```

term = factor { "*" factor | "/" factor }.

Ausdrücke auswerten

```
double expression (std::istream& in_stream)
{
    double value = term(in_stream);
    while(true){
        if (consume(in_stream, '+'))
            value += term (in_stream);
        else if (consume(in_stream, '-'))
            value -= term(in_stream)
        else
            return value;
    }
}
```

expression = term { "+" term | "-" term }.

Rekursion!

Factor

Term

Expression

Rekursion!

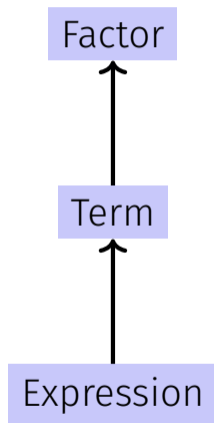
Factor

Term

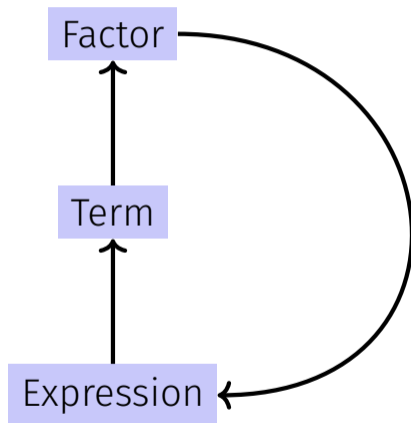
Expression

```
graph BT; Expression --> Term; Term --> Factor
```

Rekursion!



Rekursion!



EBNF — Und es funktioniert!

EBNF (calculator.cpp, Auswertung von links nach rechts):

```
factor      = unsigned_number  
            | "(" expression ")"  
            | "-" factor.  
  
term       = factor { "*" factor | "/" factor }.  
  
expression = term { "+" term | "-" term }.
```

```
std::stringstream input ("1-2-3");  
std::cout << expression (input) << "\n"; // -4
```

18. Structs

Rationale Zahlen, Struct-Definition

Rechnen mit rationalen Zahlen

- Rationale Zahlen (\mathbb{Q}) sind von der Form $\frac{n}{d}$ mit n und d in \mathbb{Z}
- C++ hat keinen „eingebauten“ Typ für rationale Zahlen

Rechnen mit rationalen Zahlen

- Rationale Zahlen (\mathbb{Q}) sind von der Form $\frac{n}{d}$ mit n und d in \mathbb{Z}
- C++ hat keinen „eingebauten“ Typ für rationale Zahlen

Ziel

Wir bauen uns selbst einen C++-Typ für rationale Zahlen!



Vision

```
// input
std::cout << "Rational number r =? ";
rational r;
std::cin >> r;
std::cout << "Rational number s =? ";
rational s;
std::cin >> s;

// computation and output
std::cout << "Sum is " << r + s << ".\n";
```

Ein erstes Struct

```
struct rational {  
    int n;  
    int d; // INV: d != 0  
};
```

Ein erstes Struct

```
struct rational {  
    int n;  
    int d; // INV: d != 0  
};
```

Member-Variable (**n**umerator)

Member-Variable (**d**enominator)

Ein erstes Struct

```
struct rational {  
    int n; ← Member-Variable  
    int d; ← // INV: d != 0  
};  
           ← Member-Variable
```

- **struct** definiert einen neuen **Typ**

Ein erstes Struct

```
struct rational {  
    int n; ← Member-Variable  
    int d; ← // INV: d != 0  
};  
           ← Member-Variable
```

- **struct** definiert einen neuen **Typ**
- Formaler Wertebereich: *kartesisches Produkt* der Wertebereiche existierender Typen

Ein erstes Struct

```
struct rational {  
    int n; ← Member-Variable  
    int d; ← // INV: d != 0  
};  
                ← Member-Variable
```

- **struct** definiert einen neuen **Typ**
- Formaler Wertebereich: *kartesisches Produkt* der Wertebereiche existierender Typen
- Echter Wertebereich: **rational** \subsetneq **int** \times **int**.

Zugriff auf Member-Variablen

```
struct rational {  
    int n;  
    int d; // INV: d != 0  
};
```

```
rational add (rational a, rational b){  
    rational result;  
    result.n = a.n * b.d + a.d * b.n;  
    result.d = a.d * b.d;  
    return result;  
}
```

$$\frac{r_n}{r_d} := \frac{a_n}{a_d} + \frac{b_n}{b_d} = \frac{a_n \cdot b_d + a_d \cdot b_n}{a_d \cdot b_d}$$

Eingabe

```
// Input r
rational r;
std::cout << "Rational number r:\n";
std::cout << " numerator =? ";
std::cin >> r.n;
std::cout << " denominator =? ";
std::cin >> r.d;

// Input s the same way
rational s;
...
```

Vision in Reichweite ...

```
// computation
const rational t = add (r, s);

// output
std::cout << "Sum is " << t.n << "/" << t.d << ".\n";
```

Struct-Definitionen: Beispiele

```
struct rational_vector_3 {  
    rational x;  
    rational y;  
    rational z;  
};
```

Zugrundeliegende Typen können fundamentale aber auch *benutzerdefinierte* Typen sein.

Struct-Definitionen: Beispiele

```
struct extended_int {  
    // represents value if is_positive==true  
    // and -value otherwise  
    unsigned int value;  
    bool is_positive;  
};
```

Die zugrundeliegenden Typen können natürlich auch *verschieden* sein.

Structs: Initialisierung und Zuweisung

```
rational s; ← Member-Variablen uninitialisiert  
(wird sich bald ändern)
```

```
rational t = {1,5};
```

```
rational u = t;
```

```
t = u;
```

```
rational v = add (u,t);
```

Structs: Initialisierung und Zuweisung

```
rational s;
```

```
rational t = {1,5}; ← Memberweise Initialisierung:  
t.n = 1, t.d = 5
```

```
rational u = t;
```

```
t = u;
```

```
rational v = add (u,t);
```

Structs: Initialisierung und Zuweisung

```
rational s;
```

```
rational t = {1,5};
```

```
rational u = t; ← Memberweise Kopie
```

```
t = u;
```

```
rational v = add (u,t);
```

Structs: Initialisierung und Zuweisung

```
rational s;
```

```
rational t = {1,5};
```

```
rational u = t;
```

```
t = u; ← Memberweise Kopie
```

```
rational v = add (u,t);
```


Structs: Initialisierung und Zuweisung

```
rational s;
```

```
rational t = {1,5};
```

```
rational u = t;
```

```
t = u;
```

```
rational v = add (u,t); ← Memberweise Kopie
```

Structs vergleichen?

Für jeden fundamentalen Typ (`int`, `double`, ...) gibt es die Vergleichsoperatoren `==` und `!=`, aber nicht für Structs!
Warum?

Structs vergleichen?

Für jeden fundamentalen Typ (`int`, `double`, ...) gibt es die Vergleichsoperatoren `==` und `!=`, aber nicht für Structs!
Warum?

- Memberweiser Vergleich ergibt im allgemeinen keinen Sinn,...

Structs vergleichen?

Für jeden fundamentalen Typ (`int`, `double`, ...) gibt es die Vergleichsoperatoren `==` und `!=`, aber nicht für Structs!
Warum?

- Memberweiser Vergleich ergibt im allgemeinen keinen Sinn,...
- ...denn dann wäre z.B. $\frac{2}{3} \neq \frac{4}{6}$

Benutzerdefinierte Operatoren

Statt

```
rational t = add(r, s);
```

würden wir lieber

```
rational t = r + s;
```

schreiben.

Benutzerdefinierte Operatoren

Statt

```
rational t = add(r, s);
```

würden wir lieber

```
rational t = r + s;
```

schreiben.

Das geht mit *Überladen von Operatoren* (\rightarrow nächste Woche).

19. Klassen

Funktions- und Operatorüberladung, Datenkapselung,
Klassen, Memberfunktionen, Konstruktoren

Funktionsüberladung

- Eine Funktion ist bestimmt durch Namen, Typen, Anzahl und Reihenfolge der Argumente

```
double sq (double x) { ... } // f1
int sq (int x) { ... } // f2
int pow (int b, int e) { ... } // f3
int pow (int e) { return pow (2,e); } // f4
```


Funktionsüberladung

- Eine Funktion ist bestimmt durch Namen, Typen, Anzahl und Reihenfolge der Argumente

```
double sq (double x) { ... } // f1
int sq (int x) { ... } // f2
int pow (int b, int e) { ... } // f3
int pow (int e) { return pow (2,e); } // f4
```

- Der Compiler wählt bei einem Funktionsaufruf automatisch die Funktion, welche „am besten passt“

Funktionsüberladung

- Eine Funktion ist bestimmt durch Namen, Typen, Anzahl und Reihenfolge der Argumente

```
double sq (double x) { ... } // f1
int sq (int x) { ... } // f2
int pow (int b, int e) { ... } // f3
int pow (int e) { return pow (2,e); } // f4
```

- Der Compiler wählt bei einem Funktionsaufruf automatisch die Funktion, welche „am besten passt“

```
std::cout << sq (3);
```

Funktionsüberladung

- Eine Funktion ist bestimmt durch Namen, Typen, Anzahl und Reihenfolge der Argumente

```
double sq (double x) { ... } // f1
int sq (int x) { ... } // f2
int pow (int b, int e) { ... } // f3
int pow (int e) { return pow (2,e); } // f4
```

- Der Compiler wählt bei einem Funktionsaufruf automatisch die Funktion, welche „am besten passt“

```
std::cout << sq (3); // Compiler wählt f2
std::cout << sq (1.414);
```

Funktionsüberladung

- Eine Funktion ist bestimmt durch Namen, Typen, Anzahl und Reihenfolge der Argumente

```
double sq (double x) { ... } // f1
int sq (int x) { ... } // f2
int pow (int b, int e) { ... } // f3
int pow (int e) { return pow (2,e); } // f4
```

- Der Compiler wählt bei einem Funktionsaufruf automatisch die Funktion, welche „am besten passt“

```
std::cout << sq (3); // Compiler wählt f2
std::cout << sq (1.414); // Compiler wählt f1
std::cout << pow (2);
```

Funktionsüberladung

- Eine Funktion ist bestimmt durch Namen, Typen, Anzahl und Reihenfolge der Argumente

```
double sq (double x) { ... } // f1
int sq (int x) { ... } // f2
int pow (int b, int e) { ... } // f3
int pow (int e) { return pow (2,e); } // f4
```

- Der Compiler wählt bei einem Funktionsaufruf automatisch die Funktion, welche „am besten passt“

```
std::cout << sq (3); // Compiler wählt f2
std::cout << sq (1.414); // Compiler wählt f1
std::cout << pow (2); // Compiler wählt f4
std::cout << pow (3,3);
```

Funktionsüberladung

- Eine Funktion ist bestimmt durch Namen, Typen, Anzahl und Reihenfolge der Argumente

```
double sq (double x) { ... } // f1
int sq (int x) { ... } // f2
int pow (int b, int e) { ... } // f3
int pow (int e) { return pow (2,e); } // f4
```

- Der Compiler wählt bei einem Funktionsaufruf automatisch die Funktion, welche „am besten passt“

```
std::cout << sq (3); // Compiler wählt f2
std::cout << sq (1.414); // Compiler wählt f1
std::cout << pow (2); // Compiler wählt f4
std::cout << pow (3,3); // Compiler wählt f3
```

Operator-Überladung

- Operatoren sind spezielle Funktionen und können auch überladen werden
- Name des Operators *op*:

```
operatorop
```

rational addieren, bisher

```
// POST: return value is the sum of a and b
rational add (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
const rational t = add (r, s);
```


rational addieren, neu

```
// POST: return value is the sum of a and b
rational operator+ (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
const rational t = r + s;
```

rational addieren, neu

```
// POST: return value is the sum of a and b
rational operator+ (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
const rational t = r + s;
```

↑
Infix-Notation

rational addieren, neu

```
// POST: return value is the sum of a and b
rational operator+ (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
const rational t = operator+ (r, s);
```



Äquivalent, aber unpraktisch: funktionale Notation

Unäres Minus

Nur ein Argument:

```
// POST: return value is -a
rational operator- (rational a)
{
    a.n = -a.n;
    return a;
}
```

Vergleichsoperatoren

können auch definiert werden, so dass sie das „Richtige“ machen:

Vergleichsoperatoren

können auch definiert werden, so dass sie das „Richtige“ machen:

```
// POST: returns true iff a == b
bool operator==(rational a, rational b)
{
    return a.n * b.d == a.d * b.n;
}
```

Vergleichsoperatoren

können auch definiert werden, so dass sie das „Richtige“ machen:

```
// POST: returns true iff a == b
bool operator==(rational a, rational b)
{
    return a.n * b.d == a.d * b.n;
}
```

$$\frac{2}{3} = \frac{4}{6} \quad \checkmark$$

Arithmetische Zuweisungen

Wir wollen z.B. schreiben

```
rational r;  
r.n = 1; r.d = 2;           // 1/2
```

```
rational s;  
s.n = 1; s.d = 3;         // 1/3
```

```
r += s;  
std::cout << r.n << "/" << r.d; // 5/6
```


Operator +=

```
rational& operator+= (rational& a, rational b)
{
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

Operator +=

```
rational& operator+= (rational& a, rational b)
{
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

- Der L-Wert **a** wird um den Wert von **b** erhöht und als L-Wert zurückgegeben.

Ein-/Ausgabeoperatoren

können auch überladen werden.

■ Bisher:

```
std::cout << "Sum is " << t.n << "/" << t.d << "\n";
```

■ Neu (gewünscht):

```
std::cout << "Sum is " << t << "\n";
```

Ein-/Ausgabeoperatoren

können auch überladen werden wie folgt:

```
// POST: r has been written to out
std::ostream& operator<< (std::ostream& out, rational r)
{
    return out << r.n << "/" << r.d;
}
```

Ein-/Ausgabeoperatoren

können auch überladen werden wie folgt:

```
// POST: r has been written to out
std::ostream& operator<< (std::ostream& out, rational r)
{
    return out << r.n << "/" << r.d;
}
```

schreibt `r` auf den Ausgabestrom
und gibt diesen als L-Wert zurück

Eingabe

```
// PRE: in starts with a rational number of the form "n/d"  
// POST: r has been read from in  
std::istream& operator>> (std::istream& in, rational& r){  
    char c; // separating character '/'  
    return in >> r.n >> c >> r.d;  
}
```

liest **r** aus dem Eingabestrom
und gibt diesen als L-Wert zurück.

Ziel erreicht!

```
// input
std::cout << "Rational number r =? ";
rational r;
std::cin >> r;

std::cout << "Rational number s =? ";
rational s;
std::cin >> s;

// computation and output
std::cout << "Sum is " << r + s << ".\n";
```

Ziel erreicht!

```
// input
std::cout << "Rational number r =? ";
rational r;
std::cin >> r;

std::cout << "Rational number s =? ";
rational s;
std::cin >> s;

// computation and output
std::cout << "Sum is " << r + s << ".\n";
```

operator >>

operator +

operator<<

Ein neuer Typ mit Funktionalität...

```
struct rational {
    int n;
    int d; // INV: d != 0
};

// POST: return value is the sum of a and b
rational operator+ (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
```

...gehört in eine Bibliothek!

rational.h

- Definition des Structs `rational`
- Funktionsdeklarationen

rational.cpp

- Arithmetische Operatoren (`operator+`, `operator+=`, ...)
- Relationale Operatoren (`operator==`, `operator>`, ...)
- Ein-/Ausgabe (`operator >>`, `operator <<`, ...)

Gedankenexperiment

Die drei Kernaufgaben der ETH:

- Forschung

Gedankenexperiment

Die drei Kernaufgaben der ETH:

- Forschung
- Lehre

Gedankenexperiment

Die drei Kernaufgaben der ETH:

- Forschung
- Lehre
- Technologietransfer

Gedankenexperiment

Die drei Kernaufgaben der ETH:

- Forschung
- Lehre
- Technologietransfer

Wir gründen die Startup-Firma RAT PACK®!

Gedankenexperiment

Die drei Kernaufgaben der ETH:

- Forschung
- Lehre
- Technologietransfer

Wir gründen die Startup-Firma RAT PACK®!

- Verkauf der `rational`-Bibliothek an Kunden
- Weiterentwicklung nach Kundenwünschen

Der Kunde ist zufrieden

“Buying RAT PACK[®] has been a game-changing move to put us on the forefront of cutting-edge technology in social media engineering.”

B. Labla, CEO

Der Kunde ist zufrieden

...und programmiert fleissig mit `rational`.

Der Kunde ist zufrieden

...und programmiert fleissig mit `rational`.

■ Ausgabe als `double`-Wert ($\frac{3}{5} \rightarrow 0.6$)

Der Kunde ist zufrieden

...und programmiert fleissig mit `rational`.

■ Ausgabe als `double`-Wert ($\frac{3}{5} \rightarrow 0.6$)

```
// POST: double approximation of r
double to_double (rational r)
{
    double result = r.n;
    return result / r.d;
}
```

Der Kunde will mehr

“Können wir rationale Zahlen mit erweitertem Wertebereich bekommen?”

Der Kunde will mehr

“Können wir rationale Zahlen mit erweitertem Wertebereich bekommen?”

■ Klar, kein Problem, z.B.:

```
struct rational {  
    int n;  
    int d;  
};
```



```
struct rational {  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

Neue Version von RAT PACK®



Nichts geht mehr!



Neue Version von RAT PACK®



Nichts geht mehr!

- Was ist denn das Problem?



Neue Version von RAT PACK®



Nichts geht mehr!

■ Was ist denn das Problem?



$-\frac{3}{5}$ ist jetzt manchmal 0.6, das kann doch nicht sein!



Neue Version von RAT PACK®



Nichts geht mehr!

■ Was ist denn das Problem?



$-\frac{3}{5}$ ist jetzt manchmal 0.6, das kann doch nicht sein!

■ Daran ist wohl Ihre Konversion nach `double` schuld, denn unsere Bibliothek ist korrekt.



Neue Version von RAT PACK®



Nichts geht mehr!

- Was ist denn das Problem?



$-\frac{3}{5}$ ist jetzt manchmal 0.6, das kann doch nicht sein!

- Daran ist wohl Ihre Konversion nach `double` schuld, denn unsere Bibliothek ist korrekt.



Bisher funktionierte es aber, also ist die neue Version schuld!



Schuldanalyse

```
// POST: double approximation of r
double to_double (rational r){
    double result = r.n;
    return result / r.d;
}
```

Schuldanalyse

```
// POST: double approximation of r
double to_double (rational r){
    double result = r.n;
    return result / r.d;
}
```

Korrekt mit...

```
struct rational {
    int n;
    int d;
};
```

Schuldanalyse

```
// POST: double approximation of r
double to_double (rational r){
    double result = r.n;
    return result / r.d;
}
```

Korrekt mit...

```
struct rational {
    int n;
    int d;
};
```

...aber nicht mit

```
struct rational {
    unsigned int n;
    unsigned int d;
    bool is_positive;
};
```

Schuldanalyse

```
// POST: double approximation of r
double to_double (rational r){
    double result = r.n;
    return result / r.d;
}
```

r.is_positive und result.is_positive
kommen nicht vor.

Korrekt mit...

```
struct rational {
    int n;
    int d;
};
```

...aber nicht mit

```
struct rational {
    unsigned int n;
    unsigned int d;
    bool is_positive;
};
```

Wir sind schuld!

- Kunde sieht und benutzt unsere **Repräsentation** rationaler Zahlen (zu Beginn `r.n`, `r.d`)

Wir sind schuld!

- Kunde sieht und benutzt unsere **Repräsentation** rationaler Zahlen (zu Beginn `r.n`, `r.d`)
- Ändern wir sie (`r.n`, `r.d`, `r.is_positive`), funktionieren Kunden-Programme nicht mehr.

Wir sind schuld!

- Kunde sieht und benutzt unsere **Repräsentation** rationaler Zahlen (zu Beginn `r.n`, `r.d`)
- Ändern wir sie (`r.n`, `r.d`, `r.is_positive`), funktionieren Kunden-Programme nicht mehr.
- Kein Kunde ist bereit, bei jeder neuen Version der Bibliothek seine Programme anzupassen.

Wir sind schuld!

- Kunde sieht und benutzt unsere **Repräsentation** rationaler Zahlen (zu Beginn `r.n`, `r.d`)
- Ändern wir sie (`r.n`, `r.d`, `r.is_positive`), funktionieren Kunden-Programme nicht mehr.
- Kein Kunde ist bereit, bei jeder neuen Version der Bibliothek seine Programme anzupassen.

⇒ RAT PACK[®] ist Geschichte...

Idee der Datenkapselung (Information Hiding)

- Ein Typ ist durch *Wertebereich* und *Funktionalität* eindeutig definiert.

Idee der Datenkapselung (Information Hiding)

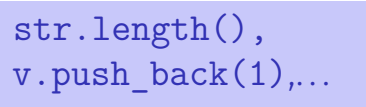
- Ein Typ ist durch *Wertebereich* und *Funktionalität* eindeutig definiert.
- Die *Repräsentation* soll *nicht sichtbar* sein.

Idee der Datenkapselung (Information Hiding)

- Ein Typ ist durch *Wertebereich* und *Funktionalität* eindeutig definiert.
- Die *Repräsentation* soll *nicht sichtbar* sein.
- \Rightarrow Dem Kunden wird keine *Repräsentation*, sondern **Funktionalität** angeboten.

Idee der Datenkapselung (Information Hiding)

- Ein Typ ist durch *Wertebereich* und *Funktionalität* eindeutig definiert.
- Die *Repräsentation* soll *nicht sichtbar* sein.
- \Rightarrow Dem Kunden wird keine *Repräsentation*, sondern **Funktionalität** angeboten.



```
str.length(),  
v.push_back(1),...
```

- sind das Konzept zur **Datenkapselung** in C++

Klassen

- sind das Konzept zur **Datenkapselung** in C++
- sind eine Variante von Structs

Klassen

- sind das Konzept zur **Datenkapselung** in C++
- sind eine Variante von Structs
- gibt es in vielen *objektorientierten Programmiersprachen*

Datenkapselung: `public` / `private`

```
class rational {  
    int n;  
    int d; // INV: d != 0  
};
```

Wird statt `struct` verwendet, wenn überhaupt etwas "versteckt" werden soll.

Datenkapselung: `public` / `private`

```
class rational {  
    int n;  
    int d; // INV: d != 0  
};
```

Wird statt `struct` verwendet, wenn überhaupt etwas "versteckt" werden soll.

Einziger Unterschied:

- `struct`: standardmässig wird **nichts** versteckt
- `class`: standardmässig wird **alles** versteckt

Datenkapselung: public / private

```
class rational {  
    int n;  
    int d; // INV: d != 0  
};
```

Anwendungscode:

```
rational r;  
r.n = 1;      // error: n is private  
r.d = 2;      // error: d is private  
int i = r.n;  // error: n is private
```

Datenkapselung: public / private

```
class rational {  
    int n;  
    int d; // INV: d != 0  
};
```

Gute Nachricht: `r.d = 0` aus Versehen geht nicht mehr

Anwendungscode:

```
rational r;  
r.n = 1;      // error: n is private  
r.d = 2;      // error: d is private  
int i = r.n;  // error: n is private
```

Datenkapselung: public / private

```
class rational {  
    int n;  
    int d; // INV: d != 0  
};
```

Gute Nachricht: `r.d = 0` aus Versehen geht nicht mehr

Schlechte Nachricht: Der Kunde kann nun gar nichts mehr machen ...

Anwendungscode:

```
rational r;  
r.n = 1;      // error: n is private  
r.d = 2;      // error: d is private  
int i = r.n;  // error: n is private
```

Datenkapselung: `public` / `private`

```
class rational {  
    int n;  
    int d; // INV: d != 0  
};
```

Gute Nachricht: `r.d = 0` aus Versehen geht nicht mehr

Schlechte Nachricht: Der Kunde kann nun gar nichts mehr machen ...

Anwendungscode:

```
rational r;  
r.n = 1; // error: n is private  
r.d = 2; // error: d is private  
int i = r.n; // error: n is private
```

...und wir auch nicht
(kein `operator+`,...)

Memberfunktionen: Deklaration

```
class rational {
public:
    // POST: return value is the numerator of this instance
    int numerator () const {
        return n;
    }
    // POST: return value is the denominator of this instance
    int denominator () const {
        return d;
    }
private:
    int n;
    int d; // INV: d!= 0
};
```


Memberfunktionen: Deklaration

```
class rational {  
public:  
    // POST: return value is the numerator of this instance  
    int numerator () const {  
        return n;  
    }  
    // POST: return value is the denominator of this instance  
    int denominator () const {  
        return d;  
    }  
private:  
    int n;  
    int d; // INV: d!= 0  
};
```

öffentlicher Bereich

Memberfunktionen: Deklaration

```
class rational {  
public:  
    // POST: return value is the numerator of this instance  
    int numerator () const {  
        return n;  
    }  
    // POST: return value is the denominator of this instance  
    int denominator () const {  
        return d;  
    }  
private:  
    int n;  
    int d; // INV: d!= 0  
};
```

öffentlicher Bereich

Memberfunktion

The diagram consists of a blue bracket on the left side of the code, spanning the two public member function declarations. A blue arrow points from the text 'Memberfunktion' (highlighted in a light blue box) to the opening curly brace of the 'numerator' function signature.

Memberfunktionen: Deklaration

```
class rational {  
public:  
    // POST: return value is the numerator of this instance  
    int numerator () const {  
        return n;  
    }  
    // POST: return value is the denominator of this instance  
    int denominator () const {  
        return d;  
    }  
private:  
    int n;  
    int d; // INV: d!= 0  
};
```

öffentlicher Bereich

Memberfunktion

Memberfunktionen haben Zugriff auf private Daten

Memberfunktionen: Aufruf

```
// Definition des Typs
class rational {
    ...
};
...
// Variable des Typs
rational r; Member-Zugriff

int n = r.numerator(); // Zaehler
int d = r.denominator(); // Nenner
```

Memberfunktionen: Definition

```
// POST: returns numerator of this instance
int numerator () const
{
    return n;
}
```

Memberfunktionen: Definition ???

```
// POST: returns numerator of this instance
int numerator () const
{
    return n;
}
```

Memberfunktionen: Definition

```
// POST: returns numerator of this instance
int numerator () const
{
    return n;                r.numerator()
}
```

- Eine Memberfunktion wird *für* einen Ausdruck der Klasse aufgerufen.

Memberfunktionen: Definition

```
// POST: returns numerator of this instance
int numerator () const
{
    return n;
}
```

r.numerator()

- Eine Memberfunktion wird *für* einen Ausdruck der Klasse aufgerufen. In der Funktion: **this** ist der Name dieses *impliziten Arguments*.

Memberfunktionen: Definition

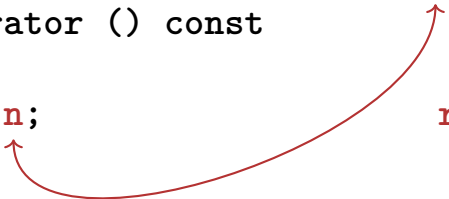
```
// POST: returns numerator of this instance
int numerator () const
{
    return n;                r.numerator()
}
```

- Eine Memberfunktion wird *für* einen Ausdruck der Klasse aufgerufen. In der Funktion: **this** ist der Name dieses *impliziten Arguments*.
- Das *const* bezieht sich auf die Instanz **this**

Memberfunktionen: Definition

```
// POST: returns numerator of this instance
int numerator () const
{
    return n;
}
```

r.numerator()



- Eine Memberfunktion wird *für* einen Ausdruck der Klasse aufgerufen. In der Funktion: **this** ist der Name dieses *impliziten Arguments*.
- Das *const* bezieht sich auf die Instanz **this**
- **n** ist Abkürzung für **this->n** (genaue Erklärung von „->“ nächste Woche)

const und Memberfunktionen

```
class rational {  
public:  
    int numerator () const  
    { return n; }  
    void set_numerator (int N)  
    { n = N;}  
    ...  
}
```

```
rational x;  
x.set_numerator(10); // ok;  
const rational y = x;  
int n = y.numerator(); // ok;  
y.set_numerator(10); // error;
```

Das **const** an einer Memberfunktion liefert das Versprechen, dass eine Instanz nicht über diese Funktion verändert wird. **const** Objekte dürfen nur **const** Memberfunktionen aufrufen!

This rational vs. dieser Bruch

```
class rational {
    int n;
    ...
public:
    int numerator () const
    {
        return n;
    }
};

rational r;
...
std::cout << r.numerator();
```

This rational vs. dieser Bruch

```
class rational {
    int n;
    ...
public:
    int numerator () const
    {
        return this->n;
    }
};

rational r;
...
std::cout << r.numerator();
```

This rational vs. dieser Bruch

So wäre es **in etwa** ...

```
class rational {
    int n;
    ...
public:
    int numerator () const
    {
        return this->n;
    }
};

rational r;
...
std::cout << r.numerator();
```

This rational vs. dieser Bruch

So wäre es **in etwa** ...

```
class rational {
    int n;
    ...
public:
    int numerator () const
    {
        return this->n;
    }
};

rational r;
...
std::cout << r.numerator();
```

... ohne Memberfunktionen

```
struct bruch {
    int n;
    ...
};

int numerator (const bruch& dieser)
{
    return dieser.n;
}

bruch r;
..
std::cout << numerator(r);
```

Member-Definition: In-Class

```
class rational {  
    int n;  
    ...  
public:  
    int numerator () const  
    {  
        return n;  
    }  
    ....  
};
```

- Keine Trennung zwischen Deklaration und Definition (schlecht für Bibliotheken)

Member-Definition: In-Class vs. Out-of-Class

```
class rational {  
    int n;  
    ...  
public:  
    int numerator () const  
    {  
        return n;  
    }  
    ....  
};
```

- Keine Trennung zwischen Deklaration und Definition (schlecht für Bibliotheken)

```
class rational {  
    int n;  
    ...  
public:  
    int numerator () const;  
    ...  
};  
  
int rational::numerator () const  
{  
    return n;  
}
```

- So geht's auch.

Initialisierung? Konstruktoren!

```
class rational
{
public:
    rational (int num, int den)
        : n (num), d (den)
    {
        assert (den != 0);
    }
    ...
};
...
rational r (2,3); // r = 2/3
```

Initialisierung? Konstruktoren!

```
class rational
{
public:
    rational (int num, int den)
        : n (num), d (den) ← Initialisierung der
                               Membervariablen
    {
        assert (den != 0); ← Funktionsrumpf.
    }
    ...
};
...
rational r (2,3); // r = 2/3
```

Initialisierung "rational = int"?

```
class rational
{
public:
    rational (int num)
        : n (num), d (1)
    {}
    ...
};
...
rational r (2); // Explizite Initialisierung mit 2
rational s = 2; // Implizite Konversion
```

Initialisierung “rational = int”?

```
class rational
{
public:
    rational (int num)
        : n (num), d (1)
    {} ← Leerer Funktionsrumpf
    ...
};
...
rational r (2); // Explizite Initialisierung mit 2
rational s = 2; // Implizite Konversion
```

Der Default-Konstruktor

```
class rational
{
public:
    ...
    rational () ← Leere Argumentliste
        : n (0), d (1)
    {}
    ...
};
...
rational r; // r = 0
```

Der Default-Konstruktor

```
class rational
{
public:
    ...
    rational () ← Leere Argumentliste
        : n (0), d (1)
    {}
    ...
};
...
rational r; // r = 0
```

⇒ Es gibt keine uninitialisierten Variablen vom Typ rational mehr!

Alternative: Default-Konstruktor löschen

```
class rational
{
public:
    ...
    rational () = delete;
    ...
};
...
rational r; // error: use of deleted function 'rational::rational'
```

⇒ Es gibt keine uninitialisierten Variablen vom Typ rational mehr!

Benutzerdefinierte Konversionen

sind definiert durch Konstruktoren mit genau *einem* Argument

```
rational (int num)
    : n (num), d (1)
    {}
```

```
rational r = 2; // implizite Konversion
```

Benutzerdefinierte Konversionen

sind definiert durch Konstruktoren mit genau *einem* Argument

```
rational (int num) ←  
    : n (num), d (1)  
    {}
```

Benutzerdefinierte Konversion von **int** nach **rational**. Damit wird **int** zu einem Typ, dessen Werte nach **rational** konvertierbar sind.

```
rational r = 2; // implizite Konversion
```

RAT PACK[®] Reloaded ...

Kundenprogramm sieht nun so aus:

```
// POST: double approximation of r
double to_double (const rational r)
{
    double result = r.numerator();
    return result / r.denominator();
}
```

RAT PACK[®] Reloaded ...

Kundenprogramm sieht nun so aus:

```
// POST: double approximation of r
double to_double (const rational r)
{
    double result = r.numerator();
    return result / r.denominator();
}
```

- Wir können die Memberfunktionen zusammen mit der Repräsentation anpassen. ✓

RAT PACK[®] Reloaded ...

vorher

```
class rational {  
    ...  
private:  
    int n;  
    int d;  
};
```

RAT PACK[®] Reloaded ...

vorher

```
class rational {  
    ...  
private:  
    int n;  
    int d;  
};  
  
int numerator () const  
{  
    return n;  
}
```

RAT PACK[®] Reloaded ...

vorher

```
class rational {  
    ...  
private:  
    int n;  
    int d;  
};  
  
int numerator () const  
{  
    return n;  
}
```

nachher

```
class rational {  
    ...  
private:  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

RAT PACK[®] Reloaded ...

vorher

```
class rational {  
    ...  
private:  
    int n;  
    int d;  
};
```

```
int numerator () const  
{  
    return n;  
}
```

nachher

```
class rational {  
    ...  
private:  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

```
int numerator () const {  
    if (is_positive)  
        return n;  
    else {  
        int result = n;  
        return -result;  
    }  
}
```


RAT PACK[®] Reloaded ?

```
class rational {  
    ...  
private:  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

```
int numerator () const  
{  
    if (is_positive)  
        return n;  
    else {  
        int result = n;  
        return -result;  
    }  
}
```

RAT PACK[®] Reloaded ?

```
class rational {  
    ...  
private:  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

```
int numerator () const  
{  
    if (is_positive)  
        return n;  
    else {  
        int result = n;  
        return -result;  
    }  
}
```

- Wertebereich von Zähler und Nenner wieder wie vorher

RAT PACK[®] Reloaded ?

```
class rational {  
    ...  
private:  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

```
int numerator () const  
{  
    if (is_positive)  
        return n;  
    else {  
        int result = n;  
        return -result;  
    }  
}
```

- Wertebereich von Zähler und Nenner wieder wie vorher
- Dazu noch möglicher Überlauf

Datenkapselung noch unvollständig

Die Sicht des Kunden (`rational.h`):

```
class rational {  
public:  
    // POST: returns numerator of *this  
    int numerator () const;  
    ...  
private:  
    // none of my business  
};
```

Datenkapselung noch unvollständig

Die Sicht des Kunden (`rational.h`):

```
class rational {
public:
    // POST: returns numerator of *this
    int numerator () const;
    ...
private:
    // none of my business
};
```

- Wir legen uns auf Zähler-/Nennertyp `int` fest.

Datenkapselung noch unvollständig

Die Sicht des Kunden (`rational.h`):

```
class rational {
public:
    // POST: returns numerator of *this
    int numerator () const;
    ...
private:
    // none of my business
};
```

- Wir legen uns auf Zähler-/Nennertyp `int` fest.
- Lösung: Nicht nur Daten, auch `Typen` kapseln.

Fix: „Unser“ Typ `rational::integer`

Die Sicht des Kunden (`rational.h`):

```
public:  
    using integer = long int; // might change  
    // POST: returns numerator of *this  
    integer numerator () const;
```

Fix: „Unser“ Typ `rational::integer`

Die Sicht des Kunden (`rational.h`):

```
public:  
    using integer = long int; // might change  
    // POST: returns numerator of *this  
    integer numerator () const;
```

- Wir stellen einen eigenen Typ zur Verfügung!

Fix: „Unser“ Typ `rational::integer`

Die Sicht des Kunden (`rational.h`):

```
public:  
    using integer = long int; // might change  
    // POST: returns numerator of *this  
    integer numerator () const;
```

- Wir stellen einen eigenen Typ zur Verfügung!
- Festlegung nur auf **Funktionalität**, z.B.:
 - implizite Konversion `int` \rightarrow `rational::integer`

Fix: „Unser“ Typ `rational::integer`

Die Sicht des Kunden (`rational.h`):

```
public:  
    using integer = long int; // might change  
    // POST: returns numerator of *this  
    integer numerator () const;
```

- Wir stellen einen eigenen Typ zur Verfügung!
- Festlegung nur auf **Funktionalität**, z.B.:
 - implizite Konversion `int` \rightarrow `rational::integer`
 - Funktion `double to_double (rational::integer)`

RAT PACK[®] Revolutions

Endlich ein Kundenprogramm, das stabil bleibt:

```
// POST: double approximation of r
double to_double (const rational r)
{
    rational::integer n = r.numerator();
    rational::integer d = r.denominator();
    return to_double (n) / to_double (d);
}
```

20. Dynamische Datenstrukturen I

Dynamischer Speicher, Adressen und Zeiger, Const-Zeiger, Arrays, Array-basierte Vektoren

Wiederholung: `vector<T>`

- Kann mit beliebiger Grösse `n` initialisiert werden

Wiederholung: `vector<T>`

- Kann mit beliebiger Grösse **n** initialisiert werden
- Unterstützt diverse Operationen:

```
e = v[i];           // Get element
v[i] = e;          // Set element
l = v.size();      // Get size
v.push_front(e);   // Prepend element
v.push_back(e);    // Append element
...
```

Wiederholung: `vector<T>`

- Kann mit beliebiger Grösse `n` initialisiert werden
- Unterstützt diverse Operationen:

```
e = v[i];           // Get element
v[i] = e;          // Set element
l = v.size();      // Get size
v.push_front(e);  // Prepend element
v.push_back(e);   // Append element
...
```

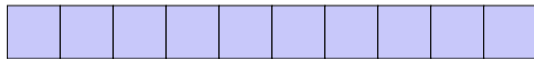
- Ein Vektor ist eine *dynamische Datenstruktur*, deren Grösse sich zur Laufzeit ändern kann

Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<T>` (später, nur kurz angeschnitten)

Vektoren im Speicher

Bereits bekannt: Ein Vektor belegt einen *zusammenhängenden* Speicherbereich



Frage: Wie *alloziert* (reserviert) man einen Speicherblock *beliebiger* Grösse zur Laufzeit, also *dynamisch*?

Der `new`-Ausdruck für Arrays

zugrundeliegender Typ

`new`
`T[expr]`

new-O Typ `int`, Wert `n`

Der `new`-Ausdruck für Arrays

zugrundeliegender Typ

`new`
`T[expr]`

new-O Typ `int`, Wert n

- **Effekt:** Neuer zusammenhängender Bereich im Speicher für n Elemente vom Typ T wird alloziert



Der `new`-Ausdruck für Arrays

zugrundeliegender Typ

`new`
`T[expr]`

new-O Typ `int`, Wert n

- **Effekt:** Neuer zusammenhängender Bereich im Speicher für n Elemente vom Typ T wird alloziert

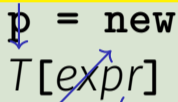


- Dieser Speicherbereich wird *Array* (der Länge n) genannt

Der `new`-Ausdruck für Arrays

zugrundeliegender Typ

```
p = new  
T[expr]
```

A diagram illustrating the mapping of the `new` expression to its components. A blue arrow points from the `new` keyword to the type `T`. Another blue arrow points from the expression `expr` to the value `n` in the explanatory text below.

new-O Typ `int`, Wert `n`

Der `new`-Ausdruck für Arrays

zugrundeliegender Typ

```
p = new  
T[expr]
```

new-O Typ `int`, Wert `n`

- **Wert:** Die Startadresse des Speicherbereichs



- **Typ:** Ein Zeiger T^* (mehr dazu gleich)

Ausblick: `new` und `delete`

```
new  
T[expr]
```

- Bisher: Speicher (lokale Variablen, Funktionsargumente) „lebt“ nur innerhalb eines Funktionsaufrufs

Ausblick: `new` und `delete`

```
new  
T[expr]
```

- Bisher: Speicher (lokale Variablen, Funktionsargumente) „lebt“ nur innerhalb eines Funktionsaufrufs
- Aber jetzt: Speicherblock im Vektor darf nicht vor dem Vektor selbst „sterben“

Ausblick: `new` und `delete`

```
new  
T[expr]
```

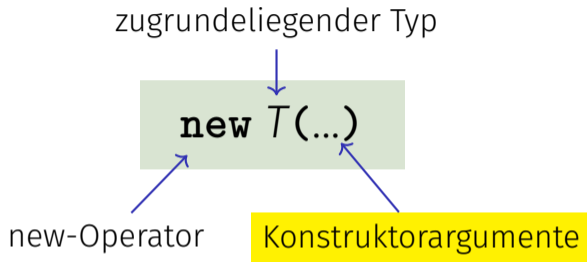
- Bisher: Speicher (lokale Variablen, Funktionsargumente) „lebt“ nur innerhalb eines Funktionsaufrufs
- Aber jetzt: Speicherblock im Vektor darf nicht vor dem Vektor selbst „sterben“
- Mittels **new** allozierter Speicher wird *nicht* automatisch *dealloziert* (= freigegeben)

Ausblick: `new` und `delete`

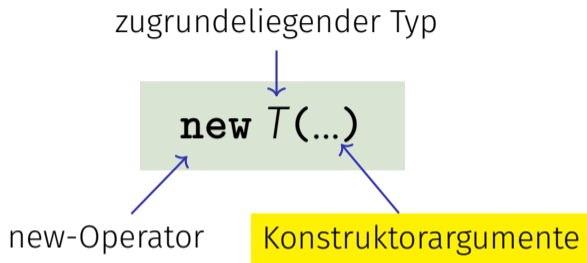
```
new  
T[expr]
```

- Bisher: Speicher (lokale Variablen, Funktionsargumente) „lebt“ nur innerhalb eines Funktionsaufrufs
- Aber jetzt: Speicherblock im Vektor darf nicht vor dem Vektor selbst „sterben“
- Mittels **new** allozierter Speicher wird *nicht* automatisch *dealloziert* (= freigegeben)
- Zu jedem **new** gehört ein **delete**, das den Speicher explizit freigibt → **in zwei Wochen**

Der `new`-Ausdruck (ohne Arrays)

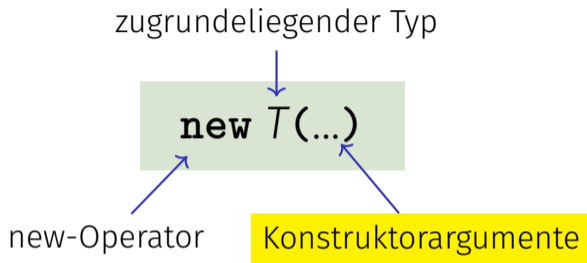


Der `new`-Ausdruck (ohne Arrays)



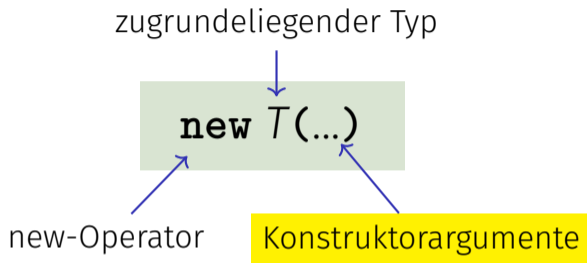
- **Effekt:** Speicher für ein neues Objekt vom Typ T wird alloziert ...

Der `new`-Ausdruck (ohne Arrays)



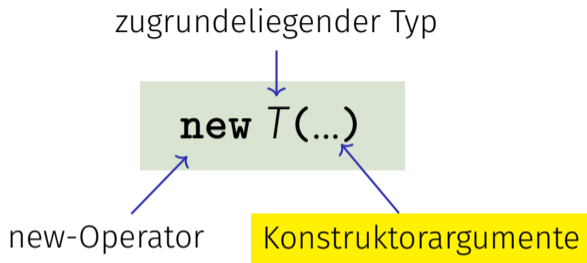
- **Effekt:** Speicher für ein neues Objekt vom Typ T wird alloziert ...
- ...und mit Hilfe des passenden Konstruktors initialisiert

Der `new`-Ausdruck (ohne Arrays)



- **Effekt:** Speicher für ein neues Objekt vom Typ T wird alloziert ...
- ...und mit Hilfe des passenden Konstruktors initialisiert
- **Wert:** Adresse des neuen T -Objekts, **Typ:** Zeiger T^*

Der `new`-Ausdruck (ohne Arrays)



- **Effekt:** Speicher für ein neues Objekt vom Typ T wird alloziert ...
- ...und mit Hilfe des passenden Konstruktors initialisiert
- **Wert:** Adresse des neuen T -Objekts, **Typ:** Zeiger T^*
- Auch hier gilt: Objekt „lebt“ bis es explizit gelöscht wird (Nutzen wird später klarer werden)

Zeiger-Typen

T* Zeiger-Typ für zugrunde liegenden
Typ **T**

Ein Ausdruck vom Typ **T*** heisst *Zeiger (auf T)*

Zeiger-Typen

T* Zeiger-Typ für zugrunde liegenden
Typ **T**

Ein Ausdruck vom Typ **T*** heisst *Zeiger (auf T)*

```
int* p; // Zeiger auf einen int
std::string* q; // Zeiger auf einen std::string
```

Zeiger-Typen

T* Zeiger-Typ für zugrunde liegenden Typ **T**

Wo ein T^* draufsteht, muss auch ein T drin sein

```
int* p = ...;  
std::string* q = p; // Kompilerfehler!
```

Zeiger-Typen

Wert eines Zeigers auf **T** ist die *Adresse* eines Objektes vom Typ **T**

Zeiger-Typen

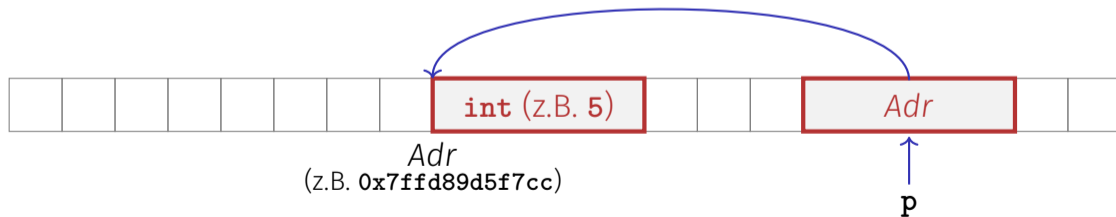
Wert eines Zeigers auf **T** ist die *Adresse* eines Objektes vom Typ **T**

```
int* p = ...;  
std::cout << p; // z.B. 0x7ffd89d5f7cc
```

Zeiger-Typen

Wert eines Zeigers auf **T** ist die Adresse eines Objektes vom Typ **T**

```
int* p = ...;  
std::cout << p; // z.B. 0x7ffd89d5f7cc
```



Adress-Operator

Frage: Wie kommt man an die Adresse eines Objekts?

1. Direkt beim Erzeugen eines neuen Objekts mittels **new**

Adress-Operator

Frage: Wie kommt man an die Adresse eines Objekts?

1. Direkt beim Erzeugen eines neuen Objekts mittels **new**
2. Für bestehende Objekte: Mittels des *Adress-Operators* **&**

`&expr` ← expr: L-Wert vom Typ *T*

Adress-Operator

Frage: Wie kommt man an die Adresse eines Objekts?

1. Direkt beim Erzeugen eines neuen Objekts mittels **new**
2. Für bestehende Objekte: Mittels des *Adress-Operators* **&**

&expr ← expr: L-Wert vom Typ *T*

- **Wert** des Ausdrucks: Die *Adresse* des Objekts (L-Wertes) *expr*

Adress-Operator

Frage: Wie kommt man an die Adresse eines Objekts?

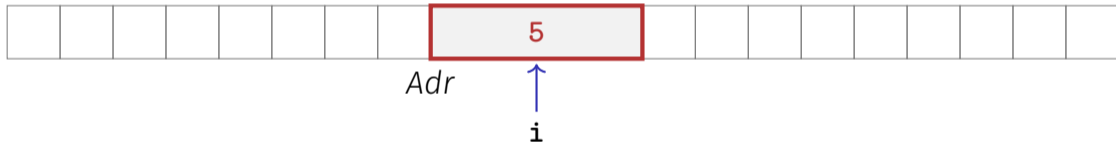
1. Direkt beim Erzeugen eines neuen Objekts mittels **new**
2. Für bestehende Objekte: Mittels des *Adress-Operators* **&**

&expr ← expr: L-Wert vom Typ *T*

- **Wert** des Ausdrucks: Die *Adresse* des Objekts (L-Wertes) *expr*
- **Typ** des Ausdrucks: Ein Zeiger T^* (vom Typ *T*)

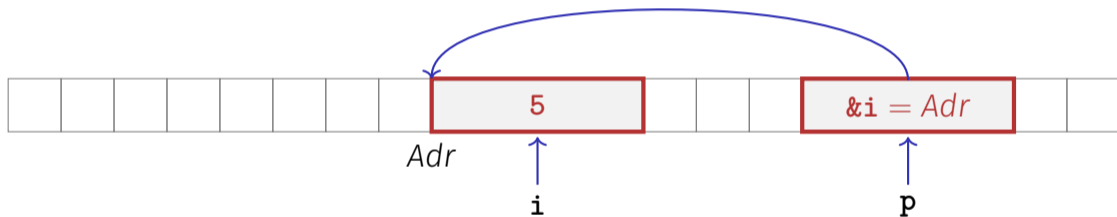
Adress-Operator

```
int i = 5; // i initialisiert mit 5
```



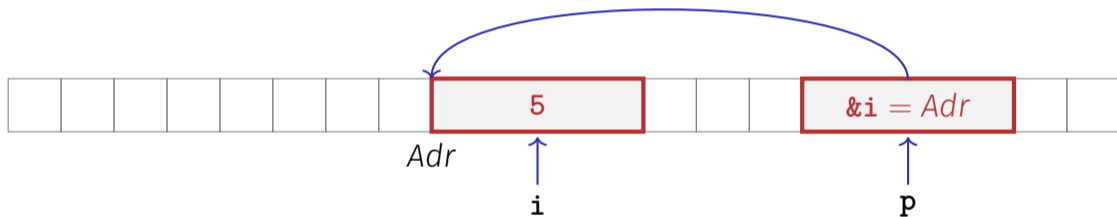
Adress-Operator

```
int i = 5; // i initialisiert mit 5  
int* p = &i; // p initialisiert mit der Adresse von i
```



Adress-Operator

```
int i = 5; // i initialisiert mit 5  
int* p = &i; // p initialisiert mit der Adresse von i
```



Nächste Frage: Wie „folgt“ man einem Zeiger?

Dereferenz-Operator

Antwort: Mittels des *Dereferenz-Operators* *

`*expr`

expr: R-Wert vom Typ T^*

Dereferenz-Operator

Antwort: Mittels des *Dereferenz-Operators* *

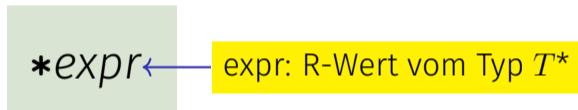


**expr* ← expr: R-Wert vom Typ T^*

- **Wert** des Ausdrucks: Der Wert des Objekts an der durch *expr* bestimmten Adresse

Dereferenz-Operator

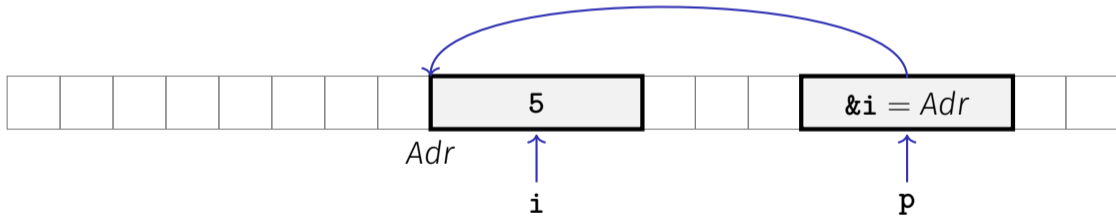
Antwort: Mittels des *Dereferenz-Operators* *



- **Wert** des Ausdrucks: Der Wert des Objekts an der durch *expr* bestimmten Adresse
- **Typ** des Ausdrucks: T

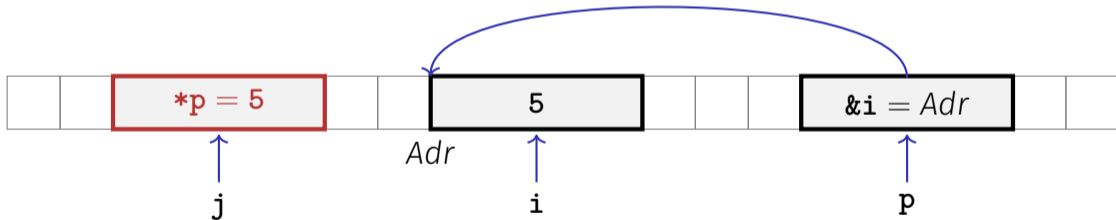
Dereferenz-Operator

```
int i = 5;  
int* p = &i; // p = Adresse von i
```

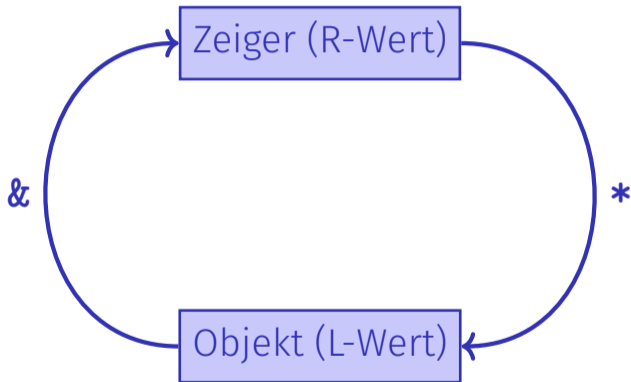


Dereferenz-Operator

```
int i = 5;  
int* p = &i; // p = Adresse von i  
int j = *p; // j = 5
```



Adress- und Dereferenzoperator



Eselsbrücke

Die Deklaration

```
T* p; // p ist vom Typ „Zeiger auf T“
```

Eselsbrücke

Die Deklaration

```
T* p; // p ist vom Typ „Zeiger auf T“
```

kann gelesen werden als

```
T *p; // *p ist vom Typ T
```

Null-Zeiger

- Spezieller Zeigerwert, der angibt, dass (noch) auf kein Objekt gezeigt wird
- Repräsentiert durch das Literal **nullptr** (konvertierbar nach **T***)

```
int* p = nullptr;
```

- Kann nicht dereferenziert werden (Laufzeitfehler)
- Dient der Vermeidung undefinierten Verhaltens

```
int* p; // Zugriff auf p ist undefiniertes Verhalten  
int* q = nullptr; // q hingegen zeigt explizit ins Nichts
```

Zeiger-Arithmetik: Zeiger plus int

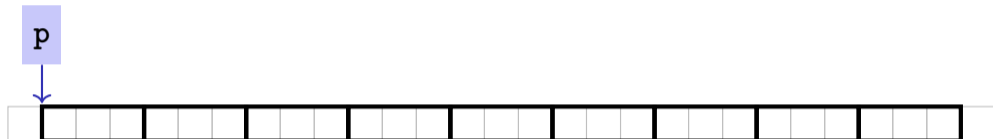
```
T* p = new T[n]; // p points to first array element
```



Frage: Wie zeigt man auf hintere Elemente?

Zeiger-Arithmetik: Zeiger plus int

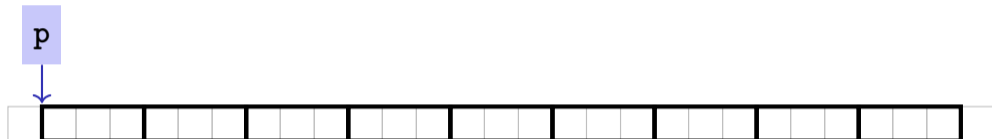
```
T* p = new T[n]; // p points to first array element
```



Frage: Wie zeigt man auf hintere Elemente? → mittels
Zeiger-Arithmetik:

Zeiger-Arithmetik: Zeiger plus int

```
T* p = new T[n]; // p points to first array element
```

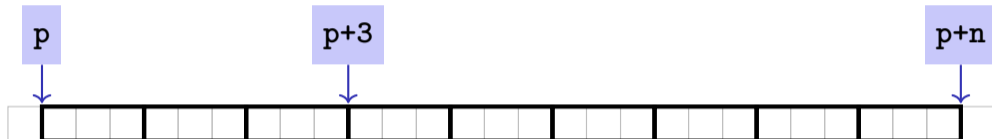


Frage: Wie zeigt man auf hintere Elemente? → mittels *Zeiger-Arithmetik*:

- `p` gibt die Adresse des *ersten* Array-Elements, `*p` dessen Wert

Zeiger-Arithmetik: Zeiger plus int

```
T* p = new T[n]; // p points to first array element
```



Frage: Wie zeigt man auf hintere Elemente? → mittels *Zeiger-Arithmetik*:

- p gibt die Adresse des *ersten* Array-Elements, $*p$ dessen Wert
- $*(p + i)$ gibt den Wert des *i-ten* Array-Elements, für $0 \leq i < n$

Zeiger-Arithmetik: Zeiger plus int

```
T* p = new T[n]; // p points to first array element
```



Frage: Wie zeigt man auf hintere Elemente? → mittels *Zeiger-Arithmetik*:

- p gibt die Adresse des *ersten* Array-Elements, $*p$ dessen Wert
- $*(p + i)$ gibt den Wert des *i-ten* Array-Elements, für $0 \leq i < n$
- $*p$ ist äquivalent zu $*(p + 0)$

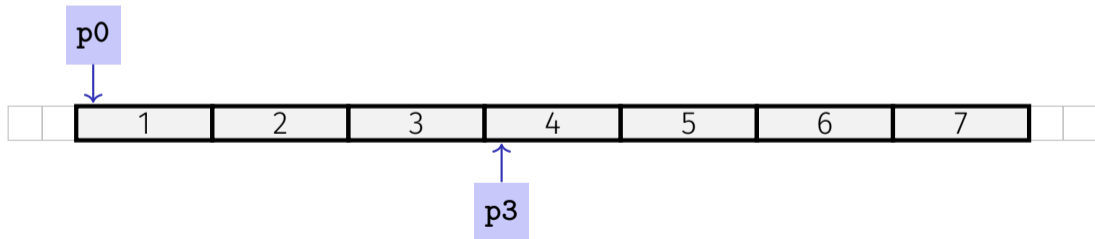
Zeiger-Arithmetik: Zeiger plus int

```
int* p0 = new int[7]{1,2,3,4,5,6,7}; // p0 points to 1st element
```



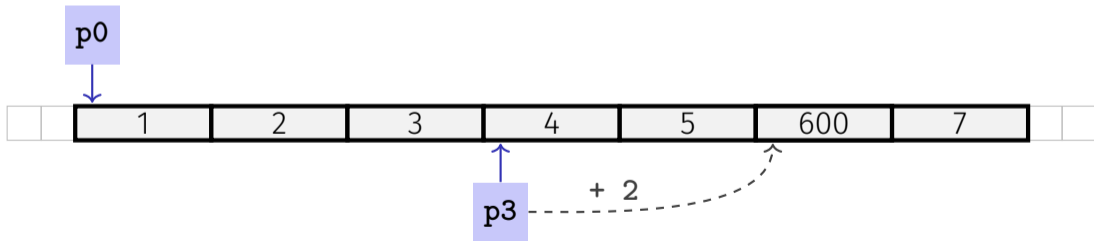
Zeiger-Arithmetik: Zeiger plus int

```
int* p0 = new int[7]{1,2,3,4,5,6,7}; // p0 points to 1st element  
int* p3 = p0 + 3; // p3 points to 4th element
```



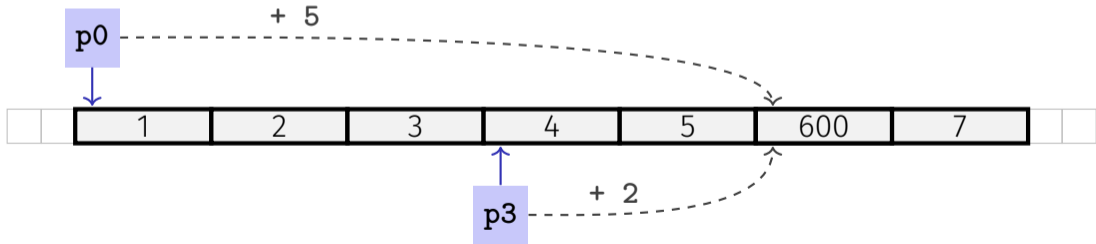
Zeiger-Arithmetik: Zeiger plus int

```
int* p0 = new int[7]{1,2,3,4,5,6,7}; // p0 points to 1st element
int* p3 = p0 + 3; // p3 points to 4th element
*(p3 + 2) = 600; // set value of 6th element to 600
std::cout << *(p0 + 5);
```



Zeiger-Arithmetik: Zeiger plus int

```
int* p0 = new int[7]{1,2,3,4,5,6,7}; // p0 points to 1st element
int* p3 = p0 + 3; // p3 points to 4th element
*(p3 + 2) = 600; // set value of 6th element to 600
std::cout << *(p0 + 5); // output 6th element's value (i.e. 600)
```



Sequenzielle Zeiger-Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```



Sequenzielle Zeiger-Iteration

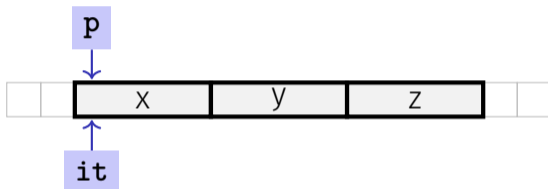
```
char* p = new char[3]{'x', 'y', 'z'};
```



```
for (char* it = p;  
     it != p + 3;  
     ++it) {  
  
    std::cout << *it << ' ';  
}
```


Sequenzielle Zeiger-Iteration

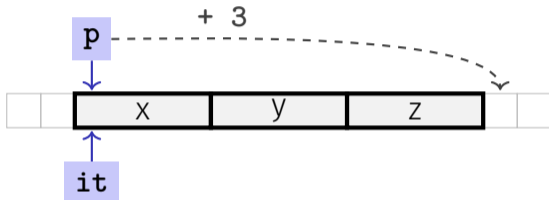
```
char* p = new char[3]{'x', 'y', 'z'};
```



```
for (char* it = p; it zeigt aufs erste Element  
    it != p + 3;  
    ++it) {  
  
    std::cout << *it << ' ';<br>  
}
```

Sequenzielle Zeiger-Iteration

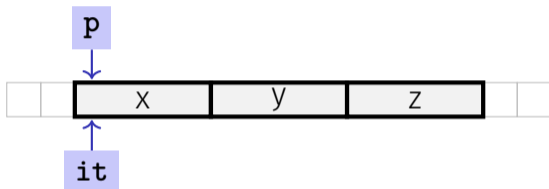
```
char* p = new char[3]{'x', 'y', 'z'};
```



```
for (char* it = p;  
    it != p + 3; ← Abbruch falls Ende erreicht  
    ++it) {  
  
    std::cout << *it << ' ';  
}
```

Sequenzielle Zeiger-Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```



```
for (char* it = p;  
     it != p + 3;  
     ++it) {
```

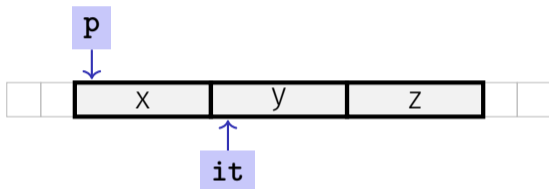
```
    std::cout << *it << ' ';
```

```
}
```

Aktuelles Element ausgeben: 'x'

Sequenzielle Zeiger-Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```



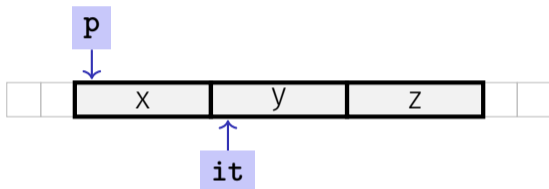
```
for (char* it = p;  
    it != p + 3;  
    ++it) {
```

Zeiger elementweise voranschieben

```
    std::cout << *it << ' '; // x  
}
```

Sequenzielle Zeiger-Iteration

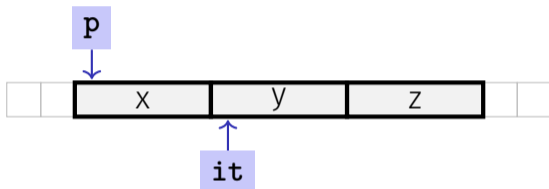
```
char* p = new char[3]{'x', 'y', 'z'};
```



```
for (char* it = p;  
     it != p + 3;  
     ++it) {  
  
    std::cout << *it << ' '; // x  
}
```

Sequenzielle Zeiger-Iteration

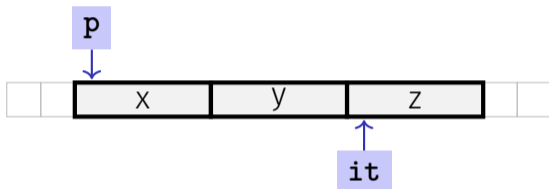
```
char* p = new char[3]{'x', 'y', 'z'};
```



```
for (char* it = p;  
     it != p + 3;  
     ++it) {  
  
    std::cout << *it << ' '; // x y  
}
```

Sequenzielle Zeiger-Iteration

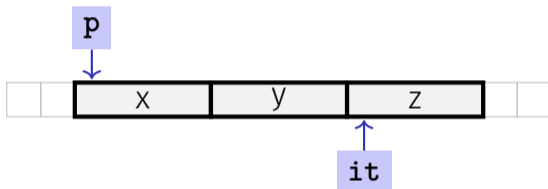
```
char* p = new char[3]{'x', 'y', 'z'};
```



```
for (char* it = p;  
     it != p + 3;  
     ++it) {  
  
    std::cout << *it << ' '; // x y  
}
```

Sequenzielle Zeiger-Iteration

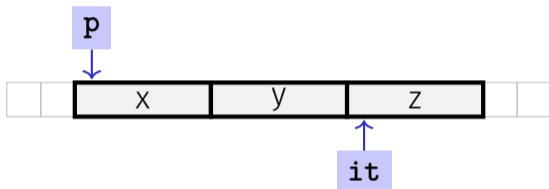
```
char* p = new char[3]{'x', 'y', 'z'};
```



```
for (char* it = p;  
     it != p + 3;  
     ++it) {  
  
    std::cout << *it << ' '; // x y  
}
```


Sequenzielle Zeiger-Iteration

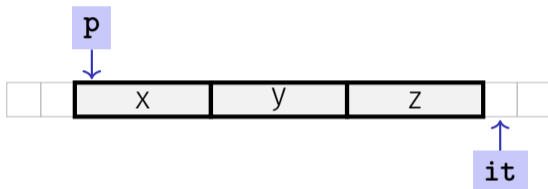
```
char* p = new char[3]{'x', 'y', 'z'};
```



```
for (char* it = p;  
     it != p + 3;  
     ++it) {  
  
    std::cout << *it << ' '; // x y z  
}
```

Sequenzielle Zeiger-Iteration

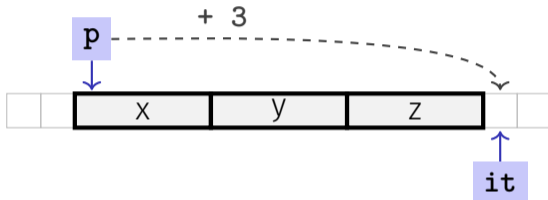
```
char* p = new char[3]{'x', 'y', 'z'};
```



```
for (char* it = p;  
     it != p + 3;  
     ++it) {  
  
    std::cout << *it << ' '; // x y z  
}
```

Sequenzielle Zeiger-Iteration

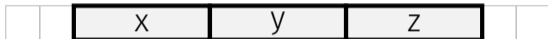
```
char* p = new char[3]{'x', 'y', 'z'};
```



```
for (char* it = p;  
     it != p + 3;  
     ++it) {  
  
    std::cout << *it << ' '; // x y z  
}
```

Wahlfreier Zugriff auf Arrays

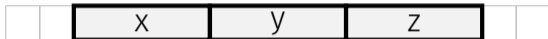
```
char* p = new char[3]{'x', 'y', 'z'};
```



- Der Ausdruck `*(p + i)`
- kann auch geschrieben werden als `p[i]`

Wahlfreier Zugriff auf Arrays

```
char* p = new char[3]{'x', 'y', 'z'};
```



- Der Ausdruck `*(p + i)`
- kann auch geschrieben werden als `p[i]`
- z.B. `p[1] == *(p + 1) == 'y'`

Wahlfreier Zugriff auf Arrays

Iteration über ein Array mittels Indizes und *wahlfreiem Zugriff*:

```
char* p = new char[3]{'x', 'y', 'z'};

for (int i = 0; i < 3; ++i)
    std::cout << p[i] << ' ';
```

Wahlfreier Zugriff auf Arrays

Iteration über ein Array mittels Indizes und *wahlfreiem Zugriff*:

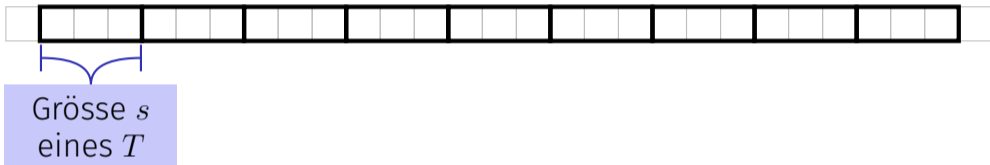
```
char* p = new char[3]{'x', 'y', 'z'};

for (int i = 0; i < 3; ++i)
    std::cout << p[i] << ' ';
```

Aber: Dies ist weniger *effizient* als der vorher gezeigte *sequenzielle* Zugriff mittels Zeiger-Iteration

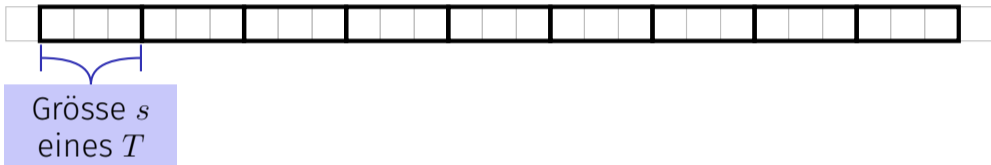
Wahlfreier Zugriff auf Arrays

```
T* p = new T[n];
```



Wahlfreier Zugriff auf Arrays

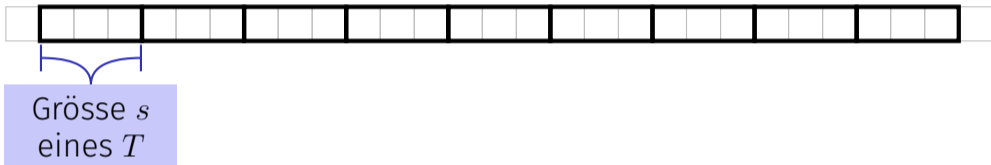
```
T* p = new T[n];
```



- Zugriff $p[i]$, also $*(p + i)$, „kostet“ Berechnung $p + i \cdot s$

Wahlfreier Zugriff auf Arrays

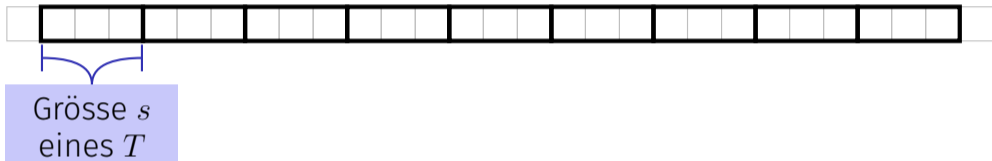
```
T* p = new T[n];
```



- Zugriff $p[i]$, also $*(p + i)$, „kostet“ Berechnung $p + i \cdot s$
- Iteration mittels *wahlfreiem Zugriff* ($p[0], p[1], \dots$) kostet eine Addition und eine Multiplikation pro Zugriff

Wahlfreier Zugriff auf Arrays

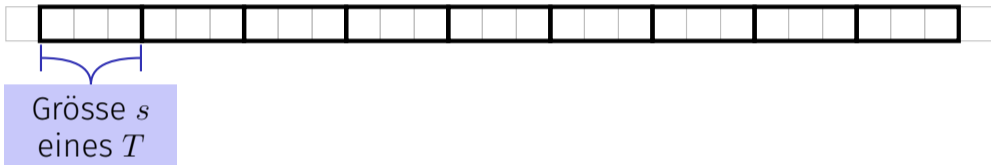
```
T* p = new T[n];
```



- Zugriff $p[i]$, also $*(p + i)$, „kostet“ Berechnung $p + i \cdot s$
- Iteration mittels *wahlfreiem Zugriff* ($p[0], p[1], \dots$) kostet eine Addition und eine Multiplikation pro Zugriff
- Iteration mittels *sequentiell* Zugriff ($++p, ++p, \dots$) kostet nur eine Addition pro Zugriff

Wahlfreier Zugriff auf Arrays

```
T* p = new T[n];
```



- Zugriff $p[i]$, also $*(p + i)$, „kostet“ Berechnung $p + i \cdot s$
- Iteration mittels *wahlfreiem Zugriff* ($p[0], p[1], \dots$) kostet eine Addition und eine Multiplikation pro Zugriff
- Iteration mittels *sequentiellem Zugriff* ($++p, ++p, \dots$) kostet nur eine Addition pro Zugriff
- Sequenzieller Zugriff ist daher für Iterationen zu bevorzugen

Ein Buch lesen ... mit wahlfreiem Zugriff

Wahlfreier Zugriff

- öffne Buch auf S.1
- klappe Buch zu
- öffne Buch auf S.2-3
- klappe Buch zu
- öffne Buch auf S.4-5
- klappe Buch zu
-

Wahlfreier Zugriff

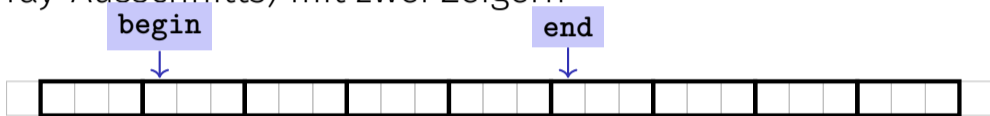
- öffne Buch auf S.1
- klappe Buch zu
- öffne Buch auf S.2-3
- klappe Buch zu
- öffne Buch auf S.4-5
- klappe Buch zu
-

Sequenzieller Zugriff

- öffne Buch auf S.1
- blättere um
- blättere um
- blättere um
- blättere um
- blättere um
- ...

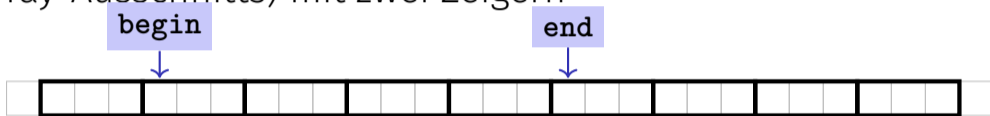
Arrays in Funktionen

Konvention in C++: Übergabe eines Arrays (oder eines Array-Ausschnitts) mit zwei Zeigern



Arrays in Funktionen

Konvention in C++: Übergabe eines Arrays (oder eines Array-Ausschnitts) mit zwei Zeigern



- **begin**: Zeiger auf das erste Element
- **end**: Zeiger *hinter* das letzte Element

Arrays in Funktionen

Konvention in C++: Übergabe eines Arrays (oder eines Array-Ausschnitts) mit zwei Zeigern



- **begin**: Zeiger auf das erste Element
- **end**: Zeiger *hinter* das letzte Element
- **[begin, end)** bezeichnet die Elemente des Array-Ausschnitts

Arrays in Funktionen

Konvention in C++: Übergabe eines Arrays (oder eines Array-Ausschnitts) mit zwei Zeigern



- **begin**: Zeiger auf das erste Element
- **end**: Zeiger *hinter* das letzte Element
- **[begin, end)** bezeichnet die Elemente des Array-Ausschnitts
- **[begin, end)** ist leer, wenn **begin == end**
- **[begin, end)** muss ein *gültiger Bereich* sein, d.h. ein echter (evtl. leerer) Array-Ausschnitt

Arrays in (mutierenden) Funktionen: fill

```
// PRE: [begin, end) ist ein gültiger Bereich
// POST: Jedes Element in [begin, end) wurde auf value gesetzt
void fill(int* begin, int* end, int value) {
    for (int* p = begin; p != end; ++p)
        *p = value;
}
```

Arrays in (mutierenden) Funktionen: fill

```
// PRE: [begin, end) ist ein gültiger Bereich  
// POST: Jedes Element in [begin, end) wurde auf value gesetzt  
void fill(int* begin, int* end, int value) {  
    for (int* p = begin; p != end; ++p)  
        *p = value;  
}
```

```
int* p = new int[5];  
fill(p, p+5, 1); // Array bei p wird zu {1, 1, 1, 1, 1}
```

Funktionen mit/ohne Effekt

- Zeiger können, wie auch Referenzen, für Funktionen mit Effekt verwendet werden. Beispiel: `fill`

Funktionen mit/ohne Effekt

- Zeiger können, wie auch Referenzen, für Funktionen mit Effekt verwendet werden. Beispiel: **fill**
- Aber viele Funktionen haben keinen Effekt, sie lesen Daten nur
- ⇒ Verwendung von **const**

Funktionen mit/ohne Effekt

- Zeiger können, wie auch Referenzen, für Funktionen mit Effekt verwendet werden. Beispiel: `fill`
- Aber viele Funktionen haben keinen Effekt, sie lesen Daten nur
- ⇒ Verwendung von `const`
- Bisher, zum Beispiel:

```
const int zero = 0;  
const int& nil = zero;
```

Positionierung von Const

`const T` ist äquivalent zu `T const` (und kann auch so geschrieben werden):

```
const int zero = ...  $\iff$  int const zero = ...  
const int& nil = ...  $\iff$  int const& nil = ...
```


Positionierung von Const

`const T` ist äquivalent zu `T const` (und kann auch so geschrieben werden):

```
const int zero = ...  $\iff$  int const zero = ...  
const int& nil = ...  $\iff$  int const& nil = ...
```

Beide Schreibweisen werden in der Praxis genutzt

Const und Zeiger

Lies Deklaration von rechts nach links

```
int const p1;
```

p1 ist eine konstante Ganzzahl

Const und Zeiger

Lies Deklaration von rechts nach links

```
int const p1;
```

p1 ist eine konstante Ganzzahl

```
int const* p2;
```

p2 ist ein Zeiger auf eine konstante Ganzzahl

Const und Zeiger

Lies Deklaration von rechts nach links

```
int const p1;
```

p1 ist eine konstante Ganzzahl

```
int const* p2;
```

p2 ist ein Zeiger auf eine konstante Ganzzahl

```
int* const p3;
```

p3 ist ein konstanter Zeiger auf eine Ganzzahl

Const und Zeiger

Lies Deklaration von rechts nach links

```
int const p1;
```

p1 ist eine konstante Ganzzahl

```
int const* p2;
```

p2 ist ein Zeiger auf eine konstante Ganzzahl

```
int* const p3;
```

p3 ist ein konstanter Zeiger auf eine Ganzzahl

```
int const* const p4;
```

p4 ist ein konstanter Zeiger auf eine konstante Ganzzahl

Nicht-mutierende Funktionen: `print`

```
// PRE: [begin, end) ist ein gültiger Bereich
// POST: Die Werte in [begin, end) wurden ausgegeben
void print(
    int const* const begin,
    const int* const end) {

    for (int const* p = begin; p != end; ++p)
        std::cout << *p << ' ';
}
```

Nicht-mutierende Funktionen: `print`

```
// PRE: [begin, end) ist ein gültiger Bereich
// POST: Die Werte in [begin, end) wurden ausgegeben
void print(
    int const* const begin, ← Const-Zeiger auf const-int
    const int* const end) ← { Ebenfalls (aber andere Schreibweise)
    for (int const* p = begin; p != end; ++p)
        std::cout << *p << ' ';
}
```

Nicht-mutierende Funktionen: `print`

```
// PRE: [begin, end) ist ein gültiger Bereich  
// POST: Die Werte in [begin, end) wurden ausgegeben
```

```
void print(  
    int const* const begin,  
    const int* const end) {
```

Const-Zeiger auf const-int

```
int const* const begin,
```

```
const int* const end) {
```

Ebenfalls (aber andere Schreibweise)

```
for (int const* p = begin; p != end; ++p)
```

```
    std::cout << *p << ' ';
```

```
}
```

Zeiger, *nicht const*, auf const-int

Arrays, new, Zeiger: Abschluss

- Arrays sind kontinuierliche Speicherblöcke statisch unbestimmter Grösse

Arrays, `new`, Zeiger: Abschluss

- Arrays sind kontinuierliche Speicherblöcke statisch unbestimmter Grösse
- `new T[n]` alloziert ein T -Array der Grösse n

Arrays, new, Zeiger: Abschluss

- Arrays sind kontinuierliche Speicherblöcke statisch unbestimmter Grösse
- `new T[n]` alloziert ein T -Array der Grösse n
- `T* p = new T[n]`: Zeiger `p` zeigt auf das erste Array-Element

Arrays, `new`, Zeiger: Abschluss

- Arrays sind kontinuierliche Speicherblöcke statisch unbestimmter Grösse
- `new T[n]` alloziert ein T -Array der Grösse n
- `T* p = new T[n]`: Zeiger `p` zeigt auf das erste Array-Element
- Zeigerarithmetik ermöglicht Zugriff auf hintere Array-Elemente

Arrays, `new`, Zeiger: Abschluss

- Arrays sind kontinuierliche Speicherblöcke statisch unbestimmter Grösse
- `new T[n]` alloziert ein T -Array der Grösse n
- `T* p = new T[n]`: Zeiger `p` zeigt auf das erste Array-Element
- Zeigerarithmetik ermöglicht Zugriff auf hintere Array-Elemente
- Sequenzielle Iteration über Arrays mittels Zeigern ist effizienter als wahlfreier Zugriff

Arrays, `new`, Zeiger: Abschluss

- Arrays sind kontinuierliche Speicherblöcke statisch unbestimmter Grösse
- `new T[n]` alloziert ein T -Array der Grösse n
- `T* p = new T[n]`: Zeiger `p` zeigt auf das erste Array-Element
- Zeigerarithmetik ermöglicht Zugriff auf hintere Array-Elemente
- Sequenzielle Iteration über Arrays mittels Zeigern ist effizienter als wahlfreier Zugriff
- `new T` alloziert Speicher für (und initialisiert) ein einzelnes T -Objekt und liefert einen Zeiger darauf

Arrays, `new`, Zeiger: Abschluss

- Arrays sind kontinuierliche Speicherblöcke statisch unbestimmter Grösse
- `new T[n]` alloziert ein T -Array der Grösse n
- `T* p = new T[n]`: Zeiger `p` zeigt auf das erste Array-Element
- Zeigerarithmetik ermöglicht Zugriff auf hintere Array-Elemente
- Sequenzielle Iteration über Arrays mittels Zeigern ist effizienter als wahlfreier Zugriff
- `new T` alloziert Speicher für (und initialisiert) ein einzelnes T -Objekt und liefert einen Zeiger darauf
- Zeiger können auf etwas (nicht) `const`es zeigen und selbst (nicht) `const` sein

Arrays, `new`, Zeiger: Abschluss

- Arrays sind kontinuierliche Speicherblöcke statisch unbestimmter Grösse
- `new T[n]` alloziert ein T -Array der Grösse n
- `T* p = new T[n]`: Zeiger `p` zeigt auf das erste Array-Element
- Zeigerarithmetik ermöglicht Zugriff auf hintere Array-Elemente
- Sequenzielle Iteration über Arrays mittels Zeigern ist effizienter als wahlfreier Zugriff
- `new T` alloziert Speicher für (und initialisiert) ein einzelnes T -Objekt und liefert einen Zeiger darauf
- Zeiger können auf etwas (nicht) `const`es zeigen und selbst (nicht) `const` sein
- Mittels `new` allozierter Speicher wird *nicht* automatisch freigegeben (mehr dazu demnächst)

Arrays, `new`, Zeiger: Abschluss

- Arrays sind kontinuierliche Speicherblöcke statisch unbestimmter Grösse
- `new T[n]` alloziert ein T -Array der Grösse n
- `T* p = new T[n]`: Zeiger `p` zeigt auf das erste Array-Element
- Zeigerarithmetik ermöglicht Zugriff auf hintere Array-Elemente
- Sequenzielle Iteration über Arrays mittels Zeigern ist effizienter als wahlfreier Zugriff
- `new T` alloziert Speicher für (und initialisiert) ein einzelnes T -Objekt und liefert einen Zeiger darauf
- Zeiger können auf etwas (nicht) `const`es zeigen und selbst (nicht) `const` sein
- Mittels `new` allozierter Speicher wird *nicht* automatisch freigegeben (mehr dazu demnächst)
- Zeiger und Referenzen sind verwandt, beide „verweisen“ auf Objekte im Speicher. Siehe auch die Extrafolien `pointers.pdf`)

Array-basierter Vektor

- Vektoren ... da war doch was 🤔

Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<T>` (später, nur kurz angeschnitten)

Array-basierter Vektor

- Vektoren ... da war doch was 🤔
- Nun wissen wir, wie man Speicherblöcke beliebiger Grösse allozieren kann ...

Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<T>` (später, nur kurz angeschnitten)

Array-basierter Vektor

- Vektoren ... da war doch was 🤔
- Nun wissen wir, wie man Speicherblöcke beliebiger Grösse allozieren kann ...
- ... und können einen Vektor, auf einem solchen Speicherblock aufbauend, implementieren

Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<T>` (später, nur kurz angeschnitten)

Array-basierter Vektor

- Vektoren ... da war doch was 🤔
- Nun wissen wir, wie man Speicherblöcke beliebiger Grösse allozieren kann ...
- ... und können einen Vektor, auf einem solchen Speicherblock aufbauend, implementieren
- **avec** – ein Array-basierter Vektor für **int**-Elemente

Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<T>` (später, nur kurz angeschnitten)

```
class avec {  
    // Private (internal) state:  
    int* elements;   
    unsigned int count;  
  
}
```

← Zeiger auf erstes Element

```
class avec {  
    // Private (internal) state:  
    int* elements; // Pointer to first element  
    unsigned int count; ←  
  
}
```

Anzahl Elemente

```
class avec {  
    // Private (internal) state:  
    int* elements; // Pointer to first element  
    unsigned int count; // Number of elements  
  
public: // Public interface:  
    avec(unsigned int size);  
    unsigned int size() const;  
    int& operator[](int i);  
    void print(std::ostream& sink) const;  
}
```



Konstruktor


```
class avec {  
    // Private (internal) state:  
    int* elements; // Pointer to first element  
    unsigned int count; // Number of elements  
  
public: // Public interface:  
    avec(unsigned int size); // Constructor  
    unsigned int size() const; ←  
    int& operator[](int i);  
    void print(std::ostream& sink) const;  
}
```

Grösse des Vektors

```
class avec {  
    // Private (internal) state:  
    int* elements; // Pointer to first element  
    unsigned int count; // Number of elements  
  
public: // Public interface:  
    avec(unsigned int size); // Constructor  
    unsigned int size() const; // Size of vector  
    int& operator[](int i);  
    void print(std::ostream& sink) const;  
}
```

Elementzugriff

```
class avec {  
    // Private (internal) state:  
    int* elements; // Pointer to first element  
    unsigned int count; // Number of elements  
  
public: // Public interface:  
    avec(unsigned int size); // Constructor  
    unsigned int size() const; // Size of vector  
    int& operator[](int i); // Access an element  
    void print(std::ostream& sink) const;  
}
```



Elemente ausgeben

```
class avec {  
    // Private (internal) state:  
    int* elements; // Pointer to first element  
    unsigned int count; // Number of elements  
  
public: // Public interface:  
    avec(unsigned int size); // Constructor  
    unsigned int size() const; // Size of vector  
    int& operator[](int i); // Access an element  
    void print(std::ostream& sink) const; // Output elems.  
}
```

Konstruktor `avec::avec()`

```
avec::avec(unsigned int size)
    : count(size) ← {
    elements = new int[size];
}
```

Grösse speichern

Konstruktor `avec::avec()`

```
avec::avec(unsigned int size)
    : count(size) {

    elements = new int[size];
}
```

← Speicher allozieren

Konstruktor `avec::avec()`

```
avec::avec(unsigned int size)
    : count(size) {

    elements = new int[size];
}
```

Nebenbemerkung: Vektor wird nicht mit einem Standardwert initialisiert

Exkurs: Zugriff auf Membervariablen

```
avec::avec(unsigned int size): count(size) {  
    elements = new int[size];  
}
```

- **elements** ist eine Membervariable unserer **avec**-Instanz

Exkurs: Zugriff auf Membervariablen

```
avec::avec(unsigned int size): count(size) {  
    elements = new int[size];  
}
```

- **elements** ist eine Membervariable unserer **avec**-Instanz
- Diese Instanz ist mittels des *Zeigers* **this** zugreifbar

Exkurs: Zugriff auf Membervariablen

```
avec::avec(unsigned int size): count(size) {  
    (*this).elements = new int[size];  
}
```

- **elements** ist eine Membervariable unserer **avec**-Instanz
- Diese Instanz ist mittels des *Zeigers* **this** zugreifbar
- **elements** ist eine Kurzschreibweise von **(*this).elements**

Exkurs: Zugriff auf Membervariablen

```
avec::avec(unsigned int size): count(size) {  
    this->elements = new int[size];  
}
```

- **elements** ist eine Membervariable unserer **avec**-Instanz
- Diese Instanz ist mittels des *Zeigers* **this** zugreifbar
- **elements** ist eine Kurzschreibweise von **(*this).elements**
- Äquivalent, aber kürzer: **this->elements**

Exkurs: Zugriff auf Membervariablen

```
avec::avec(unsigned int size): count(size) {  
    this->elements = new int[size];  
}
```

- **elements** ist eine Membervariable unserer **avec**-Instanz
- Diese Instanz ist mittels des *Zeigers* **this** zugreifbar
- **elements** ist eine Kurzschreibweise von **(*this).elements**
- Äquivalent, aber kürzer: **this->elements**
- Eselsbrücke: „Folge dem Zeiger zur Membervariablen“

Funktion `avec::size()`

```
int avec::size() const ← {  
    return this->count;  
}
```

Verändert den Vektor nicht

Funktion `avec::size()`

```
int avec::size() const {  
    return this->count;   
}
```

← Grösse zurückgeben

Anwendungsbeispiel:

```
avec v = avec(7);  
assert(v.size() == 7); // ok
```

Funktion `avec::operator []`

```
int& avec::operator [] (int i) {  
    return this->elements[i];  
}
```

 `i`-tes Element zurückgeben

Funktion `avec::operator []`

```
int& avec::operator [] (int i) {  
    return this->elements[i];  
}
```

Elementzugriff mit Indexüberprüfung:

```
int& avec::at(int i) const {  
    assert(0 <= i && i < this->count);  
  
    return this->elements[i];  
}
```


Funktion `avec::operator []`

```
int& avec::operator [] (int i) {  
    return this->elements[i];  
}
```

Anwendungsbeispiel:

```
avec v = avec(7);  
std::cout << v[6]; // Outputs a "random" value  
v[6] = 0;  
std::cout << v[6]; // Outputs 0
```

Funktion `avec::print()`


Elemente mittels sequenziellem Zugriff ausgeben:

```
void avec::print(std::ostream& sink) const {  
    for (int* p = this->elements; ← Zeiger auf erstes Element  
         p != this->elements + this->count;  
         ++p)  
    {  
        sink << *p << ' ' ;  
    }  
}
```

Funktion `avec::print()`

Elemente mittels sequenziellem Zugriff ausgeben:

```
void avec::print(std::ostream& sink) const {  
    for (int* p = this->elements;  
         p != this->elements + this->count;   
         ++p)  
    {  
        sink << *p << ' ' ;  
    }  
}
```



Abbruch falls hinter
letztem Ele-
ment

Funktion `avec::print()`

Elemente mittels sequenziellem Zugriff ausgeben:

```
void avec::print(std::ostream& sink) const {  
    for (int* p = this->elements;  
         p != this->elements + this->count;  
         ++p) {  
        sink << *p << ' ' ;  
    }  
}
```

← Zeiger elementweise voranschieben

Funktion `avec::print()`

Elemente mittels sequenziellem Zugriff ausgeben:

```
void avec::print(std::ostream& sink) const {  
    for (int* p = this->elements;  
         p != this->elements + this->count;  
         ++p)  
    {  
        sink << *p << ' '; ←  
    }  
}
```

Aktuelles Element ausgeben

Funktion `avec::print()`

Abschluss: Ausgabeoperator überladen:

```
_____ operator<< (_____ sink,  
                    _____ vec) {  
    vec.print(sink);  
    return _____;  
}
```

Funktion `avec::print()`

Abschluss: Ausgabeoperator überladen:

```
std::ostream& operator<<(std::ostream& sink,  
                        const avec& vec) {  
    vec.print(sink);  
    return sink;  
}
```

Funktion `avec::print()`

Abschluss: Ausgabeoperator überladen:

```
std::ostream& operator<<(std::ostream& sink,  
                        const avec& vec) {  
    vec.print(sink);  
    return sink;  
}
```

Beobachtungen:

- Konstante Referenz auf `vec`, da unverändert

Funktion `avec::print()`

Abschluss: Ausgabeoperator überladen:

```
std::ostream& operator<<(std::ostream& sink,  
                        const avec& vec) {  
    vec.print(sink);  
    return sink;  
}
```

Beobachtungen:

- Konstante Referenz auf `vec`, da unverändert
- Aber nicht auf `sink`: Elemente ausgeben gleich Veränderung

Funktion `avec::print()`

Abschluss: Ausgabeoperator überladen:

```
std::ostream& operator<<(std::ostream& sink,  
                        const avec& vec) {  
    vec.print(sink);  
    return sink;  
}
```

Beobachtungen:

- Konstante Referenz auf `vec`, da unverändert
- Aber nicht auf `sink`: Elemente ausgeben gleich Veränderung
- `sink` zurückgeben, um verkettete Ausgaben zu ermöglichen, z.B. `std::cout << v << '\n'`

Weitere Funktionen

```
class avec {  
    ...  
    void push_front(int e)      // Prepend e to vector  
    void push_back(int e)      // Append e to vector  
    void remove(unsigned int i) // Cut out ith element  
    ...  
}
```

Weitere Funktionen

```
class avec {  
    ...  
    void push_front(int e)      // Prepend e to vector  
    void push_back(int e)     // Append e to vector  
    void remove(unsigned int i) // Cut out ith element  
    ...  
}
```

Gemeinsamkeit: Diese Operationen müssen die *Grösse* des Vektors verändern

Arrays vergrössern/verkleinern

Ein allozierter Speicherblock (z.B. `new int [3]`) kann nicht nachträglich vergrössert/verkleinert werden

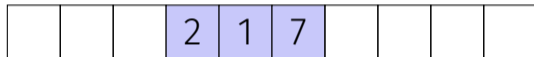
Arrays vergrössern/verkleinern

Ein allozierter Speicherblock (z.B. `new int [3]`) kann nicht nachträglich vergrössert/verkleinert werden

2	1	7
---	---	---

Arrays vergrössern/verkleinern

Ein allozierter Speicherblock (z.B. `new int [3]`) kann nicht nachträglich vergrössert/verkleinert werden

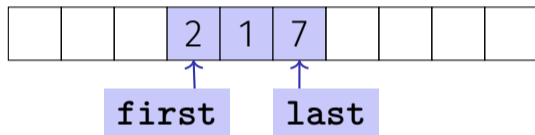


Möglichkeit:

- Mehr Speicher als initial nötig allozieren

Arrays vergrössern/verkleinern

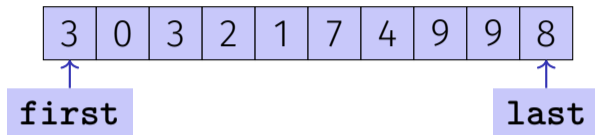
Ein allozierter Speicherblock (z.B. `new int [3]`) kann nicht nachträglich vergrössert/verkleinert werden



Möglichkeit:

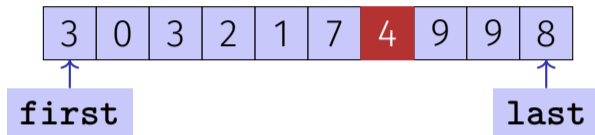
- Mehr Speicher als initial nötig allozieren
- Befüllen aus der Mitte heraus, mittels Zeigern auf erstes und letztes Element

Arrays vergrössern/verkleiner



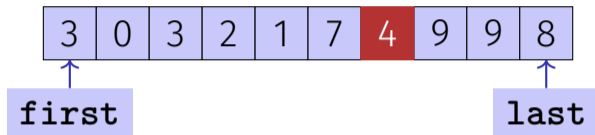
- Aber irgendwann sind alle Plätze belegt
- Dann nötig: Grösseren Speicherblock allozieren und Daten umkopieren

Arrays vergrössern/verkleiner

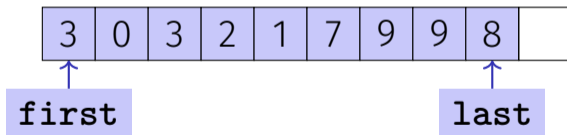


Elemente löschen erfordert verschieben (via kopieren) aller vorhergehenden oder nachfolgenden Elemente

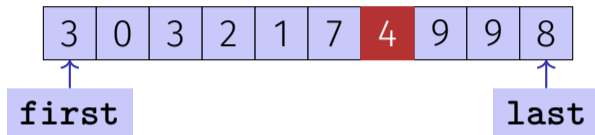
Arrays vergrössern/verkleiner



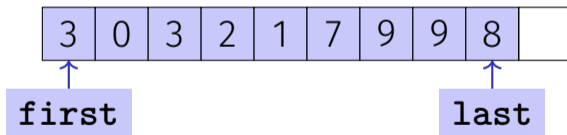
Elemente löschen erfordert verschieben (via kopieren) aller vorhergehenden oder nachfolgenden Elemente



Arrays vergrössern/verkleiner



Elemente löschen erfordert verschieben (via kopieren) aller vorhergehenden oder nachfolgenden Elemente



Ähnlich: Einfügen an beliebiger Position

21. Dynamische Datenstrukturen II

Verkettete Listen, Vektoren als verkettete Listen

Anderes Speicherlayout: Verkettete Liste

- **Kein** zusammenhängender Speicherbereich und **kein** wahlfreier Zugriff



Anderes Speicherlayout: Verkettete Liste

- **Kein** zusammenhängender Speicherbereich und **kein** wahlfreier Zugriff
- Jedes Element zeigt auf seinen Nachfolger



Anderes Speicherlayout: Verkettete Liste

- **Kein** zusammenhängender Speicherbereich und **kein** wahlfreier Zugriff
- Jedes Element zeigt auf seinen Nachfolger



Anderes Speicherlayout: Verkettete Liste

- **Kein** zusammenhängender Speicherbereich und **kein** wahlfreier Zugriff
- Jedes Element zeigt auf seinen Nachfolger
- Einfügen und Löschen **beliebiger** Elemente ist einfach



Anderes Speicherlayout: Verkettete Liste

- **Kein** zusammenhängender Speicherbereich und **kein** wahlfreier Zugriff
- Jedes Element zeigt auf seinen Nachfolger
- Einfügen und Löschen **beliebiger** Elemente ist einfach



⇒ Unser Vektor kann als verkettete Liste realisiert werden

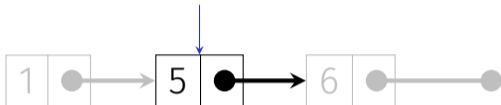
Verkettete Liste: Zoom



```
struct llnode {  
    int value;  
    llnode* next;  
  
    llnode(int v, llnode* n): value(v), next(n) {} // Constructor  
};
```

Verkettete Liste: Zoom

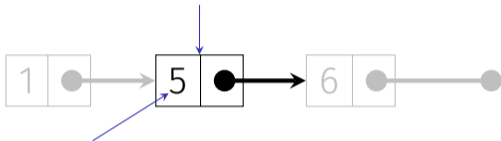
Element (Typ `struct llnode`)



```
struct llnode {  
    int value;  
    llnode* next;  
  
    llnode(int v, llnode* n): value(v), next(n) {} // Constructor  
};
```

Verkettete Liste: Zoom

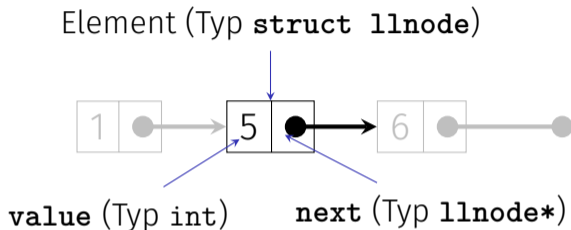
Element (Typ `struct llnode`)



value (Typ `int`)

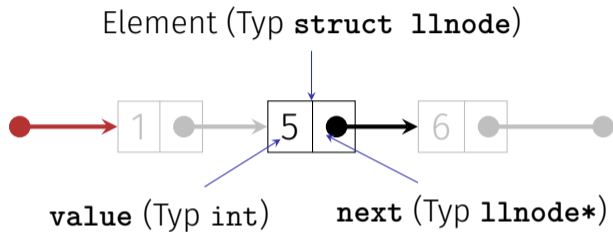
```
struct llnode {  
    int value;  
    llnode* next;  
  
    llnode(int v, llnode* n): value(v), next(n) {} // Constructor  
};
```

Verkettete Liste: Zoom



```
struct llnode {  
    int value;  
    llnode* next;  
  
    llnode(int v, llnode* n): value(v), next(n) {} // Constructor  
};
```

Vektor = Zeiger aufs erste Element



```
class llvec {  
    llnode* head;  
public: // Public interface identical to avec's  
    llvec(unsigned int size);  
    unsigned int size() const;  
    ...  
};
```


Funktion `llvec::print()`

```
struct llnode {  
    int value;  
    llnode* next;  
    ...  
};
```

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head; ← Zeiger auf erstes Element  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' ';  
    }  
}
```

Funktion `llvec::print()`

```
struct llnode {  
    int value;  
    llnode* next;  
    ...  
};
```

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr; ←  
         n = n->next)  
    {  
        sink << n->value << ' ';  
    }  
}
```

Abbruch falls Ende erreicht

Funktion `llvec::print()`

```
struct llnode {  
    int value;  
    llnode* next;  
    ...  
};
```

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next) ←  
    {  
        sink << n->value << ' ';  
    }  
}
```

Zeiger elementweise voranschieben

Funktion `llvec::print()`

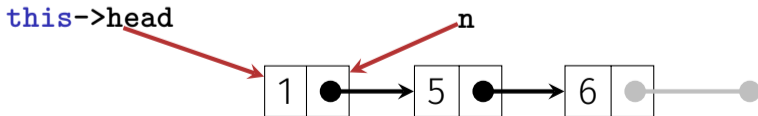
```
struct llnode {  
    int value;  
    llnode* next;  
    ...  
};
```

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' '; ←  
    }  
}
```

Aktuelles Element ausgeben

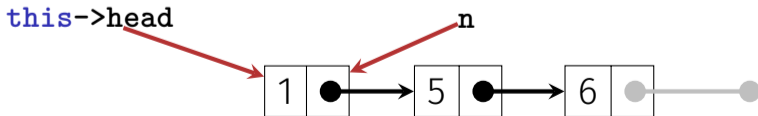
Funktion l1vec::print()

```
void l1vec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' ';  
    }  
}
```



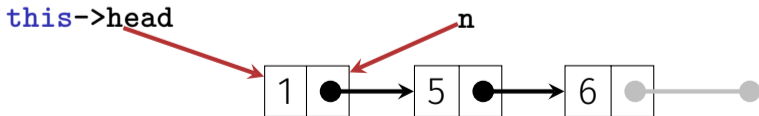
Funktion `llvec::print()`

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' ';  
    }  
}
```



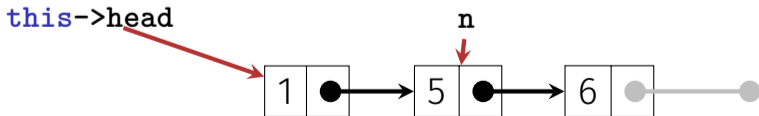
Funktion `llvec::print()`

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' '; // 1  
    }  
}
```



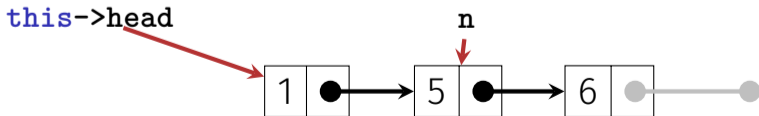
Funktion `llvec::print()`

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' '; // 1  
    }  
}
```



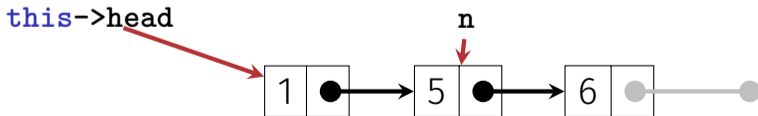
Funktion `llvec::print()`

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' '; // 1  
    }  
}
```



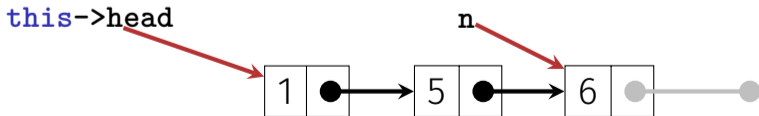
Funktion l1vec::print()

```
void l1vec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' '; // 1 5  
    }  
}
```



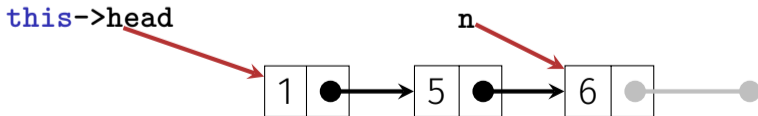
Funktion `llvec::print()`

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' '; // 1 5  
    }  
}
```



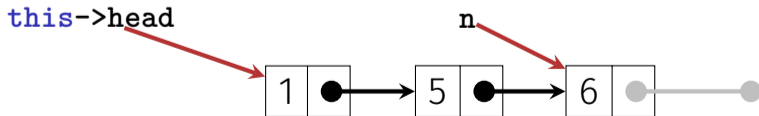
Funktion l1vec::print()

```
void l1vec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' '; // 1 5  
    }  
}
```



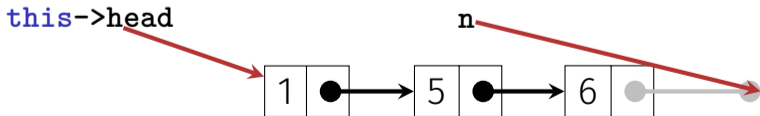
Funktion `llvec::print()`

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' '; // 1 5 6  
    }  
}
```




Funktion `llvec::print()`

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' '; // 1 5 6  
    }  
}
```



Funktion `llvec::operator []`

Zugriff auf i -tes Element ähnlich implementiert wie `print()`:

```
int& llvec::operator [] (unsigned int i) {  
    llnode* n = this->head;  Zeiger auf erstes Element  
  
    for (; 0 < i; --i)  
        n = n->next;  
  
    return n->value;  
}
```

Funktion `llvec::operator []`

Zugriff auf i -tes Element ähnlich implementiert wie `print()`:

```
int& llvec::operator [] (unsigned int i) {  
    llnode* n = this->head;  
  
    for (; 0 < i; --i)  
        n = n->next;  
  
    return n->value;  
}
```



Bis zum i -ten voranschreiten

Funktion `llvec::operator []`

Zugriff auf i -tes Element ähnlich implementiert wie `print()`:

```
int& llvec::operator [] (unsigned int i) {  
    llnode* n = this->head;  
  
    for (; 0 < i; --i)  
        n = n->next;  
  
    return n->value; }  
}
```

← `i`-tes Element zurückgeben

Funktion `llvec::push_front()`

Vorteil `llvec`: Elemente am Anfang anfügen ist sehr einfach:

```
void llvec::push_front(int e) {  
    this->head =  
        new llnode{e, this->head};  
}
```

`this->head`

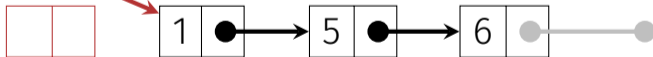


Funktion `llvec::push_front()`

Vorteil `llvec`: Elemente am Anfang anfügen ist sehr einfach:

```
void llvec::push_front(int e) {  
    this->head =  
        new llnode{e, this->head};  
}
```

`this->head`



Funktion `llvec::push_front()`

Vorteil `llvec`: Elemente am Anfang anfügen ist sehr einfach:

```
void llvec::push_front(int e) {  
    this->head =  
        new llnode{e, this->head};  
}
```

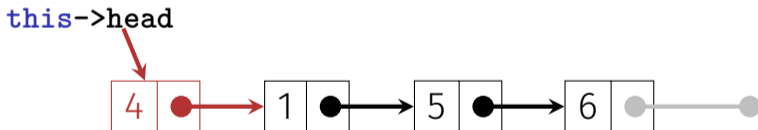
`this->head`



Funktion `llvec::push_front()`

Vorteil `llvec`: Elemente am Anfang anfügen ist sehr einfach:

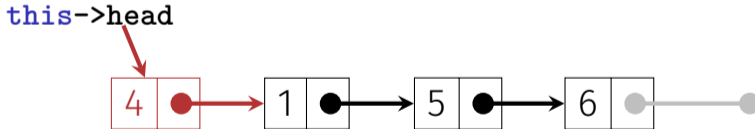
```
void llvec::push_front(int e) {  
    this->head =  
        new llnode{e, this->head};  
}
```



Funktion `llvec::push_front()`

Vorteil `llvec`: Elemente am Anfang anfügen ist sehr einfach:

```
void llvec::push_front(int e) {  
    this->head =  
        new llnode{e, this->head};  
}
```



Achtung: Wäre der neue `llnode` nicht *dynamisch* alloziert, dann würde er am Ende von `push_front` sofort wieder gelöscht (= Speicher dealloziert)

Funktion `llvec::llvec()`

Konstruktor kann mittels `push_front()` implementiert werden:

```
llvec::llvec(unsigned int size) {  
    this->head = nullptr;  
  
    for (; 0 < size; --size)  
        this->push_front(0);  
}
```

← head zeigt zunächst ins Nichts

Funktion `llvec::llvec()`

Konstruktor kann mittels `push_front()` implementiert werden:

```
llvec::llvec(unsigned int size) {  
    this->head = nullptr;  
  
    for (; 0 < size; --size)  
        this->push_front(0);  
}
```

← size mal 0 vorne anfügen

Funktion `llvec::llvec()`

Konstruktor kann mittels `push_front()` implementiert werden:



```
llvec::llvec(unsigned int size) {  
    this->head = nullptr;  
  
    for (; 0 < size; --size)  
        this->push_front(0);  
}
```

Anwendungsbeispiel:

```
llvec v = llvec(3);  
std::cout << v; // 0 0 0
```

Funktion `llvec::push_back()`

Einfach, aber ineffizient: Verkettete Liste bis ans Ende traversieren und neues Element anhängen

```
void llvec::push_back(int e) {  
    llnode* n = this->head;   Beim ersten Element beginnen ...  
  
    for (; n->next != nullptr; n = n->next);  
  
    n->next =  
        new llnode{e, nullptr};  
}
```

Funktion `llvec::push_back()`

Einfach, aber ineffizient: Verkettete Liste bis ans Ende traversieren und neues Element anhängen

```
void llvec::push_back(int e) {  
    llnode* n = this->head;  
  
    for (; n->next != nullptr; n = n->next);  
  
    n->next =  
        new llnode{e, nullptr};  
}
```

... und bis zum letzten Element gehen



Funktion `llvec::push_back()`

Einfach, aber ineffizient: Verkettete Liste bis ans Ende traversieren und neues Element anhängen

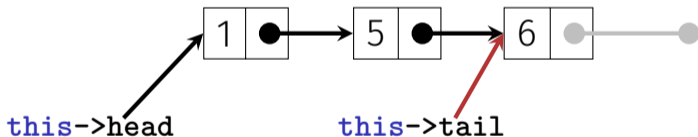
```
void llvec::push_back(int e) {  
    llnode* n = this->head;  
  
    for (; n->next != nullptr; n = n->next);  
  
    n->next =  
        new llnode{e, nullptr};  
}
```

← Neues Element an bisher
letztes anhängen

Funktion `llvec::push_back()`

- Effizienter, aber auch etwas komplexer:

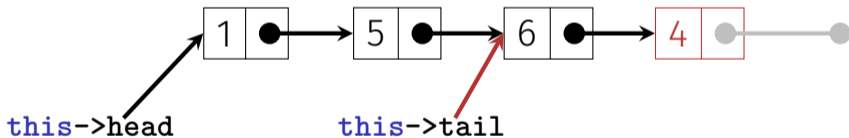
1. Zweiter Zeiger, der auf das letzte Element zeigt: `this->tail`



Funktion `llvec::push_back()`

■ Effizienter, aber auch etwas komplexer:

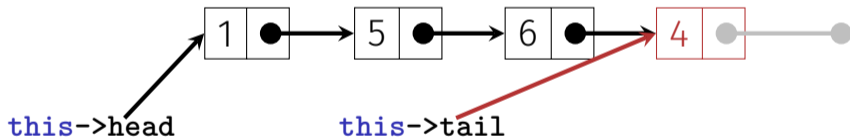
1. Zweiter Zeiger, der auf das letzte Element zeigt: `this->tail`
2. Mittels dieses Zeigers kann direkt am Ende angehängt werden



Funktion `llvec::push_back()`

■ Effizienter, aber auch etwas komplexer:

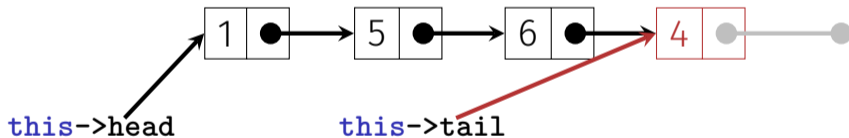
1. Zweiter Zeiger, der auf das letzte Element zeigt: `this->tail`
2. Mittels dieses Zeigers kann direkt am Ende angehängt werden



Funktion `llvec::push_back()`

- Effizienter, aber auch etwas komplexer:

1. Zweiter Zeiger, der auf das letzte Element zeigt: `this->tail`
2. Mittels dieses Zeigers kann direkt am Ende angehängt werden



- **Aber:** Verschiedene Grenzfälle, z.B. Vektor noch leer, müssen beachtet werden

Funktion `llvec::size()`

Einfach, aber ineffizient: Grösse durch abzählen *berechnen*

```
unsigned int llvec::size() const {  
    unsigned int c = 0;  
  
    for ( _____ ;  
         _____ ;  
         _____ )  
    {  
        _____ ;  
    }  
  
    return c ;  
}
```

Funktion `llvec::size()`

Einfach, aber ineffizient: Grösse durch abzählen *berechnen*

```
unsigned int llvec::size() const {  
    unsigned int c = 0; ← Zähler initial 0  
  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
        ++c;  
  
    return c;  
}
```

Funktion `llvec::size()`

Einfach, aber ineffizient: Grösse durch abzählen *berechnen*

```
unsigned int llvec::size() const {  
    unsigned int c = 0;  
  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
        ++c;  
  
    return c;  
}
```



Länge der Kette abzählen

Funktion `llvec::size()`

Einfach, aber ineffizient: Grösse durch abzählen *berechnen*

```
unsigned int llvec::size() const {  
    unsigned int c = 0;  
  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
        ++c;  
  
    return c;   
}
```

← Zähler zurückgeben

Funktion `lvec::size()`

Effizienter, aber etwas komplexer: Grösse als Membervariable *nachhalten*

1. Membervariable `unsigned int count` zur Klasse `lvec` hinzufügen

Funktion `l1vec::size()`

Effizienter, aber etwas komplexer: Grösse als Membervariable *nachhalten*

1. Membervariable `unsigned int count` zur Klasse `l1vec` hinzufügen
2. `this->count` muss nun bei *jeder* Operation, die die Grösse des Vektors verändert (z.B. `push_front`), aktualisiert werden

Effizienz: Arrays vs. Verkettete Listen

- Speicher: Unser **avec** belegt ungefähr n ints (Vektorgröße n), unser **llvec** ungefähr $3n$ ints (ein Zeiger belegt i.d.R. 8 Byte)

Effizienz: Arrays vs. Verkettete Listen

- Speicher: Unser **avec** belegt ungefähr n ints (Vektorgröße n), unser **llvec** ungefähr $3n$ ints (ein Zeiger belegt i.d.R. 8 Byte)
- Laufzeit (mit **avec** = `std::vector`, **llvec** = `std::list`):

```
prepending (insert at front) [100,000x]:
  ▶ avec: 675 ms
  ▶ llvec: 10 ms
appending (insert at back) [100,000x]:
  ▶ avec: 2 ms
  ▶ llvec: 9 ms
removing first [100,000x]:
  ▶ avec: 675 ms
  ▶ llvec: 4 ms
removing last [100,000x]:
  ▶ avec: 0 ms
  ▶ llvec: 4 ms

removing randomly [10,000x]:
  ▶ avec: 3 ms
  ▶ llvec: 113 ms
inserting randomly [10,000x]:
  ▶ avec: 16 ms
  ▶ llvec: 117 ms
fully iterate sequentially (5000 elements) [5,000x]:
  ▶ avec: 354 ms
  ▶ llvec: 525 ms
```


22. Container, Iteratoren und Algorithmen

Container, Mengen, Iteratoren, const-Iteratoren, Algorithmen, Templates

Vektoren sind Container

- Abstrakt gesehen ist ein Vektor
 1. Eine Ansammlung von Elementen
 2. Plus Operationen auf dieser Ansammlung

Vektoren sind Container

- Abstrakt gesehen ist ein Vektor
 1. Eine Ansammlung von Elementen
 2. Plus Operationen auf dieser Ansammlung
- In C++ heissen **vector** $\langle T \rangle$ und ähnliche „Ansammlungs“-Datenstrukturen *Container*

Vektoren sind Container

- Abstrakt gesehen ist ein Vektor
 1. Eine Ansammlung von Elementen
 2. Plus Operationen auf dieser Ansammlung
- In C++ heissen **vector** $\langle T \rangle$ und ähnliche „Ansammlungs“-Datenstrukturen *Container*
- In manchen Sprachen, z.B. Java, *Collections* genannt

Container-Eigenschaften

- Jeder Container hat bestimmte *charakteristische Eigenschaften*
- Ein Array-basierter Vektor z.B. die folgenden:

Container-Eigenschaften

- Jeder Container hat bestimmte *charakteristische Eigenschaften*
- Ein Array-basierter Vektor z.B. die folgenden:
 - Effizienter, index-basierter Zugriff ($v[i]$)
 - Effiziente Speichernutzung: Nur die Elemente selbst belegen Platz (plus Elementezähler)

Container-Eigenschaften

- Jeder Container hat bestimmte *charakteristische Eigenschaften*
- Ein Array-basierter Vektor z.B. die folgenden:
 - Effizienter, index-basierter Zugriff ($v[i]$)
 - Effiziente Speichernutzung: Nur die Elemente selbst belegen Platz (plus Elementezähler)
 - Einfügen/Entfernen an beliebigem Index ist potenziell ineffizient
 - Suchen eines bestimmten Elements ist potenziell ineffizient

Container-Eigenschaften

- Jeder Container hat bestimmte *charakteristische Eigenschaften*
- Ein Array-basierter Vektor z.B. die folgenden:
 - Effizienter, index-basierter Zugriff ($v[i]$)
 - Effiziente Speichernutzung: Nur die Elemente selbst belegen Platz (plus Elementezähler)
 - Einfügen/Entfernen an beliebigem Index ist potenziell ineffizient
 - Suchen eines bestimmten Elements ist potenziell ineffizient
 - Kann Elemente mehrfach enthalten
 - Elemente sind in Einfügereihenfolge enthalten (geordnet aber unsortiert)

Container in C++

- Fast jede Anwendung erfordert die Verwaltung und Manipulation von beliebig vielen Datensätzen

Container in C++

- Fast jede Anwendung erfordert die Verwaltung und Manipulation von beliebig vielen Datensätzen
- Aber mit unterschiedlichen Anforderungen (z.B. Elemente nur hinten anhängen, fast nie entfernen, oft suchen, ...)

Container in C++

- Fast jede Anwendung erfordert die Verwaltung und Manipulation von beliebig vielen Datensätzen
- Aber mit unterschiedlichen Anforderungen (z.B. Elemente nur hinten anhängen, fast nie entfernen, oft suchen, ...)
- Deswegen enthält die Standardbibliothek von C++ diverse Container mit unterschiedlichen Eigenschaften, siehe <https://en.cppreference.com/w/cpp/container>

Container in C++

- Fast jede Anwendung erfordert die Verwaltung und Manipulation von beliebig vielen Datensätzen
- Aber mit unterschiedlichen Anforderungen (z.B. Elemente nur hinten anhängen, fast nie entfernen, oft suchen, ...)
- Deswegen enthält die Standardbibliothek von C++ diverse Container mit unterschiedlichen Eigenschaften, siehe <https://en.cppreference.com/w/cpp/container>
- Viele weitere sind über Bibliotheken Dritter verfügbar, z.B. https://www.boost.org/doc/libs/1_68_0/doc/html/container.html, <https://github.com/abseil/abseil-cpp>

Beispiel-Container:

`std::unordered_set<T>`

- Eine *mathematische Menge* ist eine ungeordnete, duplikatfreie Zusammenfassung von Elementen:

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- In C++: `std::unordered_set<T>`

Beispiel-Container:

`std::unordered_set<T>`

- Eine *mathematische Menge* ist eine ungeordnete, duplikatfreie Zusammenfassung von Elementen:

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- In C++: `std::unordered_set<T>`
- Eigenschaften:
 - Kann kein Element doppelt enthalten
 - Elemente haben keine bestimmte Reihenfolge

Beispiel-Container:

`std::unordered_set<T>`

- Eine *mathematische Menge* ist eine ungeordnete, duplikatfreie Zusammenfassung von Elementen:

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- In C++: `std::unordered_set<T>`
- Eigenschaften:
 - Kann kein Element doppelt enthalten
 - Elemente haben keine bestimmte Reihenfolge
 - Kein indexbasierter Zugriff (`s[i]` nicht definiert)

Beispiel-Container:

`std::unordered_set<T>`

- Eine *mathematische Menge* ist eine ungeordnete, duplikatfreie Zusammenfassung von Elementen:

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- In C++: `std::unordered_set<T>`
- Eigenschaften:
 - Kann kein Element doppelt enthalten
 - Elemente haben keine bestimmte Reihenfolge
 - Kein indexbasierter Zugriff (`s[i]` nicht definiert)
 - Effiziente „Element enthalten?“-Prüfung
 - Effizientes Einfügen und Löschen von Elementen

Beispiel-Container:

`std::unordered_set<T>`

- Eine *mathematische Menge* ist eine ungeordnete, duplikatfreie Zusammenfassung von Elementen:

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- In C++: `std::unordered_set<T>`
- Eigenschaften:
 - Kann kein Element doppelt enthalten
 - Elemente haben keine bestimmte Reihenfolge
 - Kein indexbasierter Zugriff (`s[i]` nicht definiert)
 - Effiziente „Element enthalten?“-Prüfung
 - Effizientes Einfügen und Löschen von Elementen
- Randbemerkung: Implementiert als Hash-Tabelle

Anwendungsbeispiel

`std::unordered_set<T>`

Problem:

- Gegeben eine Sequenz an Paaren (*Name*, *Prozente*) von Code-Expert-Submissions ...

```
// Input: file submissions.txt
Friedrich 90
Schwerhoff 10
Lehner 20
Schwerhoff 11
```

Anwendungsbeispiel

`std::unordered_set<T>`

Problem:

- Gegeben eine Sequenz an Paaren (*Name*, *Prozente*) von Code-Expert-Submissions ...

```
// Input: file submissions.txt
Friedrich 90
Schwerhoff 10
Lehner 20
Schwerhoff 11
```

- ... bestimme die Abgebenden, die mindestens 50% erzielt haben

```
// Output
Friedrich
```

Anwendungsbeispiel

`std::unordered_set<T>`

```
std::ifstream in("submissions.txt"); ← Öffne submissions.txt
std::unordered_set<std::string> names;

std::string name;
unsigned int score;

while (in >> name >> score) {
    if (50 <= score)
        names.insert(name);
}

std::cout << "Unique submitters: "
           << names << '\n';
```

Anwendungsbeispiel

`std::unordered_set<T>`

```
std::ifstream in("submissions.txt");
std::unordered_set<std::string> names; ← Namen-Menge, initial leer

std::string name;
unsigned int score;

while (in >> name >> score) {
    if (50 <= score)
        names.insert(name);
}

std::cout << "Unique submitters: "
          << names << '\n';
```

Anwendungsbeispiel

std::unordered_set<T>

```
std::ifstream in("submissions.txt");  
std::unordered_set<std::string> names;
```

```
std::string name;  
unsigned int score;
```



Paar (Name, Punkte)

```
while (in >> name >> score) {  
    if (50 <= score)  
        names.insert(name);  
}
```

```
std::cout << "Unique submitters: "  
          << names << '\n';
```

Anwendungsbeispiel

`std::unordered_set<T>`

```
std::ifstream in("submissions.txt");  
std::unordered_set<std::string> names;
```

```
std::string name;  
unsigned int score;
```

```
while (in >> name >> score) {  
    if (50 <= score)  
        names.insert(name);  
}
```

Nächstes Paar einlesen



```
std::cout << "Unique submitters: "  
           << names << '\n';
```

Anwendungsbeispiel

`std::unordered_set<T>`

```
std::ifstream in("submissions.txt");  
std::unordered_set<std::string> names;
```

```
std::string name;  
unsigned int score;
```

```
while (in >> name >> score) {  
    if (50 <= score)  
        names.insert(name);  
}
```

```
std::cout << "Unique submitters: "  
          << names << '\n';
```

Namen merken falls
Punkte ausreichen



Anwendungsbeispiel

`std::unordered_set<T>`

```
std::ifstream in("submissions.txt");  
std::unordered_set<std::string> names;
```

```
std::string name;  
unsigned int score;
```

```
while (in >> name >> score) {  
    if (50 <= score)  
        names.insert(name);  
}
```

```
std::cout << "Unique submitters: "  
           << names << '\n';
```

← Gemerkte Namen ausgeben

Beispiel-Container: `std::set<T>`

- Fast gleich wie `std::unordered_set<T>`, aber die Elemente sind *geordnet*

$$\{1, 2, 1\} = \{1, 2\} \neq \{2, 1\}$$

Beispiel-Container: `std::set<T>`

- Fast gleich wie `std::unordered_set<T>`, aber die Elemente sind *geordnet*

$$\{1, 2, 1\} = \{1, 2\} \neq \{2, 1\}$$

- Elemente suchen, einfügen und löschen weiterhin effizient (besser als bei `std::vector<T>`), aber weniger effizient als bei `std::unordered_set<T>`

Beispiel-Container: `std::set<T>`

- Fast gleich wie `std::unordered_set<T>`, aber die Elemente sind *geordnet*

$$\{1, 2, 1\} = \{1, 2\} \neq \{2, 1\}$$

- Elemente suchen, einfügen und löschen weiterhin effizient (besser als bei `std::vector<T>`), aber weniger effizient als bei `std::unordered_set<T>`
- Denn das Beibehalten der Ordnung zieht etwas Aufwand nach sich

Beispiel-Container: `std::set<T>`

- Fast gleich wie `std::unordered_set<T>`, aber die Elemente sind *geordnet*

$$\{1, 2, 1\} = \{1, 2\} \neq \{2, 1\}$$

- Elemente suchen, einfügen und löschen weiterhin effizient (besser als bei `std::vector<T>`), aber weniger effizient als bei `std::unordered_set<T>`
- Denn das Beibehalten der Ordnung zieht etwas Aufwand nach sich
- Randbemerkung: Implementiert als Rot-Schwarz-Baum

Anwendungsbeispiel `std::set<T>`

```
std::ifstream in("submissions.txt");  
std::set<std::string> names;
```

set statt unordered_set ...

```
std::string name;  
unsigned int score;
```

```
while (in >> name >> score) {  
    if (50 <= score)  
        names.insert(name);  
}
```

```
std::cout << "Unique submitters: "  
           << names << '\n';
```

Anwendungsbeispiel `std::set<T>`

```
std::ifstream in("submissions.txt");  
std::set<std::string> names;
```

```
std::string name;  
unsigned int score;
```

```
while (in >> name >> score) {  
    if (50 <= score)  
        names.insert(name);  
}
```

```
std::cout << "Unique submitters: "  
           << names << '\n';
```

... und die Ausgabe erfolgt alphabetisch sortiert

Container Ausgeben

- Bereits gesehen: `avec::print()` und `llvec::print()`

Container Ausgeben

- Bereits gesehen: `avec::print()` und `llvec::print()`
- Wie sieht's mit der Ausgabe von `set`, `unordered_set`, ... aus?

Container Ausgeben

- Bereits gesehen: `avec::print()` und `llvec::print()`
- Wie sieht's mit der Ausgabe von `set`, `unordered_set`, ... aus?
- Gemeinsamkeit: Über Container-Elemente iterieren und diese ausgeben

Ähnliche Funktionen

- Viele weitere nützliche Funktionen können mittels Container-Iteration implementiert werden:
- **contains(c, e)**: wahr gdw. Container **c** Element **e** enthält

Ähnliche Funktionen

- Viele weitere nützliche Funktionen können mittels Container-Iteration implementiert werden:
- **contains(c, e)**: wahr gdw. Container **c** Element **e** enthält
- **min/max(c)**: Gibt das grösste/kleinste Element zurück

Ähnliche Funktionen

- Viele weitere nützliche Funktionen können mittels Container-Iteration implementiert werden:
- **contains(c, e)**: wahr gdw. Container **c** Element **e** enthält
- **min/max(c)**: Gibt das grösste/kleinste Element zurück
- **sort(c)**: Sortiert die Elemente von **c**

Ähnliche Funktionen

- Viele weitere nützliche Funktionen können mittels Container-Iteration implementiert werden:
- **contains(c, e)**: wahr gdw. Container **c** Element **e** enthält
- **min/max(c)**: Gibt das grösste/kleinste Element zurück
- **sort(c)**: Sortiert die Elemente von **c**
- **replace(c, e1, e2)**: Ersetzt alle **e1** in **c** mit **e2**

Ähnliche Funktionen

- Viele weitere nützliche Funktionen können mittels Container-Iteration implementiert werden:
- `contains(c, e)`: wahr gdw. Container `c` Element `e` enthält
- `min/max(c)`: Gibt das grösste/kleinste Element zurück
- `sort(c)`: Sortiert die Elemente von `c`
- `replace(c, e1, e2)`: Ersetzt alle `e1` in `c` mit `e2`
- `sample(c, n)`: Wählt zufällig `n` Elemente aus `c` aus

Ähnliche Funktionen

- Viele weitere nützliche Funktionen können mittels Container-Iteration implementiert werden:
- `contains(c, e)`: wahr gdw. Container `c` Element `e` enthält
- `min/max(c)`: Gibt das grösste/kleinste Element zurück
- `sort(c)`: Sortiert die Elemente von `c`
- `replace(c, e1, e2)`: Ersetzt alle `e1` in `c` mit `e2`
- `sample(c, n)`: Wählt zufällig `n` Elemente aus `c` aus
- ...

Zur Erinnerung: Iterieren mit Zeigern

- Iteration über ein *Array*:



Zur Erinnerung: Iterieren mit Zeigern

- Iteration über ein *Array*:

- Auf Startelement zeigen: `p = this->arr`



Zur Erinnerung: Iterieren mit Zeigern

■ Iteration über ein *Array*:

- Auf Startelement zeigen: `p = this->arr`
- Auf aktuelles Element zugreifen: `*p`



Zur Erinnerung: Iterieren mit Zeigern

■ Iteration über ein *Array*:

- Auf Startelement zeigen: `p = this->arr`
- Auf aktuelles Element zugreifen: `*p`
- Überprüfen, ob Ende erreicht:
`p == this->arr + size`



Zur Erinnerung: Iterieren mit Zeigern

■ Iteration über ein *Array*:

- Auf Startelement zeigen: `p = this->arr`
- Auf aktuelles Element zugreifen: `*p`
- Überprüfen, ob Ende erreicht:
`p == this->arr + size`
- Zeiger vorrücken: `p = p + 1`



Zur Erinnerung: Iterieren mit Zeigern

■ Iteration über ein *Array*:

- Auf Startelement zeigen: `p = this->arr`
- Auf aktuelles Element zugreifen: `*p`
- Überprüfen, ob Ende erreicht:
`p == this->arr + size`
- Zeiger vorrücken: `p = p + 1`



■ Iteration über eine *verkettete Liste*:



Zur Erinnerung: Iterieren mit Zeigern

■ Iteration über ein *Array*:

- Auf Startelement zeigen: `p = this->arr`
- Auf aktuelles Element zugreifen: `*p`
- Überprüfen, ob Ende erreicht:
`p == this->arr + size`
- Zeiger vorrücken: `p = p + 1`



■ Iteration über eine *verkettete Liste*:

- Auf Startelement zeigen: `p = this->head`



Zur Erinnerung: Iterieren mit Zeigern

■ Iteration über ein *Array*:

- Auf Startelement zeigen: `p = this->arr`
- Auf aktuelles Element zugreifen: `*p`
- Überprüfen, ob Ende erreicht:
`p == this->arr + size`
- Zeiger vorrücken: `p = p + 1`



■ Iteration über eine *verkettete Liste*:

- Auf Startelement zeigen: `p = this->head`
- Auf aktuelles Element zugreifen: `p->value`



Zur Erinnerung: Iterieren mit Zeigern

■ Iteration über ein *Array*:

- Auf Startelement zeigen: `p = this->arr`
- Auf aktuelles Element zugreifen: `*p`
- Überprüfen, ob Ende erreicht:
`p == this->arr + size`
- Zeiger vorrücken: `p = p + 1`



■ Iteration über eine *verkettete Liste*:

- Auf Startelement zeigen: `p = this->head`
- Auf aktuelles Element zugreifen: `p->value`
- Überprüfen, ob Ende erreicht:
`p == nullptr`



Zur Erinnerung: Iterieren mit Zeigern

■ Iteration über ein *Array*:

- Auf Startelement zeigen: `p = this->arr`
- Auf aktuelles Element zugreifen: `*p`
- Überprüfen, ob Ende erreicht:
`p == this->arr + size`
- Zeiger vorrücken: `p = p + 1`



■ Iteration über eine *verkettete Liste*:

- Auf Startelement zeigen: `p = this->head`
- Auf aktuelles Element zugreifen: `p->value`
- Überprüfen, ob Ende erreicht:
`p == nullptr`
- Zeiger vorrücken: `p = p->next`



Iteratoren

- Iteration erfordert nur die vier eben gesehenen Operationen
- Aber deren Implementierung hängt vom Container ab

Iteratoren

- Iteration erfordert nur die vier eben gesehenen Operationen
- Aber deren Implementierung hängt vom Container ab
- \Rightarrow Jeder C++-Container implementiert seinen eigenen *Iterator*

Iteratoren

- Iteration erfordert nur die vier eben gesehenen Operationen
- Aber deren Implementierung hängt vom Container ab
- \Rightarrow Jeder C++-Container implementiert seinen eigenen *Iterator*
- Gegeben ein Container `c`:
 - `it = c.begin()`: Iterator aufs erste Element

Iteratoren

- Iteration erfordert nur die vier eben gesehenen Operationen
- Aber deren Implementierung hängt vom Container ab
- \Rightarrow Jeder C++-Container implementiert seinen eigenen *Iterator*
- Gegeben ein Container `c`:
 - `it = c.begin()`: Iterator aufs erste Element
 - `it = c.end()`: Iterator *hinters* letzte Element

Iteratoren

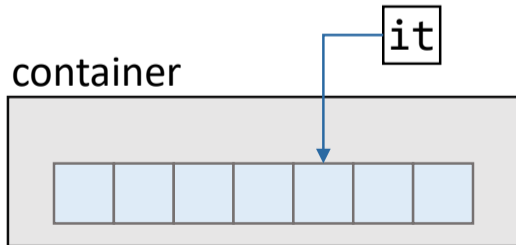
- Iteration erfordert nur die vier eben gesehenen Operationen
- Aber deren Implementierung hängt vom Container ab
- \Rightarrow Jeder C++-Container implementiert seinen eigenen *Iterator*
- Gegeben ein Container `c`:
 - `it = c.begin()`: Iterator aufs erste Element
 - `it = c.end()`: Iterator *hinters* letzte Element
 - `*it`: Zugriff aufs aktuelle Element

Iteratoren

- Iteration erfordert nur die vier eben gesehenen Operationen
- Aber deren Implementierung hängt vom Container ab
- \Rightarrow Jeder C++-Container implementiert seinen eigenen *Iterator*
- Gegeben ein Container `c`:
 - `it = c.begin()`: Iterator aufs erste Element
 - `it = c.end()`: Iterator *hinters* letzte Element
 - `*it`: Zugriff aufs aktuelle Element
 - `++it`: Iterator um ein Element verschieben
- Iteratoren sind quasi gepimpte Zeiger

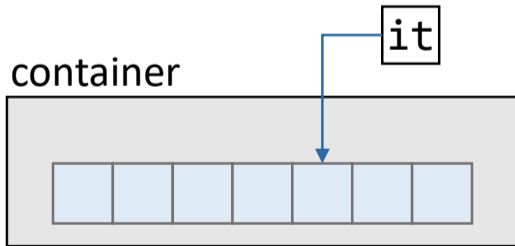
Iteratoren

- Iteratoren ermöglichen Zugriff auf verschiedene Container auf *uniforme* Weise: `*it`, `++it`, etc.



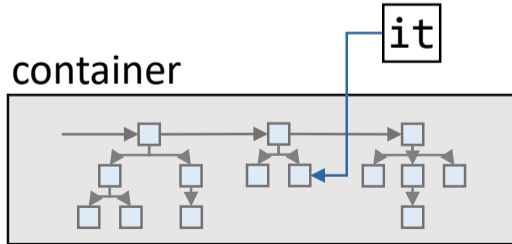
Iteratoren

- Iteratoren ermöglichen Zugriff auf verschiedene Container auf *uniforme* Weise: `*it`, `++it`, etc.
- Nutzer bleiben unabhängig von der Container-Implementierung



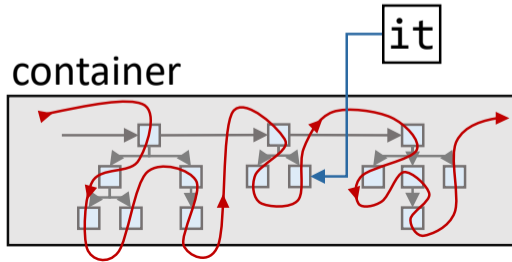
Iteratoren

- Iteratoren ermöglichen Zugriff auf verschiedene Container auf *uniforme* Weise: `*it`, `++it`, etc.
- Nutzer bleiben unabhängig von der Container-Implementierung



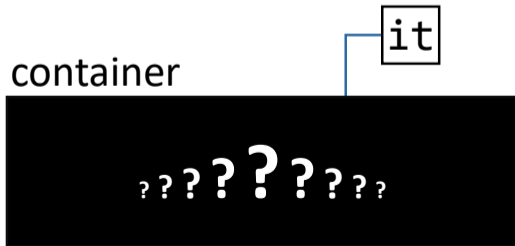
Iteratoren

- Iteratoren ermöglichen Zugriff auf verschiedene Container auf *uniforme* Weise: `*it`, `++it`, etc.
- Nutzer bleiben unabhängig von der Container-Implementierung
- Iterator weiss, wie man die Elemente „seines“ Containers abläuft



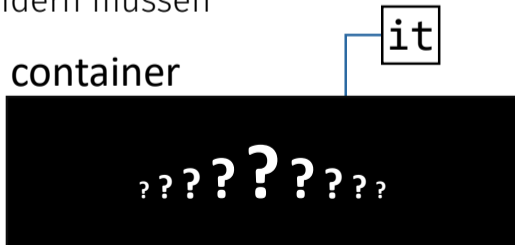
Iteratoren

- Iteratoren ermöglichen Zugriff auf verschiedene Container auf *uniforme* Weise: `*it`, `++it`, etc.
- Nutzer bleiben unabhängig von der Container-Implementierung
- Iterator weiss, wie man die Elemente „seines“ Containers abläuft
- Nutzer müssen und sollen interne Details nicht kennen



Iteratoren

- Iteratoren ermöglichen Zugriff auf verschiedene Container auf *uniforme* Weise: `*it`, `++it`, etc.
- Nutzer bleiben unabhängig von der Container-Implementierung
- Iterator weiss, wie man die Elemente „seines“ Containers abläuft
- Nutzer müssen und sollen interne Details nicht kennen
- ⇒ Containerimplementierung kann geändert werden, ohne das Nutzer Code ändern müssen



Beispiel: Iteration über `std::vector`

`it` ist ein zu `std::vector<int>` passender Iterator

```
std::vector<int> v = {1, 2, 3};  
  
for (std::vector<int>::iterator it = v.begin();  
     it != v.end();  
     ++it) {  
  
    *it = -*it;  
}  
  
std::cout << v; // -1 -2 -3
```

Beispiel: Iteration über `std::vector`


```
std::vector<int> v = {1, 2, 3};
```

it zeigt initial aufs erste Element

```
for (std::vector<int>::iterator it = v.begin(),  
     it != v.end();  
     ++it) {  
  
    *it = -*it;  
}
```

```
std::cout << v; // -1 -2 -3
```


Beispiel: Iteration über `std::vector`

```
std::vector<int> v = {1, 2, 3};  
  
for (std::vector<int>::iterator it = v.begin();  
     it != v.end();  Abbruch falls it Ende erreicht hat  
     ++it) {  
  
    *it = -*it;  
}  
  
std::cout << v; // -1 -2 -3
```

Beispiel: Iteration über `std::vector`

```
std::vector<int> v = {1, 2, 3};  
  
for (std::vector<int>::iterator it = v.begin();  
     it != v.end();  
     ++it) ← { it elementweise vorwärtssetzen  
  
    *it = -*it;  
}  
  
std::cout << v; // -1 -2 -3
```

Beispiel: Iteration über `std::vector`

```
std::vector<int> v = {1, 2, 3};  
  
for (std::vector<int>::iterator it = v.begin();  
     it != v.end();  
     ++it) {  
  
    *it = -*it; ← Aktuelles Element negieren ( $e \rightarrow -e$ )  
}  
  
std::cout << v; // -1 -2 -3
```

Beispiel: Iteration über `std::vector`

```
std::vector<int> v = {1, 2, 3};

for (std::vector<int>::iterator it = v.begin();
     it != v.end();
     ++it) {

    *it = -*it;
}

std::cout << v; // -1 -2 -3
```

Beispiel: Iteration über `std::vector`

Zur Erinnerung: Type-Aliasse können genutzt werden um oft genutzte Typnamen abzukürzen

```
using ivit = std::vector<int>::iterator; // int-vector iterator  
  
for (ivit it = v.begin();  
    ...
```

Negieren als Funktion

Wie zuvor: Übergabe eines *Arbeitsbereichs* (-intervalls)

```
void neg(std::vector<int>::iterator begin;  
        std::vector<int>::iterator end) {  
  
    for (std::vector<int>::iterator it = begin;  
         it != end;  
         ++it) {  
  
        *it = -*it;  
    }  
}
```

← Elemente im Intervall [begin, end) negieren

Negieren als Funktion

Wie zuvor: Übergabe eines *Arbeitsbereichs* (-intervalls)

```
void neg(std::vector<int>::iterator begin;  
        std::vector<int>::iterator end);
```

```
// in main():
```

```
std::vector<int> v = {1, 2, 3};
```

```
neg(v.begin(), v.begin() + (v.size() / 2));
```

← Erste Hälfte negieren

Algorithmen-Bibliothek in C++

- Die C++-Standardbibliothek enthält viele nützliche Algorithmen (Funktionen), die auf durch Iteratoren bestimmten Intervallen [*Anfang*, *Ende*) arbeiten
- Zum Beispiel **find**, **fill** and **sort**; siehe auch <https://en.cppreference.com/w/cpp/algorithm>

Algorithmen-Bibliothek in C++

- Die C++-Standardbibliothek enthält viele nützliche Algorithmen (Funktionen), die auf durch Iteratoren bestimmten Intervallen [*Anfang*, *Ende*) arbeiten
- Zum Beispiel **find**, **fill** and **sort**; siehe auch <https://en.cppreference.com/w/cpp/algorithm>
- Dank Iteratoren können diese ≥ 100 (!) Algorithmen auf beliebigen* Containern ausgeführt werden: Den 17 (!) Standardcontainern von C++, auf unserem **avec** und **llvec** (kommt gleich), etc.

Algorithmen-Bibliothek in C++

- Die C++-Standardbibliothek enthält viele nützliche Algorithmen (Funktionen), die auf durch Iteratoren bestimmten Intervallen [*Anfang*, *Ende*) arbeiten
- Zum Beispiel **find**, **fill** and **sort**; siehe auch <https://en.cppreference.com/w/cpp/algorithm>
- Dank Iteratoren können diese ≥ 100 (!) Algorithmen auf beliebigen* Containern ausgeführt werden: Den 17 (!) Standardcontainern von C++, auf unserem **avec** und **llvec** (kommt gleich), etc.
- Gäbe es diesen uniformen Zugriff auf Container-elemente nicht, müsste *sehr* viel Code dupliziert werden

Ein Iterator für `llvec`

Wir brauchen:

1. Einen `llvec`-spezifischen Iterator mit mindestens folgender Funktionalität:
 - Zugriff aktuelles Element: `operator*`
 - Iterator vorwärtssetzen: `operator++`
 - Ende-Erreicht-Prüfung: `operator!=` (oder `operator==`)

Ein Iterator für `llvec`

Wir brauchen:

1. Einen `llvec`-spezifischen Iterator mit mindestens folgender Funktionalität:
 - Zugriff aktuelles Element: `operator*`
 - Iterator vorwärtssetzen: `operator++`
 - Ende-Erreicht-Prüfung: `operator!=` (oder `operator==`)
2. Memberfunktionen `begin()` und `end()` für `llvec` um einen Iterator auf den Anfang bzw. hinter das Ende zu erhalten

Iterator `l1vec::iterator` (Schritt 1/2)

```
class l1vec {  
    ...  
public:  
    class iterator {  
        ...  
    };  
  
    ...  
}
```


- Der Iterator gehört zu unserem Vektor, daher ist `iterator` eine öffentliche *innere Klasse* von `l1vec`

Iterator `l1vec::iterator` (Schritt 1/2)

```
class l1vec {  
    ...  
public:  
    class iterator {  
        ...  
    };  
  
    ...  
}
```

- Der Iterator gehört zu unserem Vektor, daher ist `iterator` eine öffentliche *innere Klasse* von `l1vec`
- Instanzen unseres Iterators sind vom Typ `l1vec::iterator`

Iterator `llvec::iterator` (Schritt 1/2)

```
class iterator {  
    llnode* node;    
  
public:  
    iterator(llnode* n);  
    iterator& operator++();  
    int& operator*() const;  
    bool operator!=(const iterator& other) const;  
};
```

Zeiger auf aktuelles Vektor-Element

Iterator `llvec::iterator` (Schritt 1/2)

```
class iterator {
    llnode* node;

public:
    iterator(llnode* n);
    iterator& operator++();
    int& operator*() const;
    bool operator!=(const iterator& other) const;
};
```

← Erzeuge Iterator auf bestimmtes Element

Iterator `llvec::iterator` (Schritt 1/2)

```
class iterator {  
    llnode* node;  
  
public:  
    iterator(llnode* n);  
    iterator& operator++(); ←  
    int& operator*() const;  
    bool operator!=(const iterator& other) const;  
};
```

Iterator ein Element vorwärtssetzen

Iterator `llvec::iterator` (Schritt 1/2)

```
class iterator {
    llnode* node;

public:
    iterator(llnode* n);
    iterator& operator++();
    int& operator*() const; ← Zugriff auf aktuelles Element
    bool operator!=(const iterator& other) const;
};
```

Iterator `llvec::iterator` (Schritt 1/2)

```
class iterator {  
    llnode* node;  
  
public:  
    iterator(llnode* n);  
    iterator& operator++();  
    int& operator*() const;  
    bool operator!=(const iterator& other) const;  
};
```

Vergleich mit anderem Iterator



Iterator `llvec::iterator` (Schritt 1/2)

```
// Constructor
llvec::iterator::iterator(llnode* n): node(n) {}

// Pre-increment
llvec::iterator& llvec::iterator::operator++() {
    assert(this->node != nullptr);

    this->node = this->node->next;

    return *this;
}
```

Iterator `llvec::iterator` (Schritt 1/2)

```
// Constructor
```

```
llvec::iterator::iterator(llnode* n): node(n) ← {}
```

Iterator initial auf `n` zeigen lassen

```
// Pre-increment
```

```
llvec::iterator& llvec::iterator::operator++() {  
    assert(this->node != nullptr);
```

```
    this->node = this->node->next;
```

```
    return *this;
```

```
}
```

Iterator `llvec::iterator` (Schritt 1/2)

```
// Constructor
llvec::iterator::iterator(llnode* n): node(n) {}

// Pre-increment
llvec::iterator& llvec::iterator::operator++() {
    assert(this->node != nullptr);

    this->node = this->node->next;

    return *this;
}
```

Iterator `llvec::iterator` (Schritt 1/2)

```
// Constructor  
llvec::iterator::iterator(llnode* n): node(n) {}
```

```
// Pre-increment
```

```
llvec::iterator& llvec::iterator::operator++() {  
    assert(this->node != nullptr);
```

```
    this->node = this->node->next; ← Iterator ein Element vorwärtssetzen
```

```
    return *this;
```

```
}
```

Iterator `llvec::iterator` (Schritt 1/2)

```
// Constructor
llvec::iterator::iterator(llnode* n): node(n) {}

// Pre-increment
llvec::iterator& llvec::iterator::operator++() {
    assert(this->node != nullptr);

    this->node = this->node->next;

    return *this; ← Referenz auf verschobenen Iterator zurückgeben
}
```


Iterator `llvec::iterator` (Schritt 1/2)

```
// Element access
int& llvec::iterator::operator*() const {
    return this->node->value;
}

// Comparison: when are two iterators not equal?
bool llvec::iterator::operator!=(
    const llvec::iterator& other) const
{
    return this->node != other.node;
}
```

Iterator `llvec::iterator` (Schritt 1/2)

```
// Element access
int& llvec::iterator::operator*() const {
    return this->node->value; ← Zugriff auf aktuelles Element
}

// Comparison: when are two iterators not equal?
bool llvec::iterator::operator!=(
    const llvec::iterator& other) const
{
    return this->node != other.node;
}
```

Iterator `llvec::iterator` (Schritt 1/2)

```
// Element access
int& llvec::iterator::operator*() const {
    return this->node->value;
}

// Comparison: when are two iterators not equal?
bool llvec::iterator::operator!=(
    const llvec::iterator& other) const
{
    return this->node != other.node;
}
```

Iterator `llvec::iterator` (Schritt 1/2)

```
// Element access
int& llvec::iterator::operator*() const {
    return this->node->value;
}

// Comparison: when are two iterators not equal?
bool llvec::iterator::operator!=(
    const llvec::iterator& other) const
{
    return this->node != other.node; ←
```

this Iterator verschieden von **other** falls sie auf unterschiedliche Elemente zeigen

Ein Iterator für `llvec` (Wiederholung)

Wir brauchen:

1. Einen `llvec`-spezifischen Iterator mit mindestens folgender Funktionalität:

- Zugriff aktuelles Element: `operator*`
- Iterator vorwärtssetzen: `operator++`
- Ende-Erreicht-Prüfung: `operator!=` (oder `operator==`)



2. Memberfunktionen `begin()` und `end()` für `llvec` um einen Iterator auf den Anfang bzw. hinter das Ende zu erhalten

Iterator `llvec::iterator` (Schritt 2/2)

```
class llvec {  
    ...  
public:  
    class iterator {...};  
  
    iterator begin();  
    iterator end();  
  
    ...  
}
```

`llvec` braucht Memberfunktionen um Iteratoren *auf den Anfang* bzw. *hinter das Ende* des Vektors herausgeben zu können

Iterator `llvec::iterator` (Schritt 2/2)

```
llvec::iterator llvec::begin() {  
    return llvec::iterator(this->head);  
}
```

Iterator auf erstes Vektorelement



```
llvec::iterator llvec::end() {  
    return llvec::iterator(nullptr);  
}
```

Iterator `llvec::iterator` (Schritt 2/2)

```
llvec::iterator llvec::begin() {  
    return llvec::iterator(this->head);  
}
```

```
llvec::iterator llvec::end() {  
    return llvec::iterator(nullptr);  
}
```



Iterator hinter letztes Vektorelement

Typ-generischer Container

Typ-spezifischer Container



Typ-generischer Container



https://upload.wikimedia.org/wikipedia/commons/d/df/Container_01_KMJ.jpg (CC BY-SA 3.0)

(Typ-generische Container (allgemein. Templates) sind nicht klausurrelevant)

Typ-generischer Container

Klasse `cell`: Ein einfacher, 1-elementiger Container für `int`

```
class cell {  
    int element;  
  
public:  
    cell(int e);  
    int& value();  
};
```



Container-
Element

```
cell::cell(int e)  
    : element(e) {}  
  
int& cell::value() {  
    return this->element;  
}
```

Typ-generischer Container

Klasse `cell`: Ein einfacher, 1-elementiger Container für `int`

```
class cell {  
    int element;  
  
public:  
    cell(int e);  
    int& value();  
};
```

Konstruktor speichert `e` im Container

```
cell::cell(int e)  
    : element(e) {}  
  
int& cell::value() {  
    return this->element;  
}
```

Typ-generischer Container

Klasse `cell`: Ein einfacher, 1-elementiger Container für `int`

```
class cell {
    int element;

public:
    cell(int e);
    int& value();
};
```

```
cell::cell(int e)
    : element(e) {}

int& cell::value() {
    return this->element;
}
```

Zugriff aufs Element

Typ-generischer Container

Klasse `cell`: Ein einfacher, 1-elementiger Container für `int`

```
class cell {  
    int element;  
  
public:  
    cell(int e);  
    int& value();  
};
```

```
cell::cell(int e)  
    : element(e) {}  
  
int& cell::value() {  
    return this->element;  
}
```

Besser: Generische `cell<E>` für jeden Elementtyp `E` (analog zu `std::vector<E>`)

Typ-generischer Container mit Templates

Templates ermöglichen *Typ-generische* Funktionen und Klassen:

```
template<typename E> ← Sei E ein beliebiger Typ ...  
class cell {  
    E element;  
  
public:  
    cell(E e);  
    E& value();  
};
```

- Typen können als *Parameter* genutzt werden

Typ-generischer Container mit Templates

Templates ermöglichen *Typ-generische* Funktionen und Klassen:

```
template<typename E> ← Sei E ein beliebiger Typ ...  
class cell {  
    E element;  
  
public:  
    cell(E e);  
    E& value();  
};
```

← ...dann verwaltet `cell` ein Element vom Typ `E`

- Typen können als *Parameter* genutzt werden
- Typ-Parameter sind im „templatierten“ Gültigkeitsbereich

Typ-generischer Container mit Templates

- Signaturen und Implementierungen müssen „templatiert“ werden
- Bei separaten Implementierungen muss das Klassen-Präfix in generischer Form angegeben werden

```
template<typename E>
class cell {
    E element;

public:
    cell(E e);
    E& value();
};
```

```
template<typename E>
cell<E>::cell(E e)
    : element(e) {}

template<typename E>
E& cell<E>::value() {
    return this->element;
}
```


Typ-generischer Container mit Templates

```
cell<int> c1(313);  
cell<std::string> c2("terrific!")
```

- Typ-Parameter müssen bei *Deklarationen*, z.B. `cell<int>`, explizit angegeben werden ...
- ...aber überall sonst werden sie vom Compiler *inferriert*, z.B. bei `c1(313)`, d.h. beim Aufruf des generischen Konstruktors `cell(E e)` (wobei Typparameter **E** vom Compiler mit `int` instanziiert wird)

Templates: Abschluss

- Templates realisieren in C++ *statische Codegenerierung* bzw. *statische Metaprogrammierung*
- Template-Code wird pro Typ-Instanziierung *kopiert*. Bei Benutzung von `cell<int>` und `cell<std::string>` legt der Compiler zwei *instanziierte Kopien* des `cell`-Codes an: sozusagen die beiden (nicht mehr generischen) Klassen `cell_int` und `cell_stdstring`.
- Templates reduzieren Codeduplikation und fördern Code-Wiederverwendbarkeit
- Leider sind Compiler-Fehlermeldungen, die sich auf Templates beziehen, oft noch komplexer, als es C++-Fehlermeldungen ohnehin schon oft sind

23. Dynamische Datentypen und Speicherverwaltung

Problem

Letzte Woche: Dynamischer Datentyp

Haben im Vektor dynamischen Speicher angelegt, aber nicht wieder freigegeben. Insbesondere: keine Funktionen zum Entfernen von Elementen aus **llvec**.

Heute: Korrektes Speichermanagement!

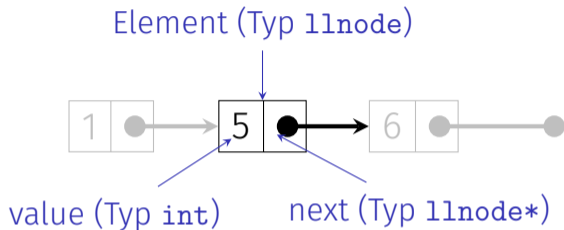
Ziel: Stapelklasse mit Speichermanagement

```
class stack{
public:
    // post: Element auf den Stapel legen
    void push(int value);
    // pre: Stack nicht leer
    // post: Entfernt oberstes Element vom Stapel
    void pop();
    // pre: Stack nicht leer
    // post: Gibt Wert des obersten Elementes zurück
    int top() const;
    // post: gibt zurück, ob Stack leer ist
    bool empty() const;
    // post: gibt den Stapel aus
    void print(std::ostream& out) const;

```

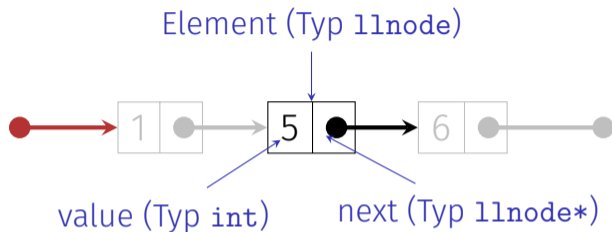
...

Erinnerung: Verkettete Liste



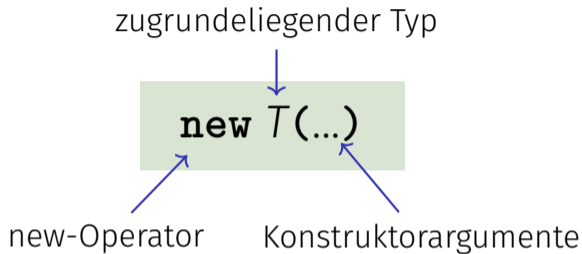
```
struct llnode {  
    int value;  
    llnode* next;  
    // constructor  
    llnode (int v, llnode* n) : value (v), next (n) {}  
};
```

Stapel = Zeiger aufs oberste Element



```
class stack {  
public:  
    void push (int value);  
    ...  
private:  
    llnode* topn;  
};
```

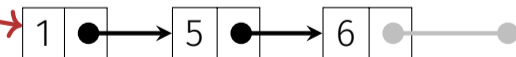
Erinnerung: der `new`-Ausdruck



- **Effekt:** Speicher für ein neues Objekt vom Typ T wird alloziert ...
- ... und mit Hilfe des passenden Konstruktors initialisiert
- **Wert:** Adresse des neuen T -Objekt, **Typ:** Zeiger \mathbf{T}^* !


```
void stack::push(int value) {  
    topn = new llnode(value, topn);  
}
```

topn



- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt

...

```
void stack::push(int value) {  
    topn = new llnode(value, topn);  
}
```

topn



- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt
...
- ...und mit Hilfe des passenden Konstruktors initialisiert.

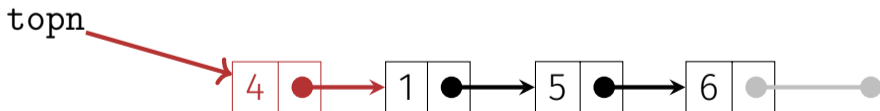
```
void stack::push(int value) {  
    topn = new llnode(value, topn);  
}
```

topn



- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt
...
- ... und mit Hilfe des passenden Konstruktors initialisiert.
- **Wert:** Adresse des neuen Objekts

```
void stack::push(int value) {  
    topn = new llnode(value, topn);  
}
```

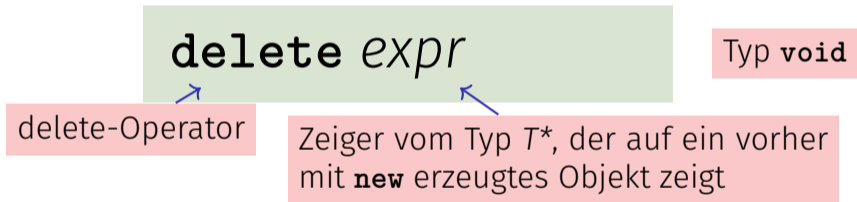


Der delete-Ausdruck

Objekte, die mit **new** erzeugt worden sind, haben **dynamische Speicherdauer**: sie “leben”, bis sie explizit *gelöscht* werden.

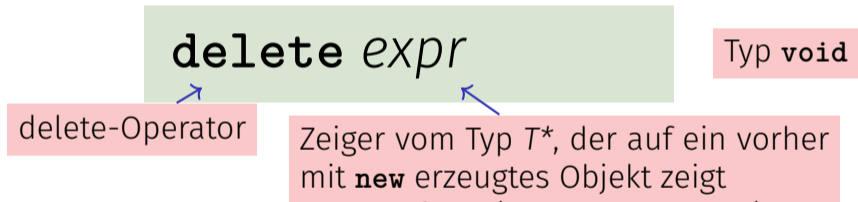
Der delete-Ausdruck

Objekte, die mit **new** erzeugt worden sind, haben **dynamische Speicherdauer**: sie “leben”, bis sie explizit *gelöscht* werden.



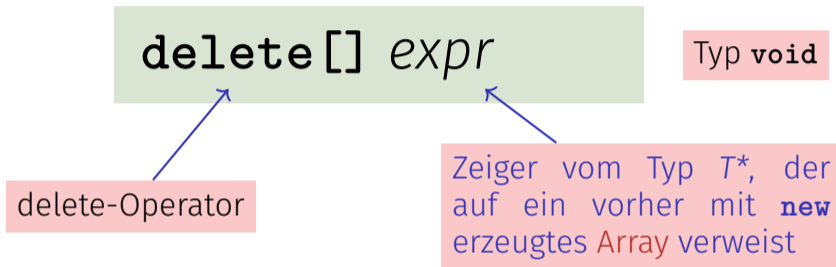
Der delete-Ausdruck

Objekte, die mit **new** erzeugt worden sind, haben **dynamische Speicherdauer**: sie “leben”, bis sie explizit *gelöscht* werden.



- **Effekt:** Objekt wird **dekonstruiert** (Erklärung folgt)
... und **Speicher wird freigegeben**.

Der delete-Ausdruck für Arrays



- **Effekt:** Array wird gelöscht, Speicher wird wieder freigegeben

Wer geboren wird, muss sterben...

Richtlinie “Dynamischer Speicher”

Zu jedem **new** gibt es ein passendes **delete**!

Wer geboren wird, muss sterben...

Richtlinie “Dynamischer Speicher”

Zu jedem **new** gibt es ein passendes **delete**!

Nichtbeachtung führt zu *Speicherlecks*:

Wer geboren wird, muss sterben...

Richtlinie “Dynamischer Speicher”

Zu jedem **new** gibt es ein passendes **delete**!

Nichtbeachtung führt zu *Speicherlecks*:

- “Alte” Objekte, die den Speicher blockieren...

Wer geboren wird, muss sterben...

Richtlinie "Dynamischer Speicher"

Zu jedem **new** gibt es ein passendes **delete**!

Nichtbeachtung führt zu *Speicherlecks*:

- "Alte" Objekte, die den Speicher blockieren...
- ...bis er irgendwann voll ist (**heap overflow**)

Aufpassen mit `new` und `delete`!

```
rational* t = new rational;  
rational* s = t;  
delete s;  
int nominator = (*t).denominator();
```

Aufpassen mit `new` und `delete`!

```
rational* t = new rational;
rational* s = t;
delete s;
int nominator = (*t).denominator();
```

← Speicher für t wird angelegt

Aufpassen mit `new` und `delete`!

```
rational* t = new rational; ← Speicher für t wird angelegt  
rational* s = t; ← Auch andere Zeiger können auf das Objekt zeigen..  
delete s;  
int nominator = (*t).denominator();
```

Aufpassen mit `new` und `delete`!

```
rational* t = new rational; ← Speicher für t wird angelegt  
rational* s = t; ← Auch andere Zeiger können auf das Objekt zeigen..  
delete s; ← ... und zur Freigabe verwendet werden.  
int nominator = (*t).denominator();
```


Aufpassen mit `new` und `delete`!

```
rational* t = new rational; ← Speicher für t wird angelegt  
rational* s = t; ← Auch andere Zeiger können auf das Objekt zeigen..  
delete s; ← ... und zur Freigabe verwendet werden.  
int nominator = (*t).denominator(); ← Fehler: Speicher freigegeben!
```

Dereferenzieren eines „dangling pointers“

- Zeiger auf freigegebene Objekte: hängende Zeiger (*dangling pointers*)

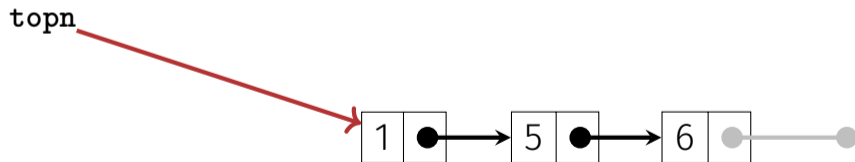
Aufpassen mit `new` und `delete`!

```
rational* t = new rational; ← Speicher für t wird angelegt  
rational* s = t; ← Auch andere Zeiger können auf das Objekt zeigen..  
delete s; ← ... und zur Freigabe verwendet werden.  
int nominator = (*t).denominator(); ← Fehler: Speicher freigegeben!
```

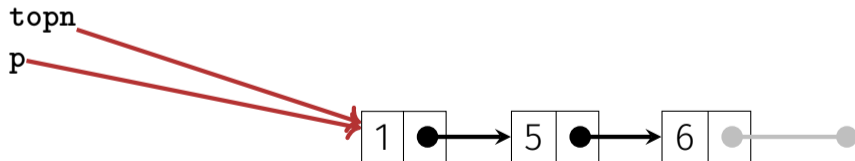
Dereferenzieren eines „dangling pointers“

- Zeiger auf freigegebene Objekte: hängende Zeiger (*dangling pointers*)
- Mehrfache Freigabe eines Objektes mit **delete** ist ein ähnlicher schwerer Fehler.

```
void stack::pop(){  
    assert (!empty());  
    llnode* p = topn;  
    topn = topn->next;  
    delete p;  
}
```



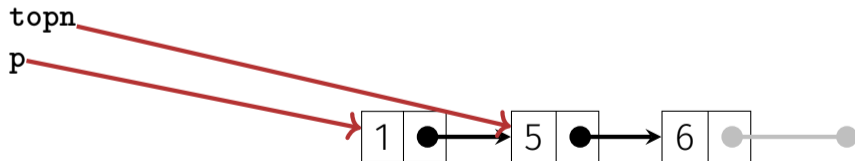
```
void stack::pop(){  
    assert (!empty());  
    llnode* p = topn;  
    topn = topn->next;  
    delete p;  
}
```



Weiter mit dem Stapel:

pop()

```
void stack::pop(){  
    assert (!empty());  
    llnode* p = topn;  
    topn = topn->next;  
    delete p;  
}
```



```
void stack::pop(){  
    assert (!empty());  
    llnode* p = topn;  
    topn = topn->next;  
    delete p;  
}
```

Erinnerung: Abkürzung für `(*topn).next`

topn

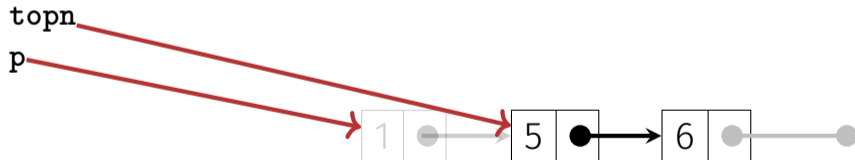
p



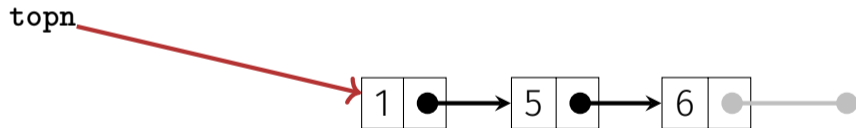
Weiter mit dem Stapel:

pop()

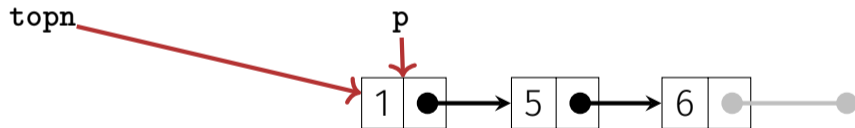
```
void stack::pop(){  
    assert (!empty());  
    llnode* p = topn;  
    topn = topn->next;  
    delete p;  
}
```



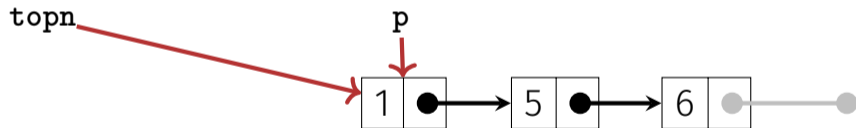
```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " ";  
}
```



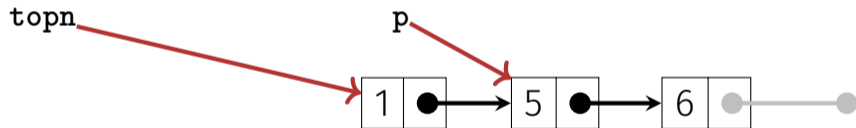

```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " ";  
}
```



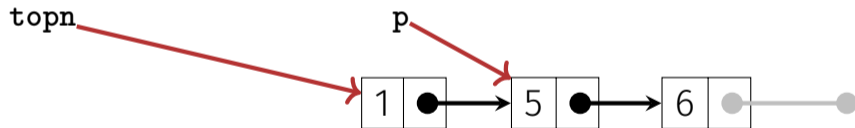
```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " "; // 1  
}
```



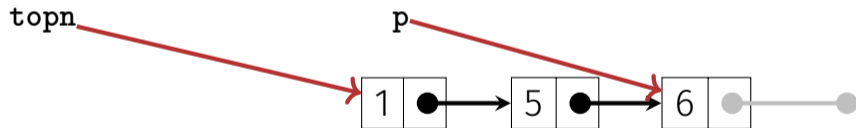
```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " "; // 1  
}
```



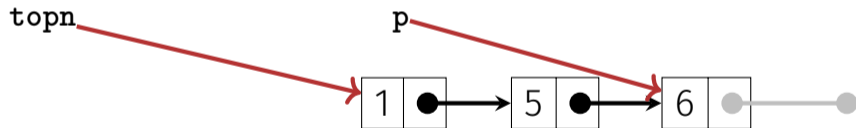
```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " "; // 1 5  
}
```



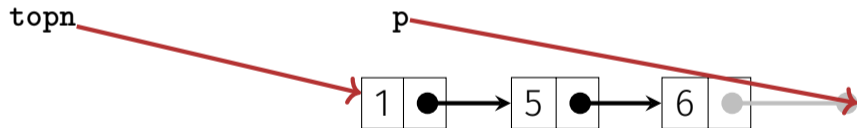
```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " "; // 1 5  
}
```



```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " "; // 1 5 6  
}
```



```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " "; // 1 5 6  
}
```



```
class stack {
public:
    void push (int value);
    void pop();
    void print (std::ostream& o) const;
    ...
private:
    llnode* topn;
};

// POST: s is written to o
std::ostream& operator<< (std::ostream& o, const stack& s){
    s.print (o);
    return o;
}
```


empty(), top()

```
bool stack::empty() const {  
    return top == nullptr;  
}
```

```
int stack::top() const {  
    assert(!empty());  
    return topn->value;  
}
```

Leerer Stapel

```
class stack{
public:
    stack() : topn (nullptr) {} // default constructor

    void push(int value);
    void pop();
    void print(std::ostream& out) const;
    int top() const;
    bool empty() const;
private:
    llnode* topn;
}
```

Zombie-Elemente

```
{  
    stack s1; // lokale Variable  
    s1.push (1);  
    s1.push (3);  
    s1.push (2);  
    std::cout << s1 << "\n"; // 2 3 1  
}  
// s1 ist gestorben (nicht mehr zugreifbar)
```

Zombie-Elemente

```
{  
  stack s1; // lokale Variable  
  s1.push (1);  
  s1.push (3);  
  s1.push (2);  
  std::cout << s1 << "\n"; // 2 3 1  
}  
// s1 ist gestorben (nicht mehr zugreifbar)
```

- ...aber die drei *Elemente* des Stapels `s1` leben weiter (Speicherleck)!

Zombie-Elemente

```
{  
  stack s1; // lokale Variable  
  s1.push (1);  
  s1.push (3);  
  s1.push (2);  
  std::cout << s1 << "\n"; // 2 3 1  
}  
// s1 ist gestorben (nicht mehr zugreifbar)
```

- ...aber die drei *Elemente* des Stapels `s1` leben weiter (Speicherleck)!
- Sie sollten zusammen mit `s1` aufgeräumt werden!

Der Destruktor

- Der Destruktor einer Klasse T ist die eindeutige Memberfunktion mit Deklaration

$$\sim T ();$$

- Wird automatisch aufgerufen, wenn die Speicherdauer eines Klassenobjekts vom Typ \mathbf{T} endet – z.B. bei Aufruf von **delete** auf einem Objekt vom Typ \mathbf{T}^* oder wenn der Gültigkeitsbereich eines Objektes vom Typ \mathbf{T} endet.
- Falls kein Destruktor deklariert ist, so wird er automatisch erzeugt und ruft die Destruktoren für die Membervariablen auf (Zeiger `topn`, kein Effekt – Grund für Zombie-Elemente)

Mit dem Destruktor klappt's!

```
// POST: the dynamic memory of *this is deleted
stack::~~stack(){
    while (topn != nullptr){
        llnode* t = topn;
        topn = t->next;
        delete t;
    }
}
```

Mit dem Destruktor klappt's!

```
// POST: the dynamic memory of *this is deleted
stack::~~stack(){
    while (topn != nullptr){
        llnode* t = topn;
        topn = t->next;
        delete t;
    }
}
```

- löscht automatisch alle Stapелеlemente, wenn der Stapel ungültig wird

Mit dem Destruktor klappt's!

```
// POST: the dynamic memory of *this is deleted
stack::~~stack(){
    while (topn != nullptr){
        llnode* t = topn;
        topn = t->next;
        delete t;
    }
}
```

- löscht automatisch alle Stapелеlemente, wenn der Stapel ungültig wird
- Unsere Stapel-Klasse scheint jetzt die Richtlinie “Dynamischer Speicher” zu befolgen (?)

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n";
```

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n";
```

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n";
```

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n";
```

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n";
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```


Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n";
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```

Stapel fertig?

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop (); // Oops, Programmabsturz!
```

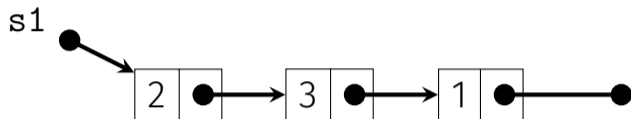
```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1
```

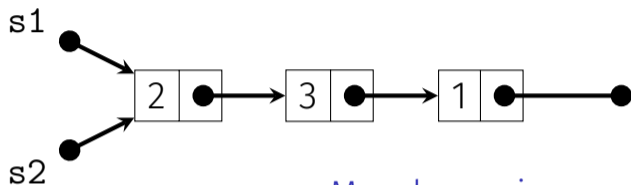
```
s2.pop (); // Oops, Programmabsturz!
```

Was ist hier schiefgegangen?



```
...  
stack s2 = s1;  
std::cout << s2 << "\n";  
  
s1.pop ();  
std::cout << s1 << "\n";  
  
s2.pop ();
```

Was ist hier schiefgegangen?



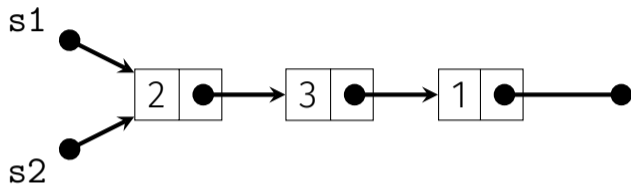
Memberweise

Initialisierung:

kopiert nur den topn-Zeiger

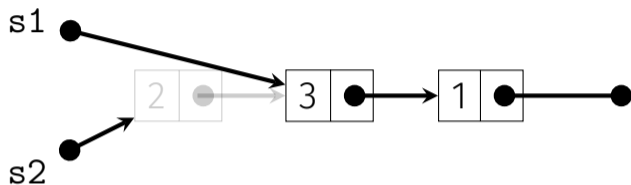
```
...  
stack s2 = s1;  
std::cout << s2 << "\n";  
  
s1.pop ();  
std::cout << s1 << "\n";  
  
s2.pop ();
```

Was ist hier schiefgegangen?



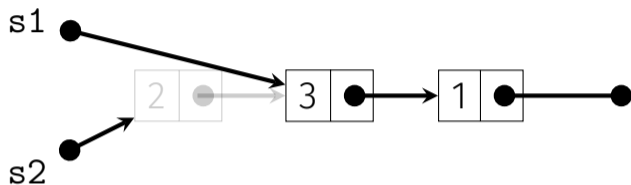
```
...  
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1  
  
s1.pop ();  
std::cout << s1 << "\n";  
  
s2.pop ();
```


Was ist hier schiefgegangen?



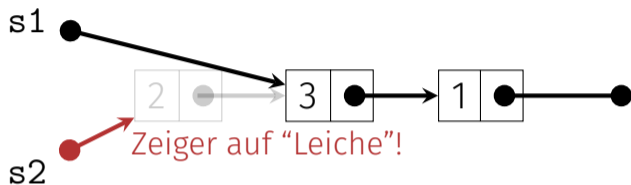
```
...  
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1  
  
s1.pop ();  
std::cout << s1 << "\n";  
  
s2.pop ();
```

Was ist hier schiefgegangen?



```
...  
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1  
  
s1.pop ();  
std::cout << s1 << "\n"; // 3 1  
  
s2.pop ();
```

Was ist hier schiefgegangen?

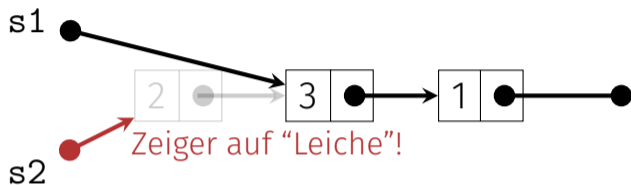


```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();
```

Was ist hier schiefgegangen?



```
...  
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1  
  
s1.pop ();  
std::cout << s1 << "\n"; // 3 1  
  
s2.pop (); // Oops, Programmabsturz!
```

Das eigentliche Problem

Schon das geht schief:

```
{  
    stack s1;  
    s1.push(1);  
    stack s2 = s1;  
}
```

Beim Verlassen des Gültigkeitsbereiches werden beide Stacks aufgeräumt (dekonstruiert). Aber beide Stacks versuchen dieselben Daten zu löschen, denn sie haben **Zugriff auf denselben Zeiger**.

Lösungsmöglichkeiten

Smart-Pointers (werden hier nicht weiter vertieft):

- Zähle die Anzahl Zeiger, die auf ein Objekt verweisen. Lösche nur wenn diese Anzahl auf 0 zurückfällt:

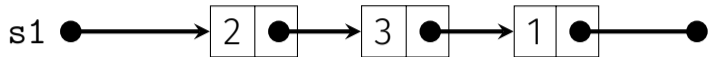
`std::shared_pointer`

- Verhindere, dass mehrere Zeiger auf ein Objekt zeigen können: **`std::unique_pointer`**.

oder:

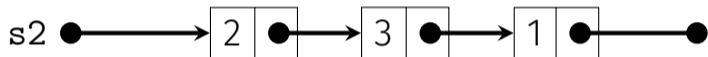
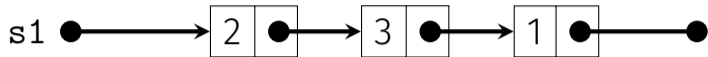
- Wir erstellen eine echte Kopie aller Daten – wie folgt.

Wir erstellen eine echte Kopie!



```
...  
stack s2 = s1;  
std::cout << s2 << "\n";  
  
s1.pop ();  
std::cout << s1 << "\n";  
  
s2.pop ();
```

Wir erstellen eine echte Kopie!



...

```
stack s2 = s1;
```

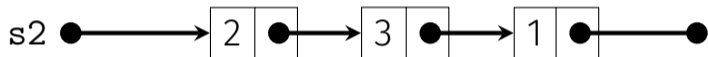
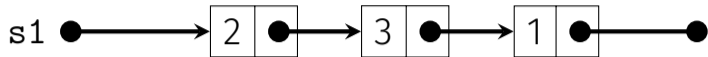
```
std::cout << s2 << "\n";
```

```
s1.pop ();
```

```
std::cout << s1 << "\n";
```

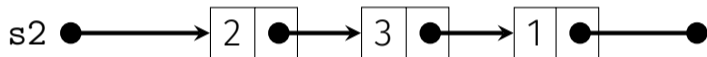
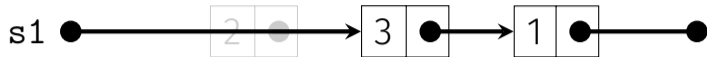
```
s2.pop ();
```


Wir erstellen eine echte Kopie!



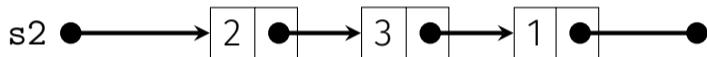
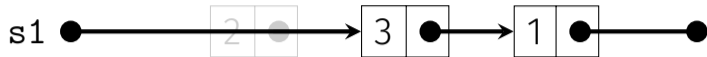
```
...  
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1  
  
s1.pop ();  
std::cout << s1 << "\n";  
  
s2.pop ();
```

Wir erstellen eine echte Kopie!



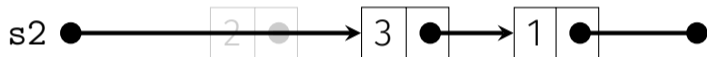
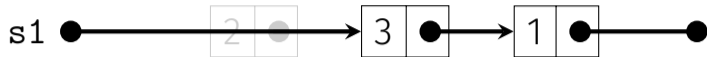
```
...  
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1  
  
s1.pop ();  
std::cout << s1 << "\n";  
  
s2.pop ();
```

Wir erstellen eine echte Kopie!



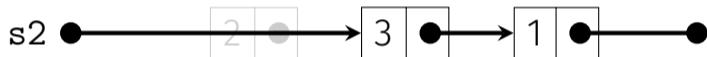
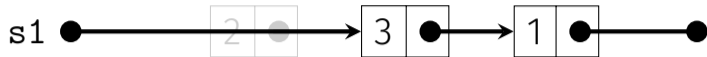
```
...  
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1  
  
s1.pop ();  
std::cout << s1 << "\n"; // 3 1  
  
s2.pop ();
```

Wir erstellen eine echte Kopie!



```
...  
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1  
  
s1.pop ();  
std::cout << s1 << "\n"; // 3 1  
  
s2.pop ();
```

Wir erstellen eine echte Kopie!



```
...  
stack s2 = s1;  
std::cout << s2 << "\n"; // 2 3 1  
  
s1.pop ();  
std::cout << s1 << "\n"; // 3 1  
  
s2.pop (); // ok
```

Der Copy-Konstruktor

- Der Copy-Konstruktor einer Klasse T ist der eindeutige Konstruktor mit Deklaration

$$T(\text{const } T\& x);$$

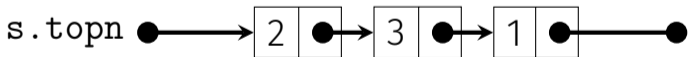
- wird automatisch aufgerufen, wenn Werte vom Typ T mit Werten vom Typ T *initialisiert* werden

$$T\ x = t; \quad (t \text{ vom Typ } T)$$
$$T\ x(t);$$

- Falls kein Copy-Konstruktor deklariert ist, so wird er automatisch erzeugt (und initialisiert memberweise – Grund für obiges Problem)

Mit dem Copy-Konstruktor klappt's!

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```

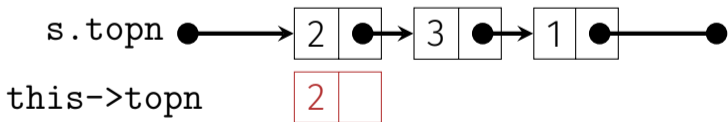


`this->topn` ●—●

prev

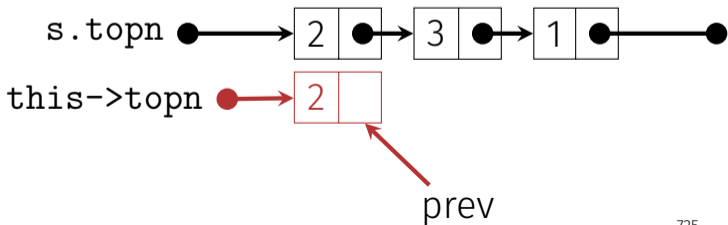
Mit dem Copy-Konstruktor klappt's!

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



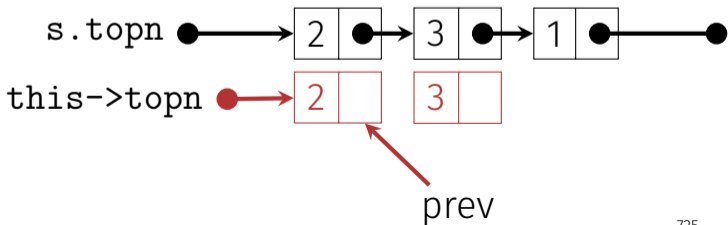
Mit dem Copy-Konstruktor klappt's!

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



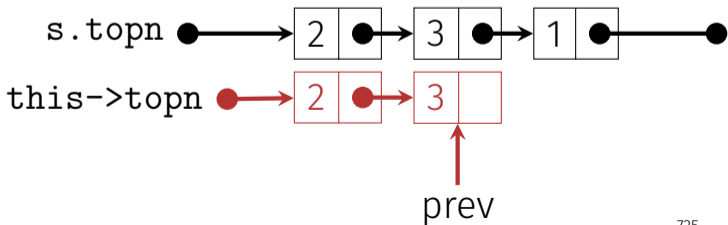
Mit dem Copy-Konstruktor klappt's!

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



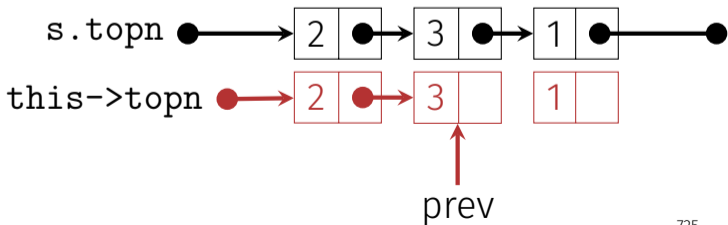
Mit dem Copy-Konstruktor klappt's!

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



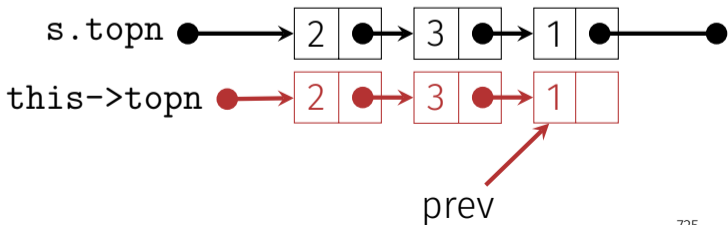
Mit dem Copy-Konstruktor klappt's!

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



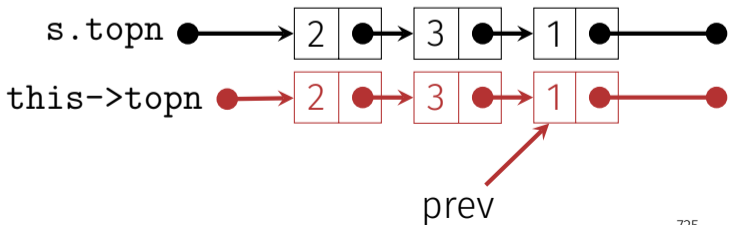
Mit dem Copy-Konstruktor klappt's!

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



Mit dem Copy-Konstruktor klappt's!

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



Initialisierung \neq Zuweisung!

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1; // Initialisierung
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1  
s2.pop (); // ok: Copy-Konstruktor!
```

Initialisierung \neq Zuweisung!

```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2;  
s2 = s1; // Zuweisung
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1  
s2.pop (); // Oops, Programmabsturz!
```


Der Zuweisungsoperator

- Überladung von **operator=** als Memberfunktion
- Wie Copy-Konstruktor ohne Initialisierer, aber zusätzlich
 - Freigabe des Speichers für den „alten“ Wert
 - Prüfen auf Selbstzuweisungen (`s1=s1`), die keinen Effekt haben sollen
- Falls kein Zuweisungsoperator deklariert ist, so wird er automatisch erzeugt (und weist memberweise zu – Grund für obiges Problem)

Mit dem Zuweisungsoperator klappt's!

```
// POST: *this (left operand) becomes a  
//           copy of s (right operand)  
stack& stack::operator= (const stack& s)
```

Mit dem Zuweisungsoperator klappt's!

```
// POST: *this (left operand) becomes a  
//           copy of s (right operand)  
stack& stack::operator= (const stack& s){  
    if (topn != s.topn){ // keine Selbstzuweisung
```

Mit dem Zuweisungsoperator klappt's!

```
// POST: *this (left operand) becomes a  
//           copy of s (right operand)  
stack& stack::operator= (const stack& s){  
    if (topn != s.topn){ // keine Selbstzuweisung  
        stack copy = s; // Kopierkonstruktor
```

Mit dem Zuweisungsoperator klappt's!

```
// POST: *this (left operand) becomes a
//           copy of s (right operand)
stack& stack::operator= (const stack& s){
    if (topn != s.topn){ // keine Selbstzuweisung
        stack copy = s; // Kopierkonstruktor
        std::swap(topn, copy.topn); // copy hat nun den Müll!
```

Mit dem Zuweisungsoperator klappt's!

```
// POST: *this (left operand) becomes a
//           copy of s (right operand)
stack& stack::operator= (const stack& s){
    if (topn != s.topn){ // keine Selbstzuweisung
        stack copy = s; // Kopierkonstruktor
        std::swap(topn, copy.topn); // copy hat nun den Müll!
    } // copy wird aufgeräumt -> Dekonstruktion
    return *this; // Rueckgabe als L-Wert (Konvention)
}
```

Mit dem Zuweisungsoperator klappt's!

```
// POST: *this (left operand) becomes a
//           copy of s (right operand)
stack& stack::operator= (const stack& s){
    if (topn != s.topn){ // keine Selbstzuweisung
        stack copy = s; // Kopierkonstruktor
        std::swap(topn, copy.topn); // copy hat nun den Müll!
    } // copy wird aufgeräumt -> Dekonstruktion
    return *this; // Rueckgabe als L-Wert (Konvention)
}
```

Cooler Trick! 😊

Fertig

```
class stack{
public:
    stack(); // constructor
    ~stack(); // destructor
    stack(const stack& s); // copy constructor
    stack& operator=(const stack& s); // assignment operator

    void push(int value);
    void pop();
    int top() const;
    bool empty() const;
    void print(std::ostream& out) const;
private:
    llnode* topn;
}
```


Dynamischer Datentyp

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Stapel)
- Mindestfunktionalität:
 - Konstruktoren
 - Destruktor
 - Copy-Konstruktor
 - Zuweisungsoperator

Dynamischer Datentyp

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Stapel)
 - Mindestfunktionalität:
 - Konstruktoren
 - Destruktor
 - Copy-Konstruktor
 - Zuweisungsoperator
- Dreierregel:** definiert eine Klasse eines davon, so muss sie auch die anderen zwei definieren!

Bäume sind

- Verallgemeinerte Listen: Knoten können mehrere Nachfolger haben
- Spezielle Graphen: Graphen bestehen aus Knoten und Kanten. Ein Baum ist ein zusammenhängender, gerichteter, azyklischer Graph.

Bäume

Verwendung

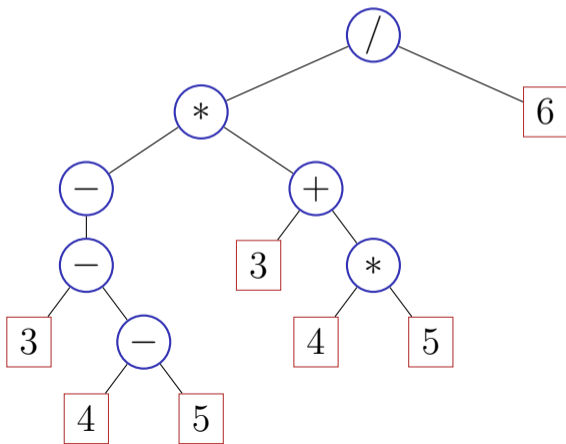
- Entscheidungsbäume: Hierarchische Darstellung von Entscheidungsregeln
- Codebäume: Darstellung eines Codes, z.B. Morsealphabet, Huffman Code
- Suchbäume: ermöglichen effizientes Suchen eines Elementes
- Syntaxbäume: Parsen und Traversieren von Ausdrücken, z.B. in einem Compiler



Bäumen werden in anderen Vorlesungen vertiefter behandelt (Datenstrukturen und Algorithmen (RW) / Algorithmen und Komplexität (Mathematik Bachelor))

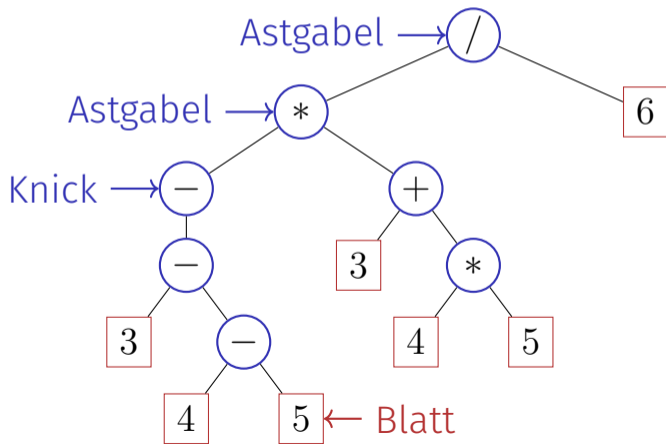
(Ausdrucks-)Bäume

$$-(3-(4-5))*(3+4*5)/6$$



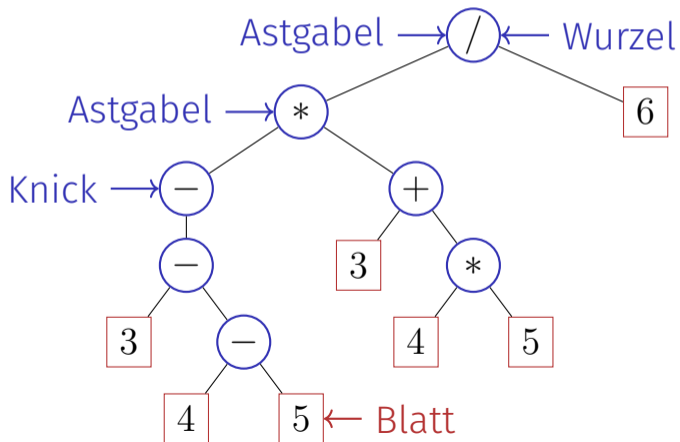
(Ausdrucks-)Bäume

$$-(3-(4-5))*(3+4*5)/6$$



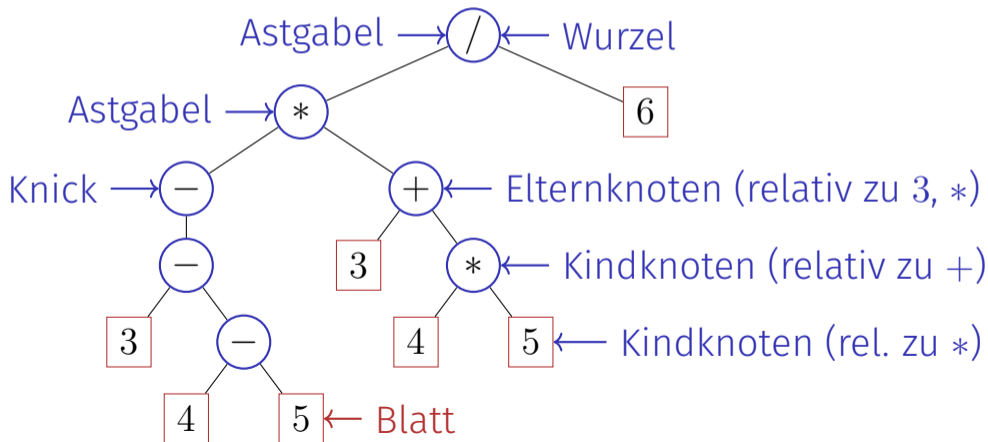
(Ausdrucks-)Bäume

$$-(3-(4-5))*(3+4*5)/6$$

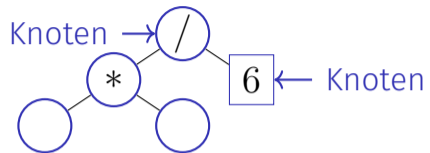


(Ausdrucks-)Bäume

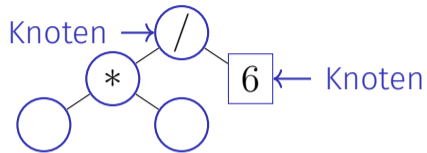
$$-(3-(4-5))*(3+4*5)/6$$



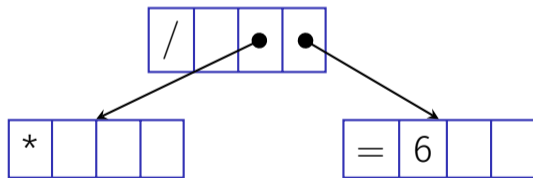
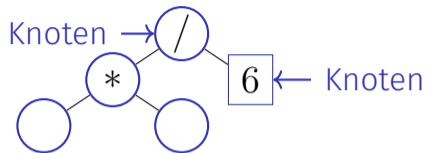
Astgabeln + Blätter + Knicke = Knoten



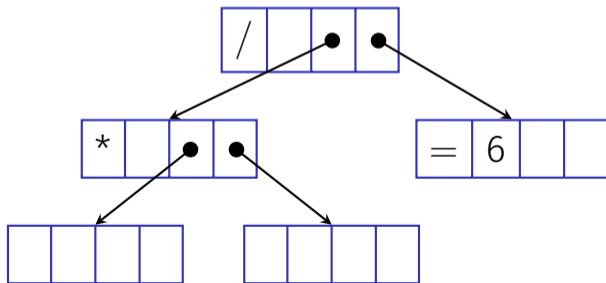
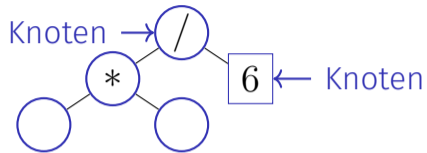
Astgabeln + Blätter + Knicke = Knoten



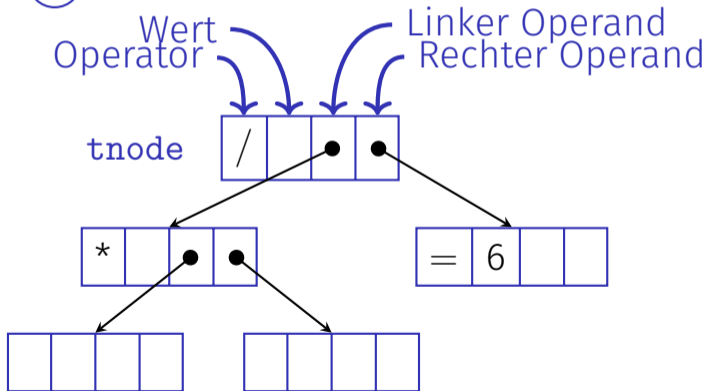
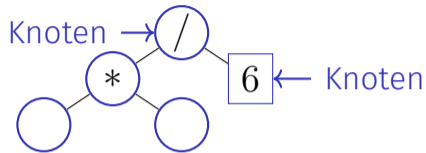
Astgabeln + Blätter + Knicke = Knoten



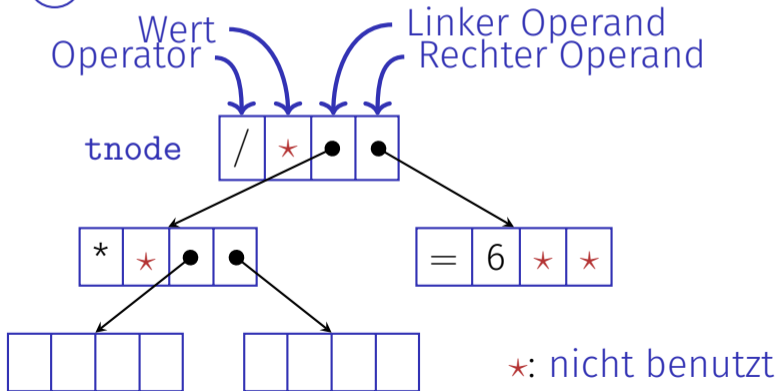
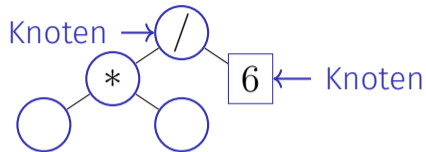
Astgabeln + Blätter + Knicke = Knoten



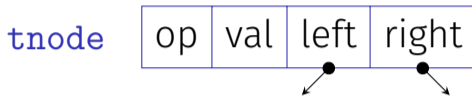
Astgabeln + Blätter + Knicke = Knoten



Astgabeln + Blätter + Knicke = Knoten



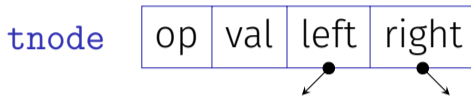
Knoten (struct tnode)



```
struct tnode {
    char op; // leaf node: op is '='
             // internal node: op is '+', '-', '*' or '/'
    double val;
    tnode* left;
    tnode* right;

    tnode(char o, double v, tnode* l, tnode* r)
        : op(o), val(v), left(l), right(r) {}
};
```

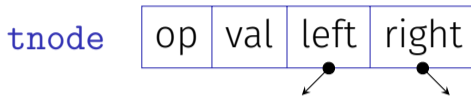
Knoten (struct tnode)



```
struct tnode {
    char op; // leaf node: op is '='
             // internal node: op is '+', '-', '*' or '/'
    double val;
    tnode* left; // == nullptr for unary minus
    tnode* right;

    tnode(char o, double v, tnode* l, tnode* r)
        : op(o), val(v), left(l), right(r) {}
};
```

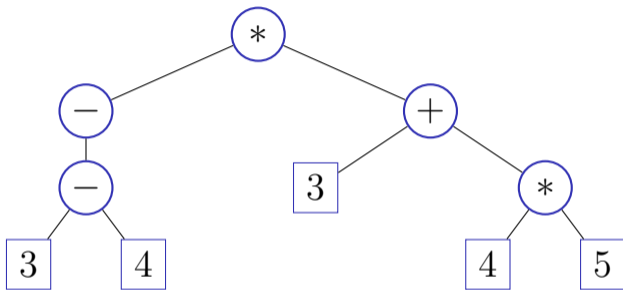

Knoten (struct tnode)



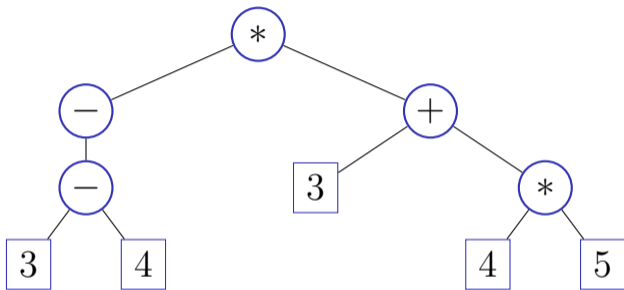
```
struct tnode {
    char op; // leaf node: op is '='
             // internal node: op is '+', '-', '*' or '/'
    double val;
    tnode* left; // == nullptr for unary minus
    tnode* right;

    tnode(char o, double v, tnode* l, tnode* r)
        : op(o), val(v), left(l), right(r) {}
};
```

Grösse = Knoten in Teilbäumen zählen

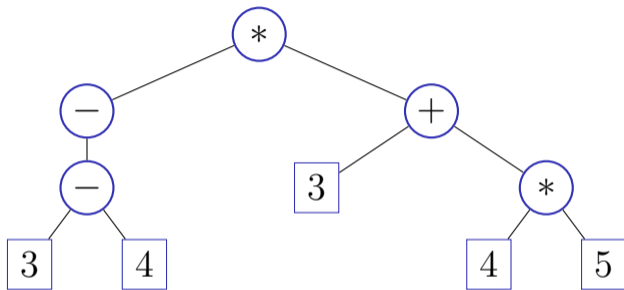


Grösse = Knoten in Teilbäumen zählen



- Grösse eines Blattes: 1
- Grösse anderer Knoten: 1 + Gesamtgrösse aller Kindknoten

Grösse = Knoten in Teilbäumen zählen



- Grösse eines Blattes: 1
- Grösse anderer Knoten: 1 + Gesamtgrösse aller Kindknoten
- Z.B. Grösse des „+“-Knoten ist 5

Knoten in Teilbäumen zählen

```
// POST: returns the size (number of nodes) of
//       the subtree with root n
int size (const tnode* n) {
    if (n){ // shortcut for n != nullptr
        return size(n->left) + size(n->right) + 1;
    }
    return 0;
}
```



Teilbäume auswerten

```
// POST: evaluates the subtree with root n
```

```
double eval(const tnode* n){
```

```
    assert(n);
```

```
    if (n->op == '=') return n->val; ← Blatt...
```

```
    double l = 0; ...oder Astgabel:
```

```
    if (n->left) l = eval(n->left); ← op unär, oder linker Ast
```

```
    double r = eval(n->right); ← rechter Ast
```

```
    switch(n->op){
```

```
        case '+': return l+r;
```

```
        case '-': return l-r;
```

```
        case '*': return l*r;
```

```
        case '/': return l/r;
```

```
        default: return 0;
```

```
    }
```

```
}
```



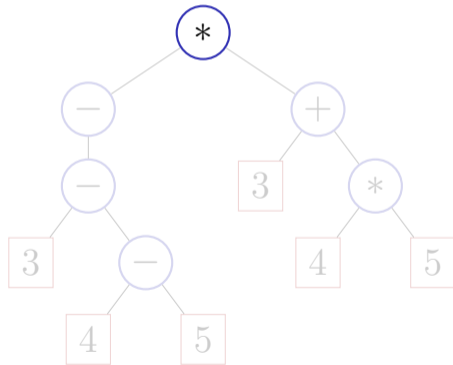
Teilbäume klonen

```
// POST: a copy of the subtree with root n is made  
//       and a pointer to its root node is returned  
tnode* copy (const tnode* n) {  
    if (n == nullptr)  
        return nullptr;  
    return new tnode (n->op, n->val, copy(n->left), copy(n->right));  
}
```



Teilbäume fällen

```
// POST: all nodes in the subtree with root n are deleted  
void clear(tnode* n) {  
    if(n){  
        clear(n->left);  
        clear(n->right);  
        delete n;  
    }  
}
```



Teilbäume nutzen

```
// Construct a tree for 1 - (-(3 + 7))
tnode* n1 = new tnode( '=', 3, nullptr, nullptr);
tnode* n2 = new tnode( '=', 7, nullptr, nullptr);
tnode* n3 = new tnode( '+', 0, n1, n2);
tnode* n4 = new tnode( '-', 0, nullptr, n3);
tnode* n5 = new tnode( '=', 1, nullptr, nullptr);
tnode* root = new tnode( '-', 0, n5, n4);

// Evaluate the overall tree
std::cout << "1 - (-(3 + 7)) = " << eval(root) << '\n';

// Evaluate a subtree
std::cout << "3 + 7 = " << eval(n3) << '\n';

clear(root); // free memory
```

Bäume pflanzen

```
class texpression {  
public:  
    texpression (double d)  
        : root (new tnode ('=', d, 0, 0)) {}  
    ...  
private:  
    tnode* root;  
};
```

erzeugt Baum mit
einem Blatt

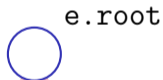


Bäume wachsen lassen

```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```



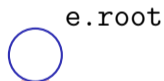
*this



e

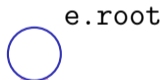
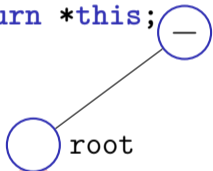
Bäume wachsen lassen

```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```



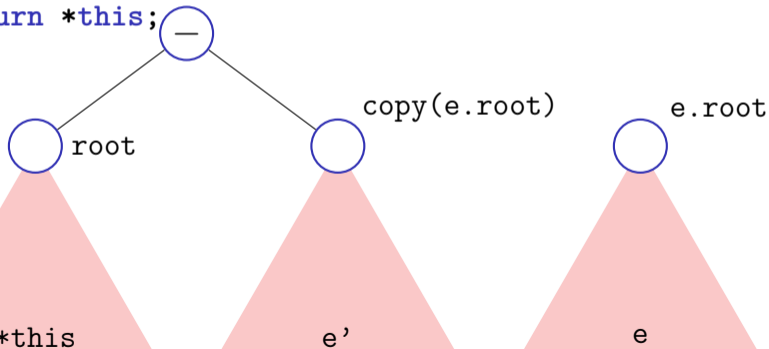
Bäume wachsen lassen

```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```



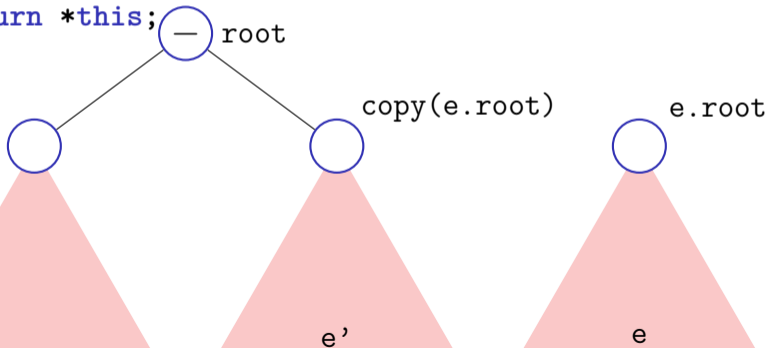
Bäume wachsen lassen

```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```



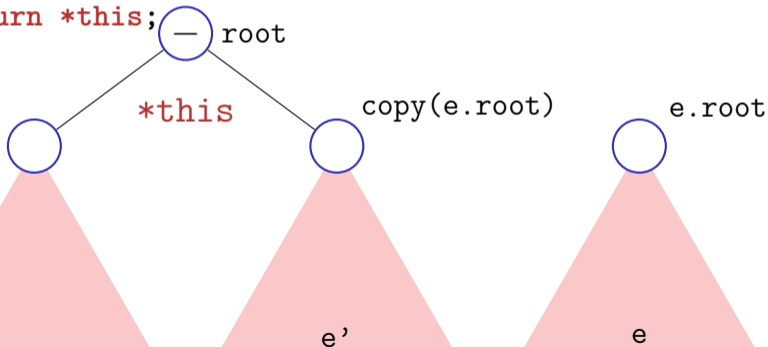
Bäume wachsen lassen

```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```



Bäume wachsen lassen

```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```



Bäume züchten

```
texpression operator- (const texpression& l,  
                      const texpression& r){  
    texpression result = l;  
    return result -= r;  
}
```

```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```

Bäume züchten

```
expression operator- (const texpression& l,  
                    const texpression& r){  
    texpression result = l;  
    return result -= r;  
}
```

```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```

3

Bäume züchten

```
texpression operator- (const texpression& l,  
                      const texpression& r){  
    texpression result = l;  
    return result -= r;  
}
```

```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```

3

4

Bäume züchten

```
expression operator- (const texpression& l,  
                    const texpression& r){  
    texpression result = l;  
    return result -= r;  
}
```

```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```

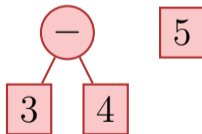
3 4

5

Bäume züchten

```
expression operator- (const texpression& l,  
                    const texpression& r){  
    texpression result = l;  
    return result -= r;  
}
```

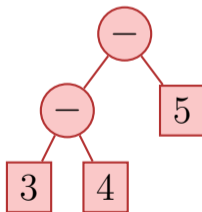
```
texpression a = 3;  
texpression b = 4;  
texpression c = 5;  
texpression d = a-b-c;
```



Bäume züchten

```
tepression operator- (const tepression& l,  
                    const tepression& r){  
    tepression result = l;  
    return result -= r;  
}
```

```
tepression a = 3;  
tepression b = 4;  
tepression c = 5;  
tepression d = a-b-c;
```



Dreierregel: Bäume klonen, reproduzieren und fällen

```
texpression::~~texpression(){  
    clear(root);  
}
```

```
texpression::texpression (const texpression& e)  
    : root(copy(e.root)) { }
```

```
texpression& texpression::operator=(const texpression& e){  
    if (root != e.root){  
        texpression cp = e;  
        std::swap(cp.root, root);  
    }  
    return *this;  
}
```

Zusammengefasst

```
class expression{
public:
    expression (double d); // constructor
    ~expression(); // destructor
    expression (const expression& e); // copy constructor
    expression& operator=(const expression& e); // assignment op
    expression operator-();
    expression& operator-=(const expression& e);
    expression& operator+=(const expression& e);
    expression& operator*=(const expression& e);
    expression& operator/=(const expression& e);
    double evaluate();
private:
    tnode* root;
};
```

Werte zu Bäumen!

```
// term = factor { "*" factor | "/" factor }
double term (std::istream& is){
    double value = factor (is);
    while (true) {
        if (consume (is, '*'))
            value *= factor (is);
        else if (consume (is, '/'))
            value /= factor (is);
        else
            return value;
    }
}
```

calculator.cpp
(Ausdruckswert)

Werte zu Bäumen!

```
using number_type = double;
```

```
// term = factor { "*" factor | "/" factor }  
number_type term (std::istream& is){  
    number_type value = factor (is);  
    while (true) {  
        if (consume (is, '*'))  
            value *= factor (is);  
        else if (consume (is, '/'))  
            value /= factor (is);  
        else  
            return value;  
    }  
}
```

double_calculator.cpp
(Ausdruckswert)

Werte zu Bäumen!

```
using number_type = texpression ;
```

```
// term = factor { "*" factor | "/" factor }
```

```
number_type term (std::istream& is){
```

```
    number_type value = factor (is);
```

```
    while (true) {
```

```
        if (consume (is, '*'))
```

```
            value *= factor (is);
```

```
        else if (consume (is, '/'))
```

```
            value /= factor (is);
```

```
    else
```

```
        return value;
```

```
    }
```

```
}
```

double_calculator.cpp
(Ausdruckswert)

→

texpression_calculator.cpp
(Ausdrucksbaum)

Abschliessende Bemerkung

- Wir haben in dieser Vorlesung die Knoten für Liste und Baum bewusst ohne Memberfunktionen implementiert. Wir betrachten sie nämlich als reine Datencontainer ohne eigene Intelligenz.⁸
- Wenn Vererbung und Polymorphie im Spiel ist, ist die Implementation der Funktionalität wie **evaluate**, **print**, **clear**, **copy** (etc.) mit Memberfunktionen vorzuziehen.
- In jedem Falle implementiert man die Speicherverwaltung der zusammengesetzten Datenstruktur Liste / Baum nicht in den Knotenklassen.

⁸Teile der Implementation waren so sogar einfacher, da der Fall `n==nullptr` einfacher abgefangen werden kann

24. Subtyping, Polymorphie und Vererbung

Ausdrückbäume, Aufgabenteilung und Modularisierung, Typhierarchien, virtuelle Funktionen, dynamische Bindung, Code-Wiederverwendung, Konzepte der objektorientierten Programmierung

Letzte Woche: Ausdrucksbäume

- Ziel: Arithmetische Ausdrücke repräsentieren, z.B.

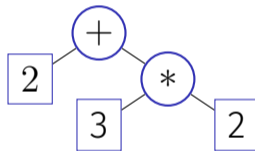
$$2 + 3 * 2$$

Letzte Woche: Ausdrucksbäume

- Ziel: Arithmetische Ausdrücke repräsentieren, z.B.

$$2 + 3 * 2$$

- Arithmetische Ausdrücke bilden eine *Baumstruktur*

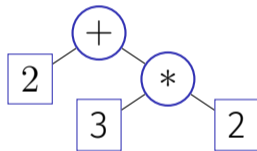


Letzte Woche: Ausdrucksbäume

- Ziel: Arithmetische Ausdrücke repräsentieren, z.B.

$$2 + 3 * 2$$

- Arithmetische Ausdrücke bilden eine *Baumstruktur*



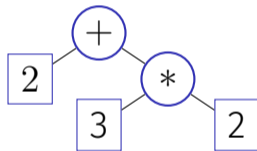
- Ausdrucksbäume bestehen aus *unterschiedlichen* Knoten:

Letzte Woche: Ausdrucksbäume

- Ziel: Arithmetische Ausdrücke repräsentieren, z.B.

$$2 + 3 * 2$$

- Arithmetische Ausdrücke bilden eine *Baumstruktur*

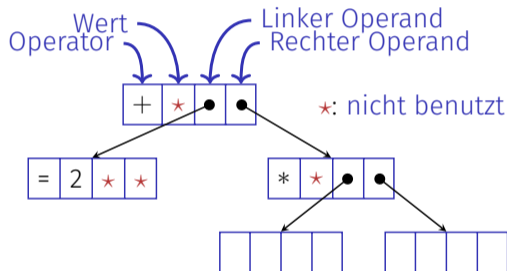


- Ausdrucksbäume bestehen aus *unterschiedlichen* Knoten: Literale (z.B. 2), binäre Operatoren (z.B. +), unäre Operatoren (z.B. $\sqrt{\quad}$), Funktionsanwendungen (z.B. \cos), etc.

Nachteile

Implementiert mittels *eines einzigen* Knotentyps:

```
struct tnode {
    char op; // Operator ('=' for literals)
    double val; // Literal's value
    tnode* left; // Left child (or nullptr)
    tnode* right; // ...
    ...
};
```



Beobachtung: `tnode` ist die „Summe“ aller benötigten Knoten (Konstanten, Addition, ...) \Rightarrow Speicherverschwendung, unelegant

Nachteile

Beobachtung: `tnode` ist die „Summe“ aller benötigten Knoten

–

Nachteile

Beobachtung: `tnode` ist die „Summe“ aller benötigten Knoten
– und jede Funktion muss diese „Summe“ wieder
„auseinander nehmen“, z.B.:

```
double eval(const tnode* n) {
    if (n->op == '=') return n->val; // n is a constant
    double l = 0;
    if (n->left) l = eval(n->left); // n is not a unary operator
    double r = eval(n->right);
    switch(n->op) {
        case '+': return l+r; // n is an addition node
        case '*': return l*r; // ...
        ...
    }
}
```

Nachteile

Beobachtung: `tnode` ist die „Summe“ aller benötigten Knoten – und jede Funktion muss diese „Summe“ wieder „auseinander nehmen“, z.B.:

```
double eval(const tnode* n) {
    if (n->op == '=') return n->val; // n is a constant
    double l = 0;
    if (n->left) l = eval(n->left); // n is not a unary operator
    double r = eval(n->right);
    switch(n->op) {
        case '+': return l+r; // n is an addition node
        case '*': return l*r; // ...
        ...
    }
}
```

⇒ Umständlich und somit fehleranfällig

Nachteile

```
struct tnode {  
    char op;  
    double val;  
    tnode* left;  
    tnode* right;  
    ...  
};
```

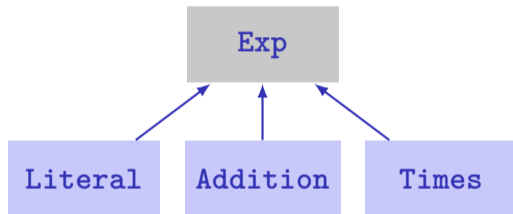
```
double eval(const tnode* n) {  
    if (n->op == '=') return n->val;  
    double l = 0;  
    if (n->left) l = eval(n->left);  
    double r = eval(n->right);  
    switch(n->op) {  
        case '+': return l+r;  
        case '*': return l*r;  
        ...  
    }
```

Dieser Code ist nicht *modular* – das ändern wir heute!

Neue Konzepte heute

1. Subtyping

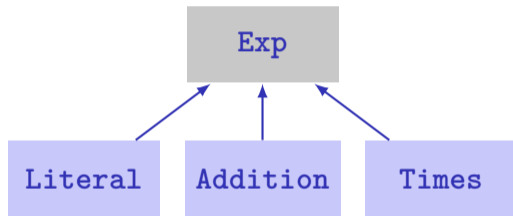
- Typhierarchie: **Exp** repräsentiert allgemeine Ausdrücke, **Literal** etc. sind konkrete Ausdrücke



Neue Konzepte heute

1. Subtyping

- Typhierarchie: **Exp** repräsentiert allgemeine Ausdrücke, **Literal** etc. sind konkrete Ausdrücke
- Jedes **Literal** etc. ist auch ein **Exp** (Subtyp-Beziehung)

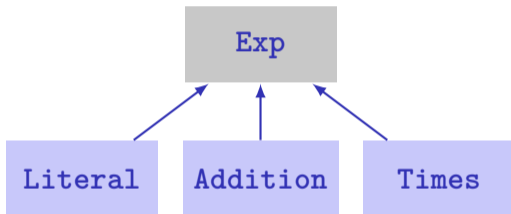


Neue Konzepte heute

1. Subtyping

- Typhierarchie: **Exp** repräsentiert allgemeine Ausdrücke, **Literal** etc. sind konkrete Ausdrücke
- Jedes **Literal** etc. ist auch ein **Exp** (Subtyp-Beziehung)
- Deswegen kann ein **Literal** etc. überall dort genutzt werden, wo ein **Exp** erwartet wird:

```
Exp* e = new Literal(132);
```



Neue Konzepte heute

2. Polymorphie und dynamische Bindung

- Eine Variable vom *statischen* Typ **Exp** kann Ausdrücke mit unterschiedlichen *dynamischen* Typen „beherbergen“:

```
Exp* e = new Literal(2); // e is the literal 2  
e = new Addition(e, e); // e is the addition 2 + 2
```

Neue Konzepte heute

2. Polymorphie und dynamische Bindung

- Eine Variable vom *statischen* Typ **Exp** kann Ausdrücke mit unterschiedlichen *dynamischen* Typen „beherbergen“:

```
Exp* e = new Literal(2); // e is the literal 2
e = new Addition(e, e); // e is the addition 2 + 2
```

- Ausgeführt werden die Memberfunktionen des *dynamischen* Typs:

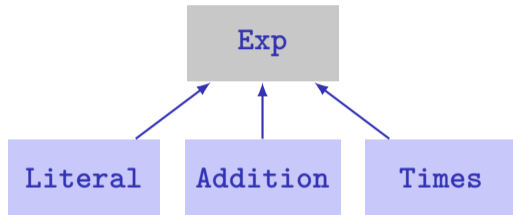
```
Exp* e = new Literal(2);
std::cout << e->eval(); // 2

e = new Addition(e, e);
std::cout << e->eval(); // 4
```

Neue Konzepte heute

3. Vererbung

- Manche Funktionalität ist für mehrere Mitglieder der Typhierarchie gleich

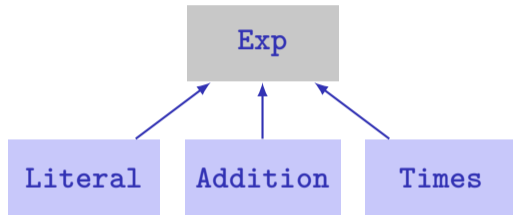


Neue Konzepte heute

3. Vererbung

- Manche Funktionalität ist für mehrere Mitglieder der Typhierarchie gleich
- Z.B. die Berechnung der Grösse (Verschachtelungstiefe) binärer Ausdrücke (**Addition**, **Times**):

$$1 + \text{size}(\text{left operand}) + \text{size}(\text{right operand})$$



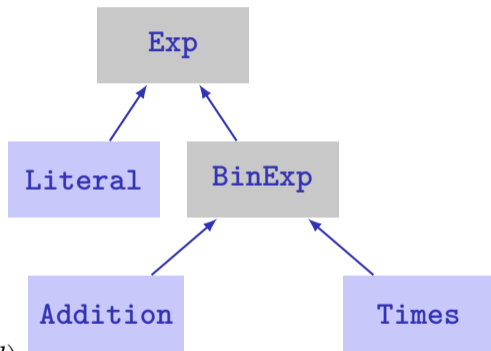
Neue Konzepte heute

3. Vererbung

- Manche Funktionalität ist für mehrere Mitglieder der Typhierarchie gleich
- Z.B. die Berechnung der Grösse (Verschachtelungstiefe) binärer Ausdrücke (**Addition**, **Times**):

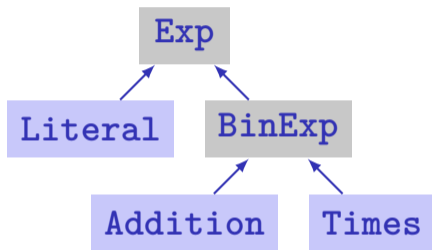
$1 + \text{size}(\text{left operand}) + \text{size}(\text{right operand})$

⇒ Funktionalität einmal implementieren und dann an Subtypen *vererben*



Vorteile

- Subtyping, Polymorphie und dynamische Bindung ermöglichen *Modularisierung durch Spezialisierung*

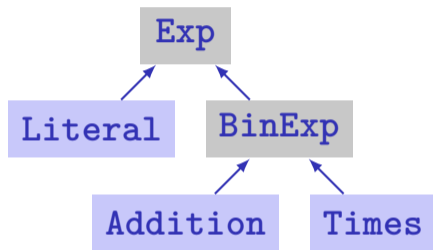


```
Exp* e = new Literal(2);  
std::cout << e->eval();
```

```
e = new Addition(e, e);  
std::cout << e->eval();
```

Vorteile

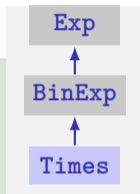
- Subtyping, Polymorphie und dynamische Bindung ermöglichen *Modularisierung durch Spezialisierung*
- Vererbung erlaubt gemeinsamen Code trotz Modularisierung
⇒ *Codeduplikation vermeiden*



```
Exp* e = new Literal(2);  
std::cout << e->eval();  
  
e = new Addition(e, e);  
std::cout << e->eval();
```

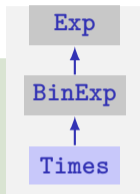
Syntax und Terminologie

```
struct Exp {  
    ...  
}  
  
struct BinExp : public Exp {  
    ...  
}  
  
struct Times : public BinExp {  
    ...  
}
```



Syntax und Terminologie

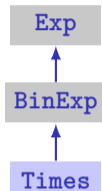
```
struct Exp {  
    ...  
}  
  
struct BinExp : public Exp {  
    ...  
}  
  
struct Times : public BinExp {  
    ...  
}
```



Anmerkung: Wir konzentrieren uns heute auf die neuen Konzepte (Subtyping, ...) und ignorieren den davon unabhängigen Aspekt der Kapselung (**class, private** vs. **public** Membervariablen)

Syntax und Terminologie

```
struct Exp {  
    ...  
}  
  
struct BinExp : public Exp {  
    ...  
}  
  
struct Times : public BinExp {  
    ...  
}
```



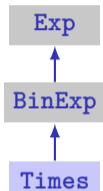
- **BinExp** ist eine von **Exp** *abgeleitete Klasse*¹

¹Subklasse, Kindklasse

²Superklasse, Elternklasse

Syntax und Terminologie

```
struct Exp {  
    ...  
}  
  
struct BinExp : public Exp {  
    ...  
}  
  
struct Times : public BinExp {  
    ...  
}
```



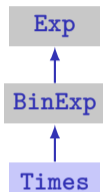
- **BinExp** ist eine von **Exp** *abgeleitete Klasse*¹
- **Exp** ist die *Basisklasse*² von **BinExp**

¹Subklasse, Kindklasse

²Superklasse, Elternklasse

Syntax und Terminologie

```
struct Exp {  
    ...  
}  
  
struct BinExp : public Exp {  
    ...  
}  
  
struct Times : public BinExp {  
    ...  
}
```



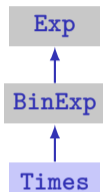
- **BinExp** ist eine von **Exp** *abgeleitete Klasse*¹
- **Exp** ist die *Basisklasse*² von **BinExp**
- **BinExp** *erbt* von **Exp**

¹Subklasse, Kindklasse

²Superklasse, Elternklasse

Syntax und Terminologie

```
struct Exp {  
    ...  
}  
  
struct BinExp : public Exp {  
    ...  
}  
  
struct Times : public BinExp {  
    ...  
}
```



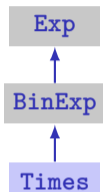
- **BinExp** ist eine von **Exp** *abgeleitete Klasse*¹
- **Exp** ist die *Basisklasse*² von **BinExp**
- **BinExp** *erbt* von **Exp**
- Die Vererbung von **Exp** zu **BinExp** ist *öffentlich* (**public**), daher ist **BinExp** ein *Subtyp* von **Exp**

¹Subklasse, Kindklasse

²Superklasse, Elternklasse

Syntax und Terminologie

```
struct Exp {  
    ...  
}  
  
struct BinExp : public Exp {  
    ...  
}  
  
struct Times : public BinExp {  
    ...  
}
```



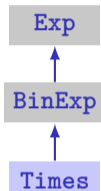
- **BinExp** ist eine von **Exp** *abgeleitete Klasse*¹
- **Exp** ist die *Basisklasse*² von **BinExp**
- **BinExp** *erbt* von **Exp**
- Die Vererbung von **Exp** zu **BinExp** ist *öffentlich* (**public**), daher ist **BinExp** ein *Subtyp* von **Exp**
- Analog: **Times** und **BinExp**

¹Subklasse, Kindklasse

²Superklasse, Elternklasse

Syntax und Terminologie

```
struct Exp {  
    ...  
}  
  
struct BinExp : public Exp {  
    ...  
}  
  
struct Times : public BinExp {  
    ...  
}
```



- **BinExp** ist eine von **Exp** *abgeleitete Klasse*¹
- **Exp** ist die *Basisklasse*² von **BinExp**
- **BinExp** *erbt* von **Exp**
- Die Vererbung von **Exp** zu **BinExp** ist *öffentlich* (**public**), daher ist **BinExp** ein *Subtyp* von **Exp**
- Analog: **Times** und **BinExp**
- Subtyprelation ist transitiv: **Times** ist ebenfalls ein Subtyp von **Exp**

¹Subklasse, Kindklasse

²Superklasse, Elternklasse

Abstrakte Klasse Exp und konkrete Klasse Literal

```
struct Exp {  
    virtual int size() const = 0;  
    virtual double eval() const = 0;  
};
```

Abstrakte Klasse Exp und konkrete Klasse Literal

```
struct Exp {  
    virtual int size() const = 0;  
    virtual double eval() const = 0;  
};
```

Aktiviert dynamische Bindung

Abstrakte Klasse Exp und konkrete Klasse Literal

```
struct Exp {  
    virtual int size() const = 0;  
    virtual double eval() const = 0;  
};
```

Erzwingt Implementierung
durch abgeleitete Klassen ...

Abstrakte Klasse Exp und konkrete Klasse Literal

```
struct Exp {  
    virtual int size() const = 0;  
    virtual double eval() const = 0;  
};
```

. das macht `Exp` zu einer *abstrakten* Klasse

Erzwingt Implementierung durch abgeleitete Klassen ...

Abstrakte Klasse Exp und konkrete Klasse Literal

```
struct Exp {  
    virtual int size() const = 0;  
    virtual double eval() const = 0;  
};
```

```
struct Literal : public Exp {  
    double val;  
  
    Literal(double v);  
    int size() const;  
    double eval() const;  
};
```


Abstrakte Klasse Exp und konkrete Klasse Literal

```
struct Exp {  
    virtual int size() const = 0;  
    virtual double eval() const = 0;  
};
```

```
struct Literal : public Exp { ← Literal erbt von Exp ...  
    double val;  
  
    Literal(double v);  
    int size() const;  
    double eval() const;  
};
```

Abstrakte Klasse Exp und konkrete Klasse Literal

```
struct Exp {  
    virtual int size() const = 0;  
    virtual double eval() const = 0;  
};
```

```
struct Literal : public Exp {  
    double val;  
  
    Literal(double v);  
    int size() const;  
    double eval() const;  
};
```

← Literal erbt von Exp ...

← ...ist aber ansonsten eine ganz normale Klasse

Literal: Implementierung

```
Literal::Literal(double v): val(v) {}
```

Literal: Implementierung

```
Literal::Literal(double v): val(v) {}
```

```
int Literal::size() const {  
    return 1;  
}
```

Literal: Implementierung

```
Literal::Literal(double v): val(v) {}
```

```
int Literal::size() const {  
    return 1;  
}
```

```
double Literal::eval() const {  
    return this->val;  
}
```

Subtyping: Ein Literal ist ein Ausdruck

Ein Zeiger auf einen Subtyp kann überall dort verwendet werden, wo ein Zeiger auf einen Supertyp gefordert ist:

```
Literal* lit = new Literal(5);
```

Subtyping: Ein Literal ist ein Ausdruck

Ein Zeiger auf einen Subtyp kann überall dort verwendet werden, wo ein Zeiger auf einen Supertyp gefordert ist:

```
Literal* lit = new Literal(5);  
Exp* e = lit; // OK: Literal is a subtype of Exp
```

Subtyping: Ein Literal ist ein Ausdruck

Ein Zeiger auf einen Subtyp kann überall dort verwendet werden, wo ein Zeiger auf einen Supertyp gefordert ist:

```
Literal* lit = new Literal(5);  
Exp* e = lit; // OK: Literal is a subtype of Exp
```

Aber nicht umgekehrt:

```
Exp* e = ...  
Literal* lit = e; // ERROR: Exp is not a subtype of Literal
```


Polymorphie: Ein Literal verhält sich wie ein Literal

```
struct Exp {  
    ...  
    virtual double eval();  
};  
  
double Literal::eval() {  
    return this->val;  
}
```

```
Exp* e = new Literal(3);  
std::cout << e->eval(); // 3
```

Polymorphie: Ein Literal verhält sich wie ein Literal

```
struct Exp {  
    ...  
    virtual double eval();  
};  
  
double Literal::eval() {  
    return this->val;  
}
```

```
Exp* e = new Literal(3);  
std::cout << e->eval(); // 3
```

- Virtuelle Memberfunktionen: der *dynamische* Typ (hier: **Literal**) bestimmt die auszuführenden Memberfunktionen
⇒ *dynamische Bindung*

Polymorphie: Ein Literal verhält sich wie ein Literal

```
struct Exp {  
    ...  
    virtual double eval();  
};  
  
double Literal::eval() {  
    return this->val;  
}
```

```
Exp* e = new Literal(3);  
std::cout << e->eval(); // 3
```

- *Virtuelle* Memberfunktionen: der *dynamische* Typ (hier: **Literal**) bestimmt die auszuführenden Memberfunktionen
⇒ *dynamische Bindung*
- Ohne **virtual** bestimmt der *statische* Typ (hier: **Exp**) die auszuführende Funktion

Polymorphie: Ein Literal verhält sich wie ein Literal

```
struct Exp {  
    ...  
    virtual double eval();  
};  
  
double Literal::eval() {  
    return this->val;  
}
```

```
Exp* e = new Literal(3);  
std::cout << e->eval(); // 3
```

- *Virtuelle* Memberfunktionen: der *dynamische* Typ (hier: **Literal**) bestimmt die auszuführenden Memberfunktionen
⇒ *dynamische Bindung*
- Ohne **virtual** bestimmt der *statische* Typ (hier: **Exp**) die auszuführende Funktion
- Wir vertiefen das nicht weiter

Weitere Ausdrücke: Addition und Times

```
struct Addition : public Exp {  
    Exp* left; // left operand  
    Exp* right; // right operand  
    ...  
};
```

Weitere Ausdrücke: Addition und Times

```
struct Addition : public Exp {  
    Exp* left; // left operand  
    Exp* right; // right operand  
    ...  
};
```

```
struct Times : public Exp {  
    Exp* left; // left operand  
    Exp* right; // right operand  
    ...  
};
```

Weitere Ausdrücke: Addition und Times

```
struct Addition : public Exp {  
    Exp* left; // left operand  
    Exp* right; // right operand  
    ...  
};
```

```
int Addition::size() const {  
    return 1 + left->size()  
           + right->size();  
}
```

```
struct Times : public Exp {  
    Exp* left; // left operand  
    Exp* right; // right operand  
    ...  
};
```

Weitere Ausdrücke: Addition und Times

```
struct Addition : public Exp {  
    Exp* left; // left operand  
    Exp* right; // right operand  
    ...  
};
```

```
int Addition::size() const {  
    return 1 + left->size()  
           + right->size();  
}
```

```
struct Times : public Exp {  
    Exp* left; // left operand  
    Exp* right; // right operand  
    ...  
};
```

```
int Times::size() const {  
    return 1 + left->size()  
           + right->size();  
}
```


Weitere Ausdrücke: Addition und Times

```
struct Addition : public Exp {  
    Exp* left; // left operand  
    Exp* right; // right operand  
    ...  
};
```

```
int Addition::size() const {  
    return 1 + left->size()  
           + right->size();  
}
```

```
struct Times : public Exp {  
    Exp* left; // left operand  
    Exp* right; // right operand  
    ...  
};
```

```
int Times::size() const {  
    return 1 + left->size()  
           + right->size();  
}
```

😊 Aufgabenteilung

Weitere Ausdrücke: Addition und Times

```
struct Addition : public Exp {  
    Exp* left; // left operand  
    Exp* right; // right operand  
    ...  
};
```

```
int Addition::size() const {  
    return 1 + left->size()  
           + right->size();  
}
```

```
struct Times : public Exp {  
    Exp* left; // left operand  
    Exp* right; // right operand  
    ...  
};
```

```
int Times::size() const {  
    return 1 + left->size()  
           + right->size();  
}
```



Aufgabenteilung



Codeduplizierung

Gemeinsamkeiten auslagern ...: BinExp

```
struct BinExp : public Exp {  
    Exp* left;  
    Exp* right;  
  
    BinExp(Exp* l, Exp* r);  
    int size() const;  
};
```

```
BinExp::BinExp(Exp* l, Exp* r): left(l), right(r) {}
```

Gemeinsamkeiten auslagern ...: BinExp

```
struct BinExp : public Exp {  
    Exp* left;  
    Exp* right;  
  
    BinExp(Exp* l, Exp* r);  
    int size() const;  
};
```

```
BinExp::BinExp(Exp* l, Exp* r): left(l), right(r) {}
```

```
int BinExp::size() const {  
    return 1 + this->left->size() + this->right->size();  
}
```

Bemerkung: **BinExp** implementiert **eval** nicht und ist daher, genau wie **Exp**, eine abstrakte Klasse

...Gemeinsamkeiten erben: Addition

```
struct Addition : public BinExp {  
    Addition(Exp* l, Exp* r);  
    double eval() const;  
};
```

...Gemeinsamkeiten erben: Addition

```
struct Addition : public BinExp {  
    Addition(Exp* l, Exp* r);  
    double eval() const;  
};
```

← **Addition** erbt Membervariablen (**left**, **right**) und Funktionen (**size**) von **BinExp**

...Gemeinsamkeiten erben: Addition

```
struct Addition : public BinExp {  
    Addition(Exp* l, Exp* r);  
    double eval() const;  
};
```

```
Addition::Addition(Exp* l, Exp* r): BinExp(l, r) {}
```

Aufruf des *Superkonstruktors* (Konstruktor von **BinExp**)
zwecks Initialisierung der Mem-
bervariablen **left** und **right**

...Gemeinsamkeiten erben: Addition

```
struct Addition : public BinExp {  
    Addition(Exp* l, Exp* r);  
    double eval() const;  
};
```

```
Addition::Addition(Exp* l, Exp* r): BinExp(l, r) {}
```

```
double Addition::eval() const {  
    return  
        this->left->eval() +  
        this->right->eval();  
}
```


...Gemeinsamkeiten erben: Times

```
struct Times : public BinExp {  
    Times(Exp* l, Exp* r);  
    double eval() const;  
};
```

```
Times::Times(Exp* l, Exp* r): BinExp(l, r) {}
```

```
double Times::eval() const {  
    return  
        this->left->eval() *  
        this->right->eval();  
}
```

Beobachtung: `Additon::eval()` und `Times::eval()` sind sich sehr ähnlich und könnten ebenfalls zusammengelegt werden. Das dafür notwendige Konzept der *funktionalen Programmierung* geht jedoch über diesen Kurs hinaus.

Weitere Ausdrücke und Operationen

- Weitere Ausdrücke, als von **Exp** abgeleitete Klassen, sind möglich, z.B. $-$, $/$, $\sqrt{\quad}$, \cos , \log

Weitere Ausdrücke und Operationen

- Weitere Ausdrücke, als von **Exp** abgeleitete Klassen, sind möglich, z.B. $-$, $/$, $\sqrt{\quad}$, \cos , \log
- Eine ehemalige Bonusaufgabe (Teil der heutigen Vorlesungsbeispiele auf Code Expert) veranschaulicht, was alles möglich ist: Variablen, trigonometrische Funktionen, Parsing, Pretty-Printing, numerische Vereinfachungen, symbolische Ableitungen, ...

Mission: Monolithisch \rightarrow modular \checkmark

```
struct tnode {  
    char op;  
    double val;  
    tnode* left;  
    tnode* right;  
    ...  
}
```

```
double eval(const tnode* n) {  
    if (n->op == '=') return n->val;  
    double l = 0;  
    if (n->left != 0) l = eval(n->left);  
    double r = eval(n->right);  
    switch(n->op) {  
        case '+': return l + r;  
        case '*': return l * r;  
        case '-': return l - r;  
        case '/': return l / r;  
        default:  
            // unknown operator  
            assert (false);  
    }  
}
```

```
int size (const tnode* n) const { ... }
```

```
...
```

```
struct Literal : public Exp {  
    double val;  
    ...  
    double eval() const {  
        return val;  
    }  
};
```

```
struct Addition : public Exp {  
    ...  
    double eval() const {  
        return left->eval() + right->eval();  
    }  
};
```

```
struct Times : public Exp {  
    ...  
    double eval() const {  
        return left->eval() * right->eval();  
    }  
};
```

```
struct Cos : public Exp {  
    ...  
    double eval() const {  
        return std::cos(argument->eval());  
    }  
};
```



Es gibt noch so viel mehr ...

Nicht gezeigt/besprochen:

- Private Vererbung (`class B : public A`)
- Subtyping und Polymorphie ohne Zeiger
- Nicht-virtuelle Memberfunktionen und statische Bindung (~~`virtual`~~ `double eval()`)
- Überschreiben geerbter Memberfunktionen und Aufrufen der überschriebenen Implementierung
- Mehrfachvererbung (multiple inheritance)
- ...

Objektorientierte Programmierung

Im letzten Kursdrittel wurden einige Konzepte der *objektorientierten Programmierung* vorgestellt, die auf den kommenden Folien noch einmal kurz zusammengefasst werden.

Kapselung (Wochen 10-13):

- Verbergen der Implementierungsdetails von Typen (privater Bereich) vor Benutzern
- Definition einer Schnittstelle (öffentlicher Bereich) zum kontrollierten Zugriff auf Werte und Funktionalität
- Ermöglicht das Sicherstellen von Invarianten, sowie den Austausch von Implementierungen ohne Anpassungen von Benutzercode

Objektorientierte Programmierung

Subtyping (Woche 14):

- Typhierarchien mit Super- und Subtypen können angelegt werden um Verwandtschaftbeziehungen sowie Abstraktionen und Spezialisierungen zu modellieren
- Ein Subtyp unterstützen mindestens die Funktionalität, die auch der Supertyp unterstützt – i.d.R. aber mehr, d.h. Subtypen erweitern die Schnittstelle (den öffentlichen Bereich) ihrer Supertypen
- Daher können Subtypen überall dort eingesetzt werden, wo Supertypen verlangt sind ...
- ... und Funktionen, die auf abstrakteren Typen (Supertypen) operieren können, können auch auf spezialisierteren Typen (Subtypen) operieren
- Die in Woche 7 vorgestellten Streams bilden eine solche Typhierarchie: **ostream** ist der abstrakte Supertyp, **ofstream** etc. sind spezialisierte Subtypen

Objektorientierte Programmierung

Polymorphie und *dynamische Bindung* (Woche 14):

- Ein Zeiger vom statischen Typ T_1 kann zur Laufzeit auf Objekte vom (dynamischen) Typ T_2 zeigen, falls T_2 ein Subtyp von T_1 ist
- Wird eine virtuelle Memberfunktion von einem solchen Zeiger aus aufgerufen, so entscheidet der dynamische Typ darüber, welche Funktion ausgeführt wird
- D.h.: Trotz gleichem statischen Typ kann beim Zugriff auf eine gemeinsame Schnittstelle (Memberfunktionen) eines solchen Zeigers ein anderes Verhalten auftreten
- Zusammen mit Subtyping ermöglicht es dies, neue konkrete Typen (Streams, Ausdrücke, ...) zu einem bestehenden System hinzuzufügen, ohne dieses abändern zu müssen

Objektorientierte Programmierung

Vererbung (Woche 14):

- Abgeleitete Klassen erben die Funktionalität, d.h. die Implementierungen von Memberfunktionen, ihrer Elternklassen
- Dies ermöglicht es, gemeinsam genutzten Code wiederverwenden zu können und vermeidet so Codeduplikation
- Geerbte Implementierungen können auch überschrieben werden, um zu erreichen, dass eine abgeleitete Klasse sich anders verhält als ihre Elternklasse (im Kurs nicht gezeigt)

25. Zusammenfassung

Zweck und Format

Nennung der wichtigsten Stichwörter zu den Kapiteln.
Checkliste: „kann ich mit jedem Begriff etwas anfangen?“

- Ⓜ Motivation: Motivierendes Beispiel zum Kapitel
- Ⓚ Konzepte: Konzepte, die nicht von der Implementation (Sprache) C++abhängen
- Ⓢ Sprachlich (C++): alles was mit der gewählten Sprache zusammenhängt
- Ⓟ Beispiele: genannte Beispiele der Vorlesung

Kapitelüberblick

- 1. Einführung
- 2. Ganze Zahlen
- 3. Wahrheitswerte
- 4. Defensives Programmieren
- 5./6. Kontrollanweisungen
- 7./8. Fließkommazahlen
- 9./10. Funktionen
- 11. Referenztypen
- 12./13. Vektoren und Strings
- 14./15. Rekursion
- 16. Structs und Overloading
- 17. Klassen
- 18./19. Dynamische Datenstrukturen
- 20. Container, Iteratoren und Algorithmen
- 21. Dynamische Datentypen und Speicherverwaltung
- 22. Subtyping, Polymorphie und Vererbung

1. Einführung

- (M) Euklidischer Algorithmus
- (K) Algorithmus, Turingmaschine, Programmiersprachen, Kompilation, Syntax und Semantik
- Werte und Effekte, (Fundamental)typen, Literale, Variablen, Bezeichner, Objekte, Ausdrücke, Operatoren, Anweisungen
- (S) Include-Direktiven `#include <iostream>`
- Hauptfunktion `int main(){...}`
- Kommentare, Layout `// Kommentar`
- Typen, Variablen, L-Wert `a` , R-Wert `a+b`
- Ausdrucksanweisung `b=b*b;` , Deklarationsanweisung `int a;`, Rückgabeeanweisung `return 0;`

2. Ganze Zahlen

- Celsius to Fahrenheit
 - Assoziativität und Präzedenz, Stelligkeit
 - Ausdrucksbäume, Auswertungsreihenfolge
 - Arithmetische Operatoren
 - Binärzahldarstellung, Hexadezimale Zahlen, Wertebereich
 - Zahlendarstellung mit Vorzeichen, Zweierkomplement
- Arithmetische Operatoren `9 * celsius / 5 + 32`
 - Inkrement / Dekrement `expr++`
 - Arithmetische Zuweisungen `expr1 += expr2`
 - Konversion `int` ↔ `unsigned int`
- Celsius to Fahrenheit, Ersatzwiderstand

3. Wahrheitswerte

- Ⓚ
 - Boole'sche Funktionen, Vollständigkeit
 - DeMorgan'sche Regeln
- Ⓢ
 - Der Typ `bool`
 - Logische Operationen `a && !b`
 - Relationale Operationen `x < y`
 - Präzedenzen `7 + x < y && y != 3 * z`
 - Kurzschlussauswertung `x != 0 && z / x > y`
 - Die `assert`-Anweisung, `#include <cassert>`
- ⓑ
 - Div-Mod Identität.

4. Defensives Programmieren

- Ⓚ ■ Assertions und Konstanten
- Ⓢ ■ Die `assert`-Anweisung, `#include <cassert>`
■ `const int speed_of_light=2999792458`
- Ⓟ ■ Assertions für den GGT

5./6. Kontrollanweisungen

- (M) Linearer Kontrollfluss vs. interessante Programme, Spaghetti-Code
- (K) Auswahlanweisungen, Iterationsanweisungen
 - (Vermeidung von) Endlosschleifen, Halteproblem
 - Sichtbarkeits- und Gültigkeitsbereich, Automatische Speicherdauer
 - Äquivalenz von Iterationsanweisungen
- (S) if Anweisungen `if (a % 2 == 0) {..}`
 - for Anweisungen `for (unsigned int i = 1; i <= n; ++i) ...`
 - while und do-Anweisungen `while (n > 1) {...}`
 - Blöcke, Sprunganweisungen `if (a < 0) continue;`
 - Switch Anweisung `switch(grade) {case 6: }`
- (B) Summenberechnung (Gauss), Primzahltest, Collatz-Folge, Fibonacci Zahlen, Taschenrechner, Notenausgabe

7./8. Fließkommazahlen

- (M) Richtig Rechnen: Celsius / Fahrenheit
- (K) Fixkomma- vs. Fließkommazahldarstellung
 - (Löcher im) Wertebereich
 - Rechnen mit Fließkommazahlen, Umrechnung
 - Fließkommazahlensysteme, Normalisierung, IEEE Standard 754
 - *Richtlinien für das Rechnen mit Fließkommazahlen*
- (S) Typen `float`, `double`
 - Fließkommaliterale `1.23e-7f`
- (B) Celsius/Fahrenheit, Euler, Harmonische Zahlen

9./10. Funktionen

- Ⓜ ■ Potenzberechnung
- Ⓚ ■ Kapselung von Funktionalität
 - Funktionen, formale Argumente, Aufrufargumente
 - Gültigkeitsbereich, Vorwärts-Deklaration
 - Prozedurales Programmieren, Modularisierung, Getrennte Übersetzung
 - *Stepwise Refinement*
- Ⓢ ■ Funktionsdeklaration, -definition
 - `double pow(double b, int e){ ... }`
 - Funktionsaufruf `pow (2.0, -2)`
 - Der typ `void`
- Ⓟ ■ Potenzberechnung, perfekte Zahlen, Minimum, Kalender

11. Referenztypen

- Ⓜ ■ Funktion Swap
- Ⓚ ■ Werte-/ Referenzsemantik, Pass by Value / Pass by Reference, Return by Reference
 - Lebensdauer von Objekten / Temporäre Objekte
 - Konstanten
- Ⓢ ■ Referenztyp `int& a`
 - Call by Reference und Return by Reference `int& increment (int& i)`
 - Const-Richtlinie, Const-Referenzen, Referenzrichtlinie
- Ⓟ ■ Swap, Inkrement

12./13. Vektoren und Strings

- Ⓜ
 - Iteration über Daten: Sieb des Eratosthenes
- Ⓚ
 - Vektoren, Speicherlayout, Wahlfreier Zugriff
 - (Fehlende) Grenzenprüfung
 - Vektoren
 - Zeichen: ASCII, UTF8, Texte, Strings
- Ⓢ
 - Vektor Typen `std::vector<int> a {4,3,5,2,1};`
 - Zeichen und Texte, der Typ `char c = 'a';`, Konversion nach `int`
 - Vektoren von Vektoren
 - Ströme `std::istream`, `std::ostream`
- Ⓟ
 - Sieb des Eratosthenes, Caesar-Code, Kürzeste Wege

14./15. Rekursion

- Ⓜ
 - Rekursive math. Funktionen, Das n-Queen Problem, , Lindenmayer-Systeme, Kommandozeilenrechner
- Ⓚ
 - Rekursion
 - Aufrufstapel, Gedächtnis der Rekursion
 - Korrektheit, Terminierung,
 - Rekursion vs. Iteration
 - Backtracking, EBNF, Formale Grammatiken, Parsen
- Ⓟ
 - Fakultät, GGT, Sudoku-Löser, Taschenrechner

16. Structs und Overloading

- ① ■ Datentyp Rationale Zahlen selber bauen
- ② ■ Heterogene Datenstruktur
 - Funktions- und Operator-Overloading
 - Datenkapselung
- ③ ■ Struct Definition `struct rational {int n; int d;};`
 - Mitgliedszugriff `result.n = a.n * b.d + a.d * b.n;`
 - Initialisierung und Zuweisung,
 - Überladen von Funktionen `pow(2)` vs. `pow(3,3)`; , Überladen von Operatoren
- ④ ■ rationale Zahlen, komplexe Zahlen

17. Klassen

- (M) Rationale Zahlen mit Kapselung
- (K) Kapselung, Konstruktion, Mitgliedsfunktionen
- (S) Klassen `class rational { ... };`
 - Zugriffssteuerung `public: / private:`
 - Mitgliedsfunktionen `int rational::denominator () const`
 - Das implizite Argument der Memberfunktionen
- (B) Endlicher Ring, Komplexe Zahlen

18./19. Dynamische Datenstrukturen

- ① ■ Unser eigener Vektor
- ② ■ Allokation, Zeiger-Typen, Verkettete Liste, Allokation, Deallokation, Dynamischer Datentyp
- ③ ■ Die **new** Anweisung
 - Zeiger `int* x;`, Nullzeiger `nullptr.`
 - Adress-, Dereferenzoperator `int *ip = &i; int j = *ip;`
 - Zeiger und Const `const int *a;`
- ④ ■ Verkettete Liste, Stack

20. Container, Iteratoren und Algorithmen

- Ⓜ
 - Vektoren sind Container
- Ⓚ
 - Iterieren mit Zeigern
 - Container und Iteratoren
 - Algorithmen
- Ⓢ
 - Iteratoren `std::vector<int>::iterator`
 - Algorithmen der Standardbibliothek `std::fill (a, a+5, 1);`
 - Einen Iterator implementieren
 - Iteratoren und `const`
- Ⓟ
 - Ausgeben eines Vektors, einer Menge

21. Dynamische Datentypen und Speicherverwaltung

- Ⓜ
 - Stack
 - Ausdrucksbaum
- Ⓚ
 - Richtlinie „Dynamischer Speicher“
 - Gemeinsamer Zeiger-Zugriff
 - Dynamischer Datentyp
 - Baumstruktur
- Ⓢ
 - **new** und **delete**
 - Desktruktor `stack::~~stack()`
 - Kopierkonstruktor `stack::stack(const stack& s)`
 - Zuweisungsoperator
`stack& stack::operator=(const stack& s)`
 - Dreierregel
- Ⓟ
 - Binärer Suchbaum

22. Subtyping, Polymorphie und Vererbung

- ① ■ Erweitern und Verallgemeinern von Ausdrucksbäumen
- ② ■ Subtyping
 - Polymorphie und dynamische Bindung
 - Vererbung
- ③ ■ Basisklasse `struct Exp{}`
 - Abgeleitete Klasse `struct BinExp: public Exp{}`
 - Abstrakte Klasse
 - `struct Exp{virtual int size() const = 0...}`
 - Polymorphie `virtual double eval()`
- ④ ■ Ausdrucksknoten und Erweiterungen

Ende

Ende der Vorlesung.