Informatik I

Vorlesung am D-ITET der ETH Zürich

Felix Friedrich

HS 2017

Willkommen

zur Vorlesung Informatik I !

am ITET Department der ETH Zürich.

Ort und Zeit:

Mittwoch 8:15 - 10:00, ETF E1.
Pause 9:00 - 9:15, leichte Verschiebung möglich.

Vorlesungshomepage

`http://lec.inf.ethz.ch/itet/informatik1`

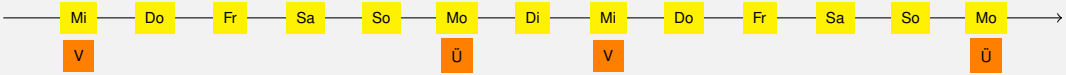
Team

Chefassistent	Martin Bättig	
Assistenten	Ivana Unkovic	Francois Serre
	Hossein Shafagh	Marc Bitterli
	Christoph Amevor	Temmy Bounedjar
	Michael Prasthofer	Sean Bone
	Patrik Hadorn	Nathaneal Köhler
	Robin Worreby	Alexander Hedges
	Christelle Gloor	Yvan Bosshard
	Alessio Bähler	
Dozent	FF	

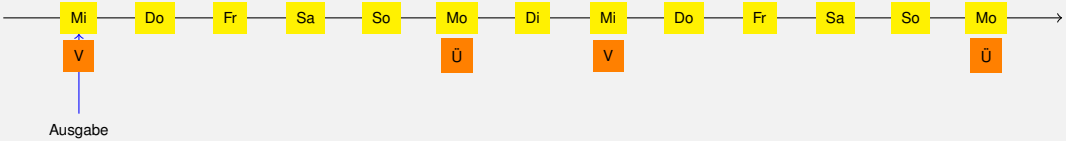
Einschreibung in Übungsgruppen

- Gruppeneinteilung selbstständig via Webseite `http://echo.ethz.ch`
- Funktioniert nach Belegung dieser Vorlesung in myStudies.
- Die gezeigten Räume und Termine abhängig vom Studiengang.

Ablauf

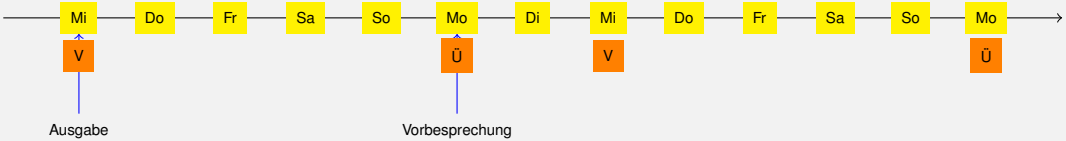


Ablauf



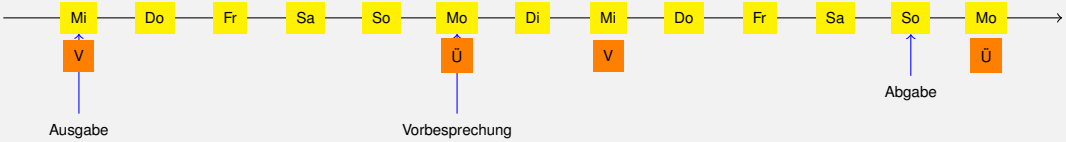
- Übungsblattausgabe zur Vorlesung (online).
- Vorbesprechung in der folgenden Übung.
- Bearbeitung der Übung bis spätestens am Tag vor der nächsten Übung (23:59h).
- Nachbesprechung der Übung am Montag. Feeback zu den Abgaben innerhalb einer Woche nach Nachbesprechung.

Ablauf



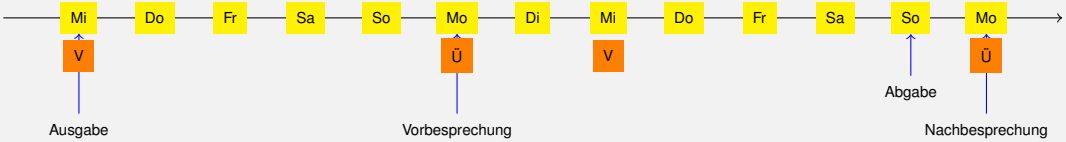
- Übungsblattausgabe zur Vorlesung (online).
- Vorbesprechung in der folgenden Übung.
- Bearbeitung der Übung bis spätestens am Tag vor der nächsten Übung (23:59h).
- Nachbesprechung der Übung am Montag. Feeback zu den Abgaben innerhalb einer Woche nach Nachbesprechung.

Ablauf



- Übungsblattausgabe zur Vorlesung (online).
- Vorbesprechung in der folgenden Übung.
- Bearbeitung der Übung bis spätestens am Tag vor der nächsten Übung (23:59h).
- Nachbesprechung der Übung am Montag. Feeback zu den Abgaben innerhalb einer Woche nach Nachbesprechung.

Ablauf



- Übungsblattausgabe zur Vorlesung (online).
- Vorbesprechung in der folgenden Übung.
- Bearbeitung der Übung bis spätestens am Tag vor der nächsten Übung (23:59h).
- Nachbesprechung der Übung am Montag. Feeback zu den Abgaben innerhalb einer Woche nach Nachbesprechung.

Zu den Übungen

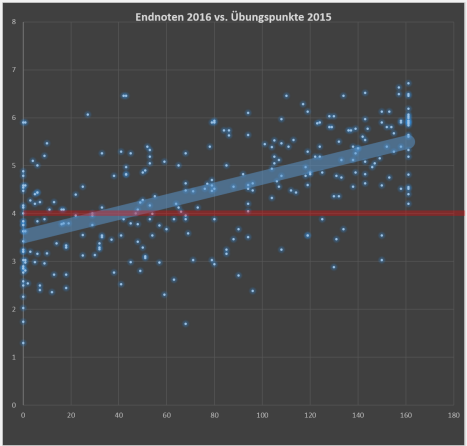
- An der ETH ist (seit HS 2013) für die Prüfungszulassung kein Testat erforderlich.

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!

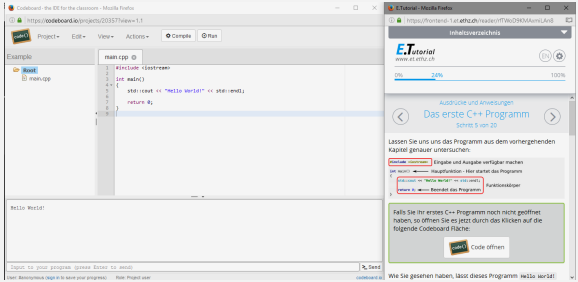


Fehlende Resourcen sind keine Entschuldigung!

Für die Übungen verwenden wir eine Online-Entwicklungsumgebung, benötigt lediglich einen Browser, Internetverbindung und Ihr ETH Login.

Falls Sie keinen Zugang zu einem Computer haben: in der ETH stehen an vielen Orten öffentlich Computer bereit.

Online Tutorial

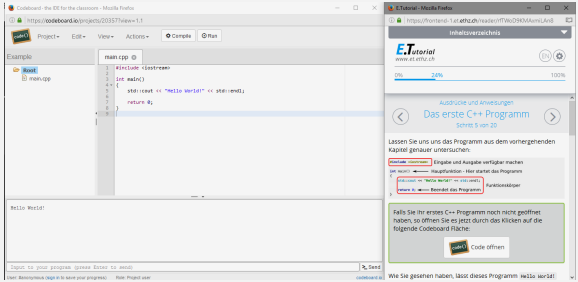


Zum Einstieg stellen wir ein *Online-C++ Tutorial* zur Verfügung.

Ziel: Ausgleich der unterschiedlichen Programmierkenntnisse.

Schriftlicher Minitest zur *Selbsteinschätzung* in der ersten Übungsstunde.

Online Tutorial

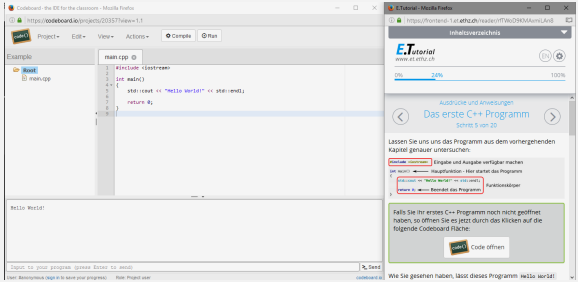


Zum Einstieg stellen wir ein *Online-C++ Tutorial* zur Verfügung.

Ziel: Ausgleich der unterschiedlichen Programmierkenntnisse.

Schriftlicher Minitest zur *Selbsteinschätzung* in der ersten Übungsstunde.

Online Tutorial



Zum Einstieg stellen wir ein *Online-C++ Tutorial* zur Verfügung.

Ziel: Ausgleich der unterschiedlichen Programmierkenntnisse.

Schriftlicher Minitest zur *Selbsteinschätzung* in der ersten Übungsstunde.

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 sind keine Hilfsmittel zugelassen.

Es wird sowohl praktisches Wissen (Programmierfähigkeit[1]) als auch theoretisches Wissen (Hintergründe, Systematik) geprüft.

[1] soweit in schriftlicher Prüfung möglich

Unser Angebot

- 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 über alle Serien, wobei volle Punktzahl einem Bonus von 0.25 entspricht.

Akademische Lauterkeit

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

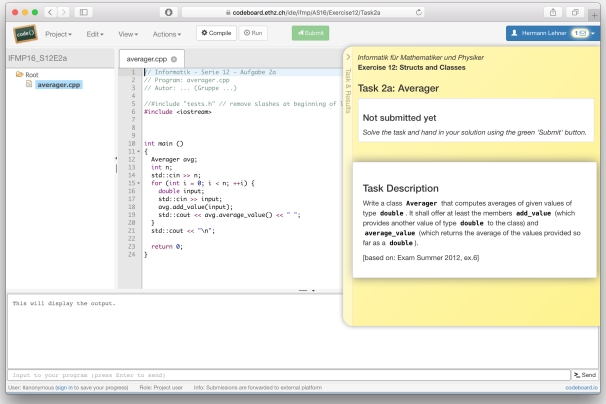
Wir prüfen das (zum Teil automatisiert) nach und behalten uns insbesondere mündliche Prüfgespräche vor.

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

Codeboard

Codeboard ist eine Online-IDE: Programmieren im Browser!

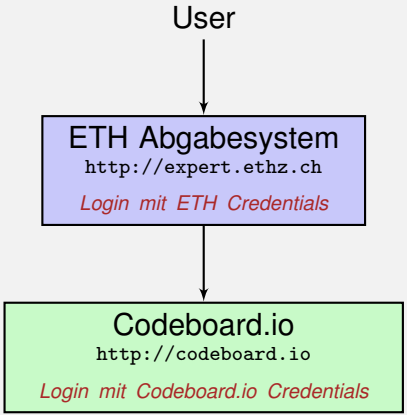
- Falls vorhanden, bringen Sie Ihren Laptop/Tablet/... mit in den Unterricht.
- Sie können direkt in der Vorlesung Beispiele ausprobieren, ohne dass Sie irgendwelche Tools installieren müssen.



Expert

Unser Übungssystem besteht aus zwei unabhängigen Systemen, die miteinander kommunizieren:

- **Das ETH Abgabesystem:** Ermöglicht es uns, Ihre Aufgaben zu bewerten
- **Die Online IDE:** Die Programmierumgebung



Übungseinschreibung

Codeboard.io Registrierung

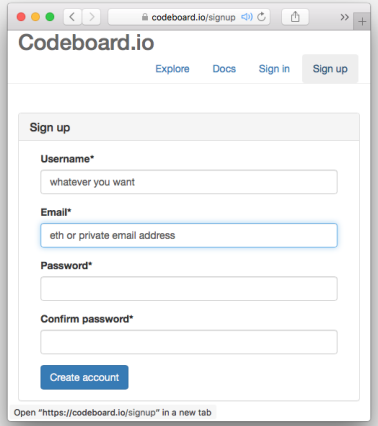
Gehen Sie auf `http://codeboard.io` und erstellen Sie dort ein Konto, bleiben Sie am besten eingeloggt.

Einschreibung in Übungsgruppen

Gehen Sie auf `http://expert.ethz.ch/ifee1y17e01t1` und schreiben Sie sich dort in eine Übungsgruppe ein.

Codeboard.io Registrierung

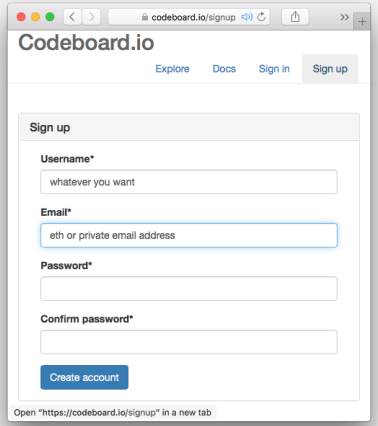
Falls Sie noch keinen **Codeboard.io** Account haben ...



- Wir verwenden die Online IDE **Codeboard.io**

Codeboard.io Registrierung

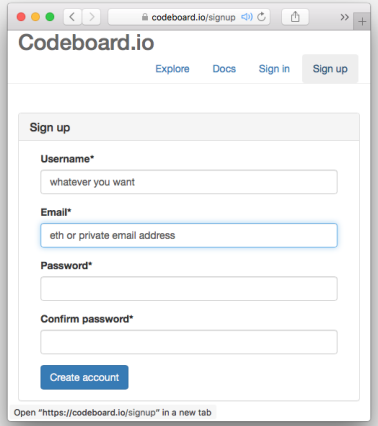
Falls Sie noch keinen **Codeboard.io** Account haben ...



- Wir verwenden die Online IDE **Codeboard.io**
- Erstellen Sie dort einen Account, um Ihren Fortschritt abzuspeichern und später Submissions anzuschauen

Codeboard.io Registrierung

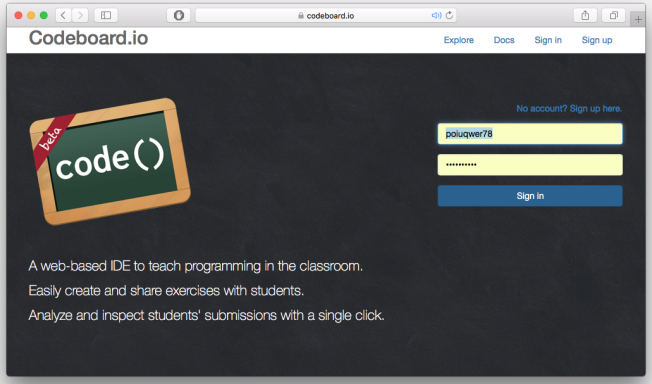
Falls Sie noch keinen **Codeboard.io** Account haben ...



- Wir verwenden die Online IDE **Codeboard.io**
- Erstellen Sie dort einen Account, um Ihren Fortschritt abzuspeichern und später Submissions anzuschauen
- Anmeldedaten können beliebig gewählt werden! *Verwenden Sie nicht das ETH Passwort.*

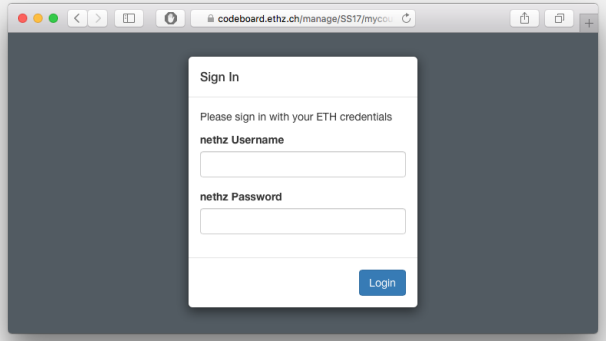
Codeboard.io Login

Falls Sie schon einen Account haben, loggen Sie sich ein:



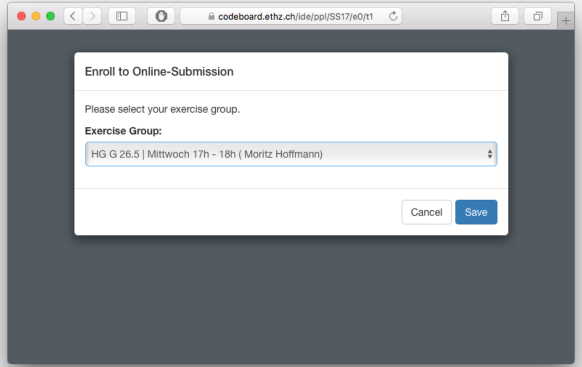
Einschreibung in Übungsgruppen - I

- Besuchen Sie `http://expert.ethz.ch/ifee1y17e01t1`
- Loggen Sie sich mit Ihrem nethz Account ein.



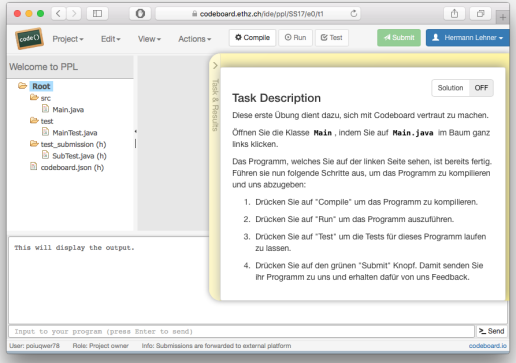
Einschreibung in Übungsgruppen - II

Schreiben Sie sich in diesem Dialog in eine Übungsgruppe ein.



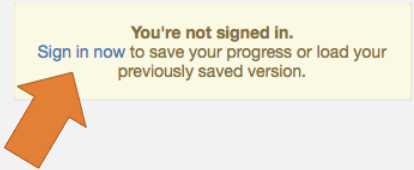
Die erste Übung

Sie sind nun eingeschrieben und die erste Übung ist geladen.
Folgen Sie den Anweisungen in der gelben Box.



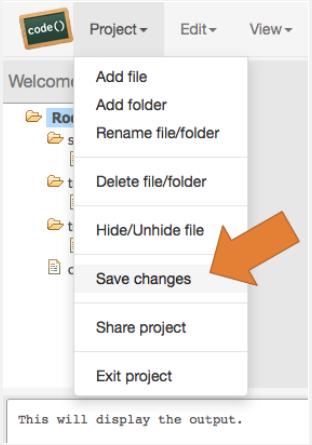
Die erste Übung - Codeboard.io Login

Achtung! Falls Sie diese Nachricht sehen, klicken Sie auf Sign in now und melden Sie sich dort mit Ihrem **Codeboard.io** Account ein.



Die erste Übung - Fortschritt speichern!

Achtung! Speichern Sie Ihren Fortschritt regelmässig ab. So können Sie jederzeit an einem anderen Ort weiterarbeiten.



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 $\neq$ Computer Science

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

Mike Fellows, US-Informatiker (1991)

Computer Science $\subseteq$ Informatik

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

Computer Science $\subseteq$ Informatik

- 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*

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

Informatik $\neq$ EDV-Kenntnisse

Informatik: *Grundlagenwissen*

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

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.)

- 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.

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 $b \neq 0$

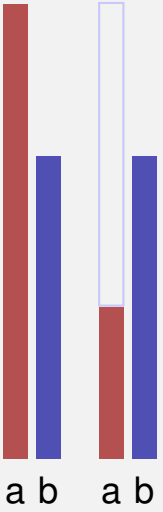
Wenn $a > b$ dann

$a \leftarrow a - b$

Sonst:

$b \leftarrow b - a$

Ergebnis: a.



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 $b \neq 0$

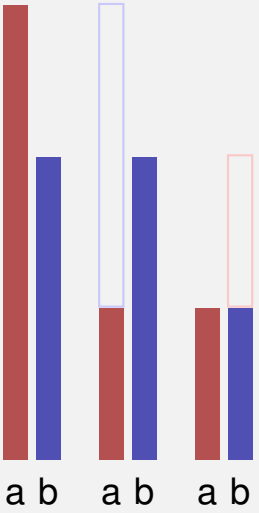
Wenn $a > b$ dann

$a \leftarrow a - b$

Sonst:

$b \leftarrow b - a$

Ergebnis: a.



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 $b \neq 0$

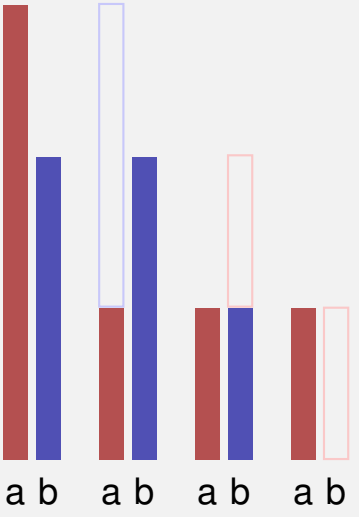
Wenn $a > b$ dann

$a \leftarrow a - b$

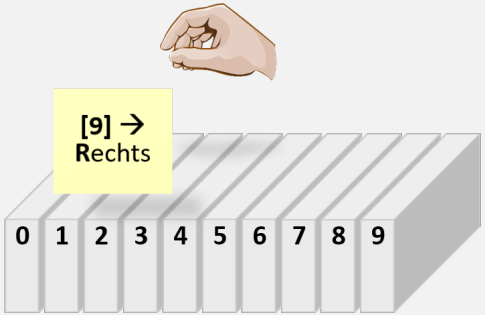
Sonst:

$b \leftarrow b - a$

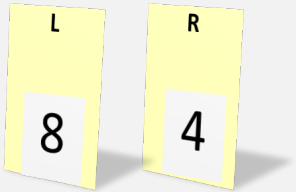
Ergebnis: a.



Live Demo: Turing Maschine



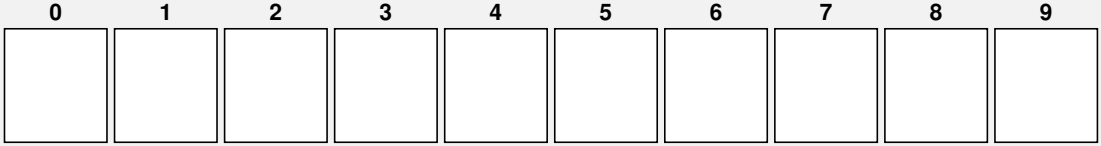
Speicher

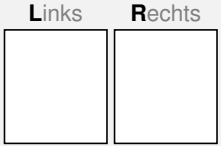


Register

Euklid in der Box

Speicher

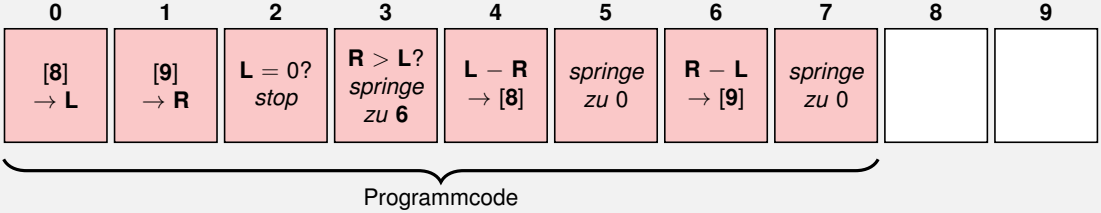


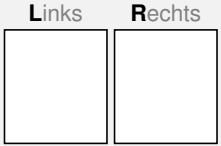


Register

Euklid in der Box

Speicher

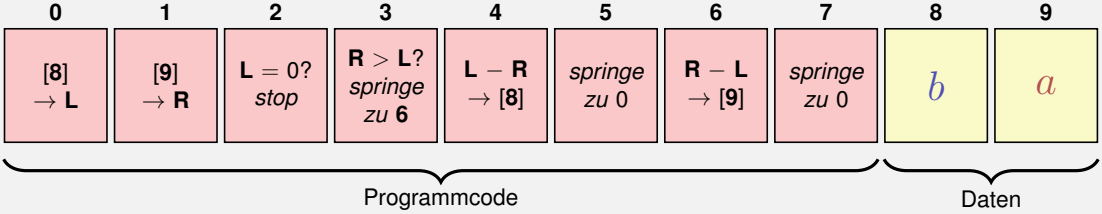


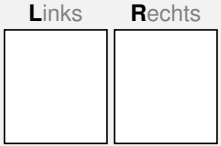


Register

Euklid in der Box

Speicher

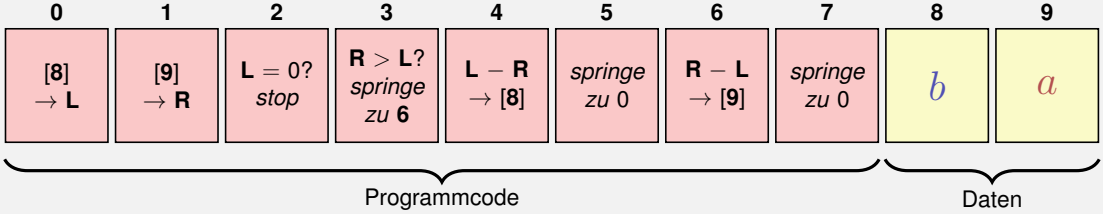


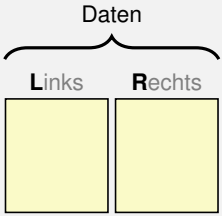


Register

Euklid in der Box

Speicher





Register

Euklid in der Box

Speicher

0	1	2	3	4	5	6	7	8	9
[8] → **L**	[9] → **R**	**L** = 0? *stop*	**R** > **L**? *springe zu* **6**	**L** − **R** → [8]	*springe zu* 0	**R** − **L** → [9]	*springe zu* 0	b	a

Links	**R**echts
b	a

Register

Solange $b \neq 0$

Wenn $a > b$ dann

$a \leftarrow a - b$

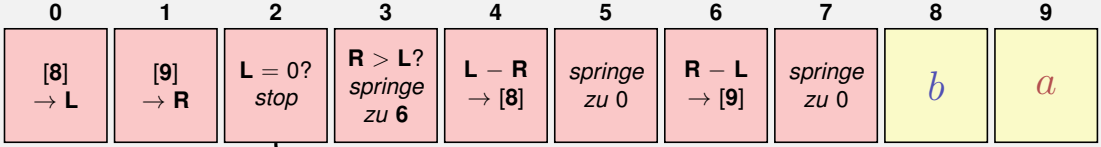
Sonst:

$b \leftarrow b - a$

Ergebnis: a.

Euklid in der Box

Speicher



Solange $b \neq 0$

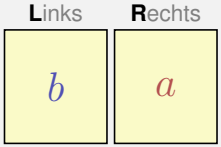
Wenn $a > b$ dann

$a \leftarrow a - b$

Sonst:

$b \leftarrow b - a$

Ergebnis: a.



Register

Euklid in der Box

Speicher

0	1	2	3	4	5	6	7	8	9
[8] → **L**	[9] → **R**	**L** $= 0$? *stop*	**R** $>$ **L**? *springe zu* **6**	**L** $-$ **R** → [8]	*springe zu* 0	**R** $-$ **L** → [9]	*springe zu* 0	b	a

Solange $b \neq 0$

Wenn $a > b$ dann

$a \leftarrow a - b$

Sonst:

$b \leftarrow b - a$

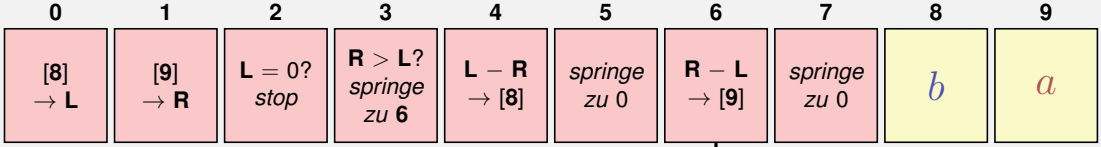
Ergebnis: a.

Links	**R**echts
b	a

Register

Euklid in der Box

Speicher



Solange $b \neq 0$

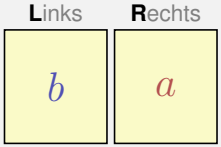
Wenn $a > b$ dann

$a \leftarrow a - b$

Sonst:

$b \leftarrow b - a$

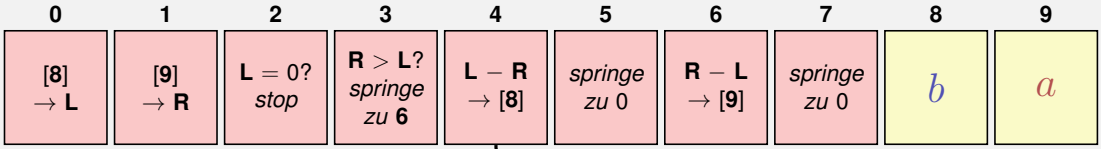
Ergebnis: a.



Register

Euklid in der Box

Speicher



Solange $b \neq 0$

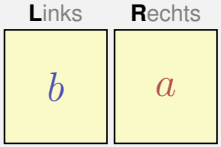
Wenn $a > b$ dann

$a \leftarrow a - b$

Sonst:

$b \leftarrow b - a$

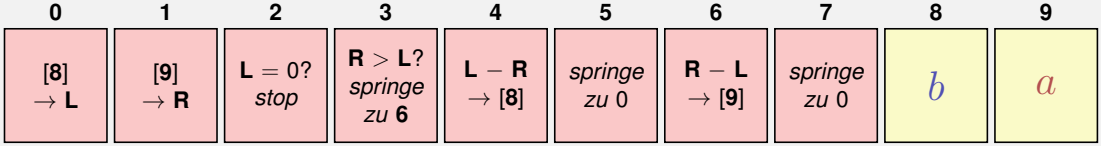
Ergebnis: a.

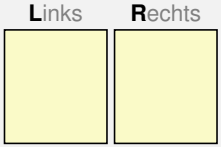


Register

Euklid in der Box

Speicher





Register

Solange $b \neq 0$

Wenn $a > b$ dann

$a \leftarrow a - b$

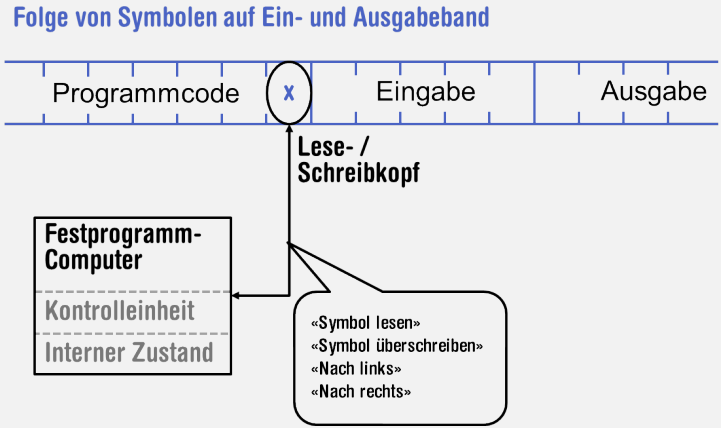
Sonst:

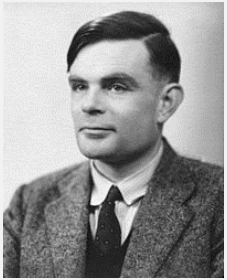
$b \leftarrow b - a$

Ergebnis: a.

Computer – Konzept

Eine geniale Idee: Universelle Turingmaschine (Alan Turing, 1936)



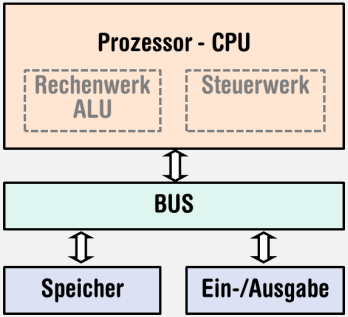


Alan Turing

Computer – Umsetzung

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

Von Neumann Architektur



Konrad Zuse

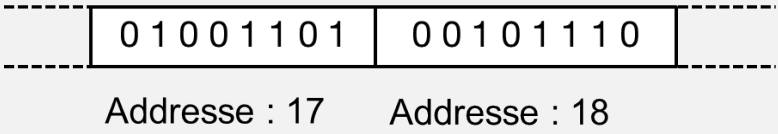
John von Neumann

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.



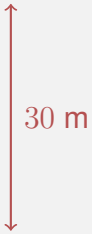
Rechengeschwindigkeit

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

[2] Uniprozessor 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

[2] Uniprozessor Computer bei 1GHz

Rechengeschwindigkeit

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

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

arbeitet ein heutiger Desktop-PC mehr als 100 Millionen Instruktionen ab.[2]

[2] Uniprozessor Computer bei 1GHz

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

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, NZZ Online, 1.9.2017

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!

Programmiersprachen

- Sprache, die der Computer "versteht", ist sehr primitiv (Maschinensprache).
- Einfache Operationen müssen in 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
  $\rightarrow$ Abstraktion!

Warum C++?

Andere populäre höhere Programmiersprachen: Java, C#, Objective-C, Modula, Oberon, Python ...

Warum C++?

Andere populäre höhere Programmiersprachen: Java, C#, Objective-C, Modula, Oberon, Python ...

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

Deutsch vs. C++

Deutsch

Es ist nicht genug zu wissen,
man muss auch anwenden.
(Johann Wolfgang von Goethe)

C++

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

Syntax und Semantik

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

Syntax und Semantik

- Entsprechende Regeln für ein Computerprogramm sind einfacher, aber auch strenger, denn Computer sind vergleichsweise dumm.

Syntax und Semantik von C++

Syntax

- Was *ist* ein C++ Programm?
- Ist es *grammatikalisch* korrekt?

Semantik

- Was *bedeutet* ein Programm?
- Welchen Algorithmus realisiert ein Programm?

Was braucht es zum Programmieren?

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

Was braucht es zum Programmieren?

- **Computer:** Gerät zum Ausführen von Programmen in Maschinensprache
- **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;
}
```

```
// 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;
}
```

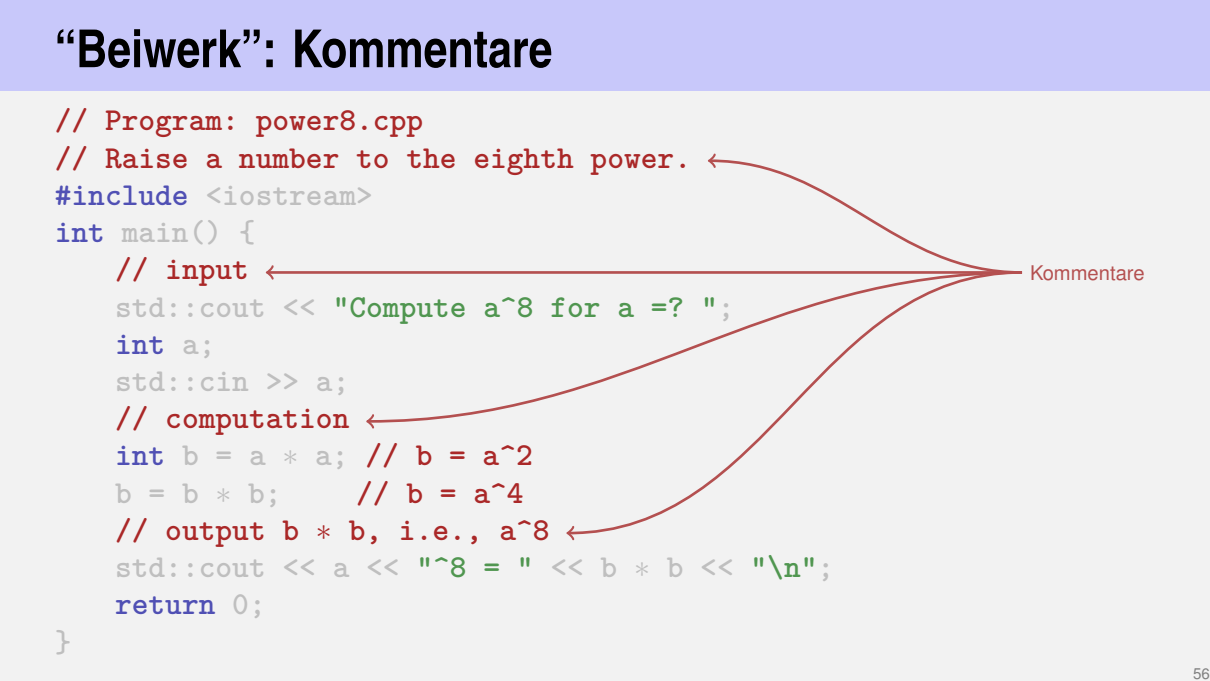
```
// Program: power8.cpp
// Raise a number to the eighth power.
#include <iostream>
int main(){
    // input
    std::cout << "Compute a^8 for a =? ";
    int a;
    std::cin >> a;
    // computation
    int b = a * a; // b = a^2 ←——— Berechne einen Wert (a^2)!
    b = b * b;     // b = a^4
    // 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 und Layout

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;}
```

Kommentare und Layout

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>
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() {
    // 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;
}
```

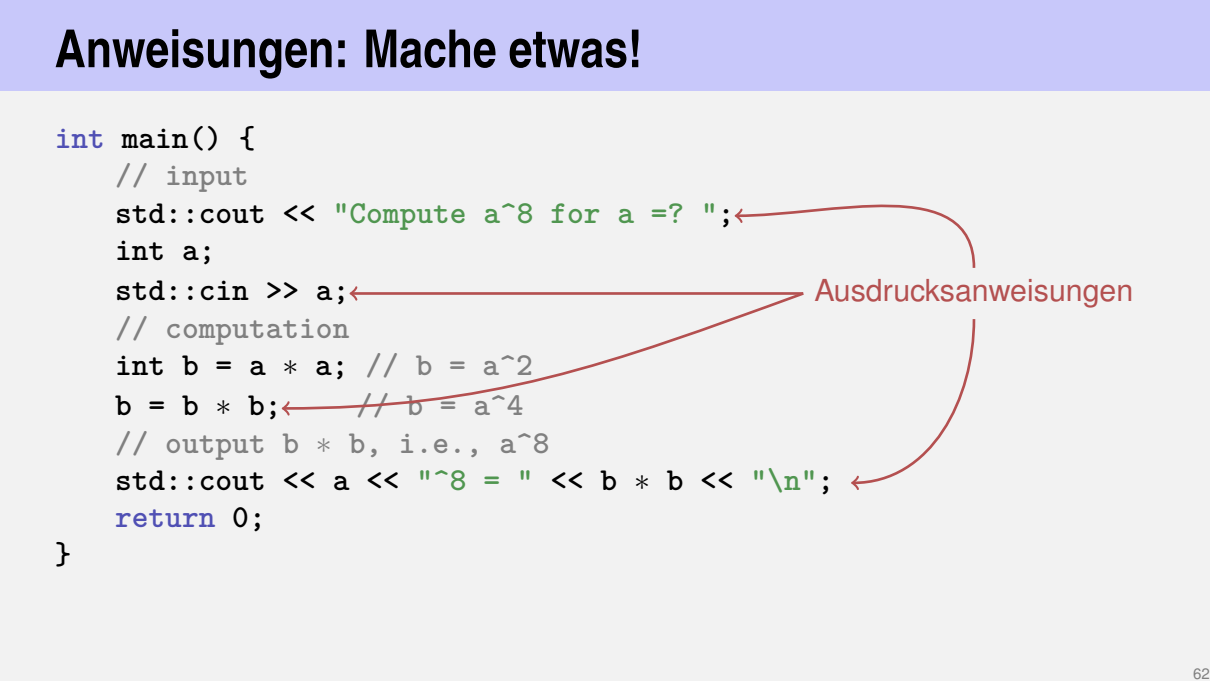
Funktionsdeklaration der main-Funktion

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;
}
```



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 – Effekte

```
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;
}
```

Effekt: Ausgabe des Strings `Compute ...`

Effekt: Eingabe einer Zahl und Speichern in a

Effekt: Speichern des berechneten Wertes von a*a in b

Effekt: Speichern des berechneten Wertes von b*b in b

Effekt: Rückgabe des Wertes 0

Effekt: Ausgabe des Wertes von a und des berechneten Wertes von b*b

Anweisungen – Variablendefinitionen

```
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;
}
```

Deklarationsanweisungen

Typ-namen

Variablen

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

Variablen

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

Beispiel

`int a;` definiert Variable mit

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

Ausdrücke: Berechne einen Wert!

- repräsentieren *Berechnungen*

Ausdrücke: Berechne einen Wert!

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

Ausdrücke: Berechne einen Wert!

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

Ausdrücke: Berechne einen Wert!

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

Ausdrücke: Berechne einen Wert!

- 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!

- 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;
```

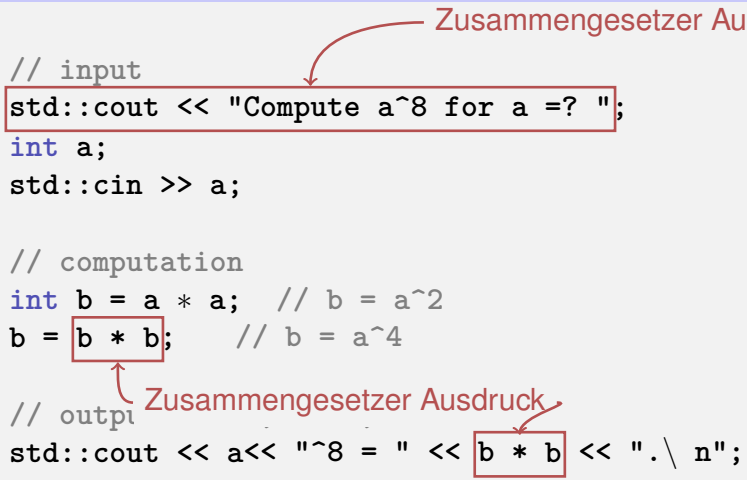
Ausdrücke

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

// 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;
```

Variablenname, primärer Ausdruck (+ Name und Adresse)

Variablenname, primärer Ausdruck (+ Name und Adresse)

Literal, primärer Ausdruck


Zusammengesetzer 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; // b = a^4
Zusammengesetzer Ausdruck
// outpu
std::cout << a<< "^8 = " << b * b << ".\ n";
return 0;

Ausdrücke

Baukasten

```
// 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";

retur
```

↑ Vierfach zusammengesetzter Ausdruck

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

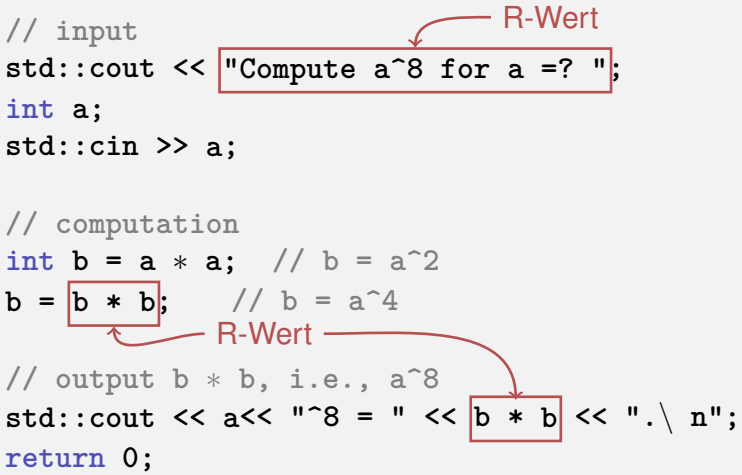
```
// input
std::cout << "Compute a^8 for a =? ";
int a;
std::cin >> a;

// 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-Wert (Ausdruck + Adresse)

L-Wert (Ausdruck + Adresse)

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

L-Werte und R-Werte



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

  Beispiel: Literal `0`

- Jeder L-Wert kann als R-Wert benutzt werden (aber nicht umgekehrt).

L-Werte und R-Werte

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

- Ausdruck der kein L-Wert ist

  Beispiel: Literal `0`

- Jeder L-Wert kann als R-Wert benutzt werden (aber nicht umgekehrt).

  Jedes e-Bike kann als normales Fahrrad benutzt werden, aber nicht umgekehrt.

L-Werte und R-Werte

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

- Ausdruck der kein L-Wert ist

  Beispiel: Literal `0`

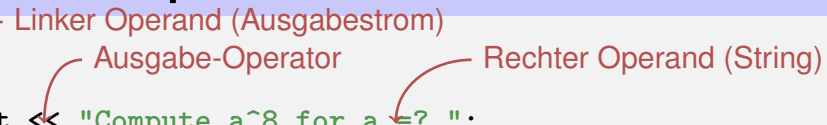
- Jeder L-Wert kann als R-Wert benutzt werden (aber nicht umgekehrt).
- Ein R-Wert kann seinen Wert *nicht ändern*.

Operatoren und Operanden

Baukasten

```
// 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;
```

Operatoren und Operanden




```
// 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;
```

Operatoren und Operanden


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

// compute a^4
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)

Operatoren und Operanden


```
// 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`

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`

- Arithmetischer Ausdruck,

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

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

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

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

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

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

`9 * celsius / 5 + 32`

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

Wie ist der Ausdruck geklammert?

Präzedenz

Punkt vor Strichrechnung

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

bedeutet

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

Präzedenz

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

Regel 3: Stelligkeit

Unäre Operatoren +, − vor binären +, −.

```
-3 - 4
```

bedeutet

```
(-3) - 4
```

Klammerung

Jeder Ausdruck kann mit Hilfe der

- Assoziativitäten
- Präzedenzen
- Stelligkeiten

der beteiligten Operatoren eindeutig geklammert werden.

Ausdrucksbäume

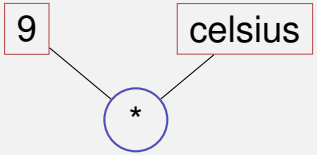
Klammerung ergibt Ausdrucksbaum

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

Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

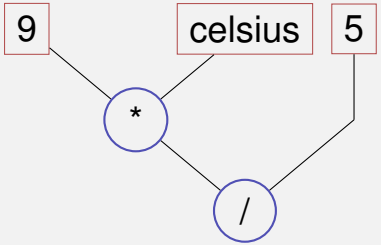
```
(9 * celsius) / 5 + 32
```



Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

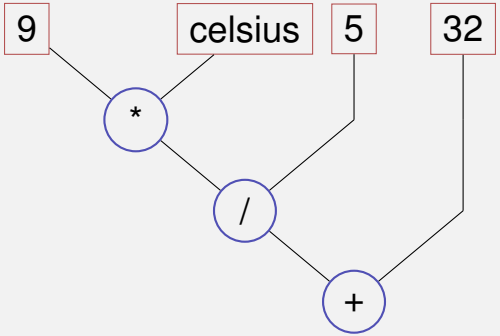
```
((9 * celsius) / 5) + 32
```



Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

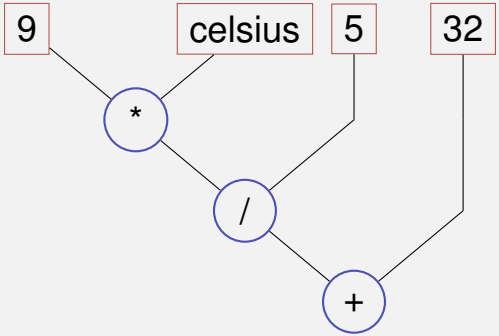
```
(((9 * celsius) / 5) + 32)
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

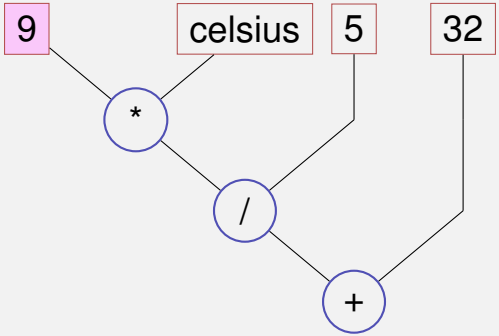
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

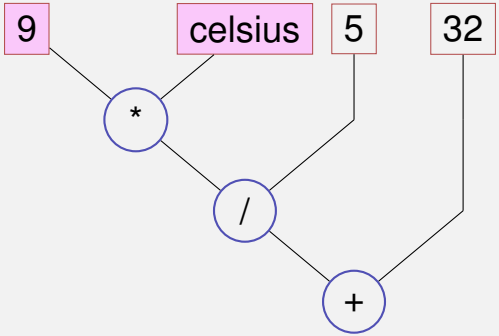
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

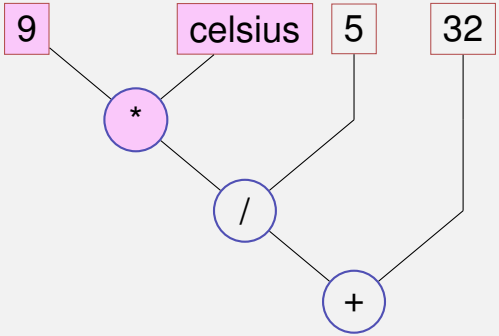
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

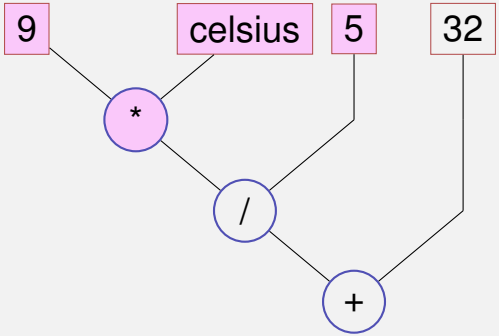
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

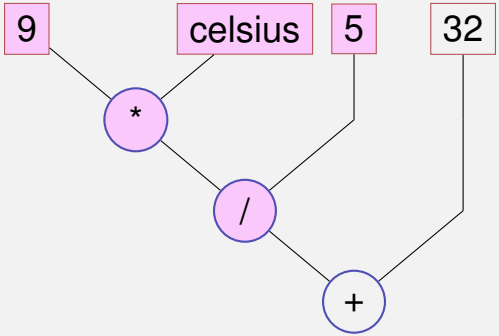
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

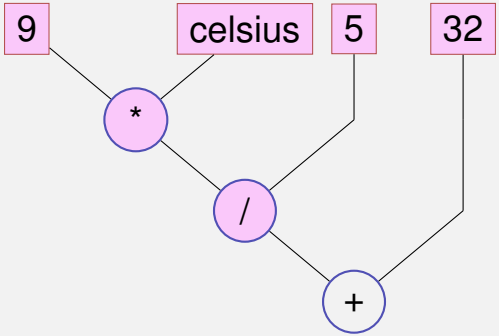
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

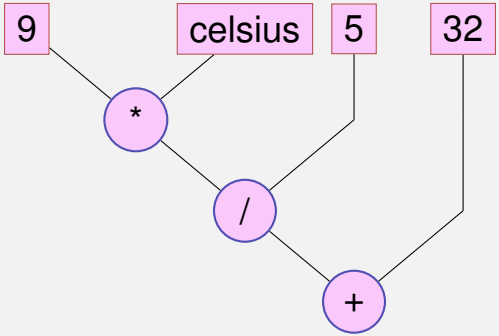
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

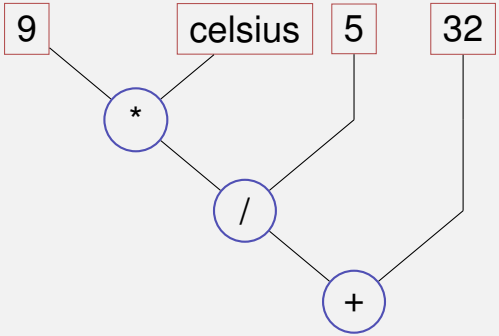
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

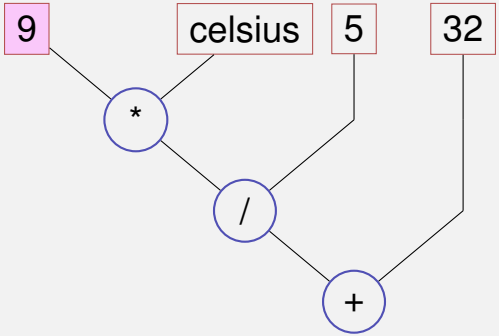
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

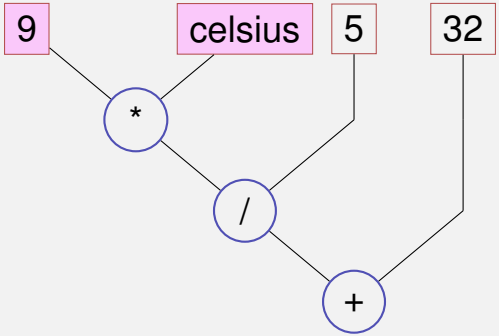
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

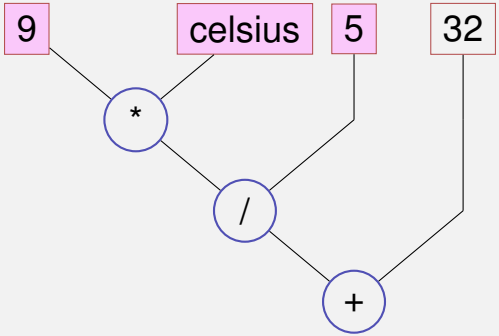
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

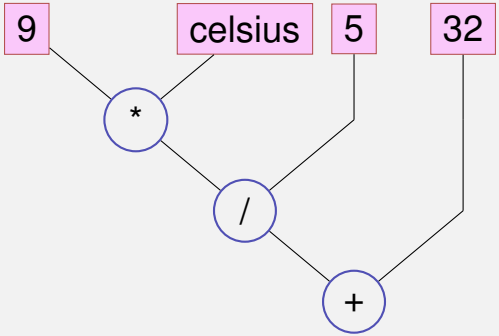
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

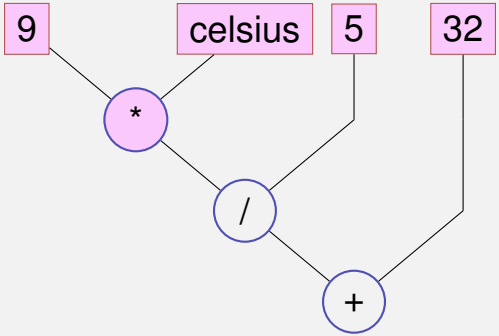
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

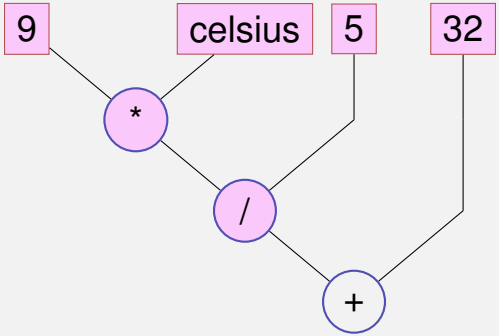
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

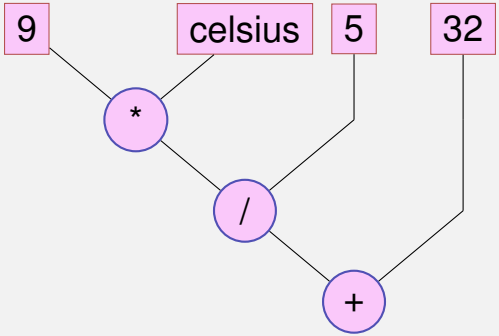
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

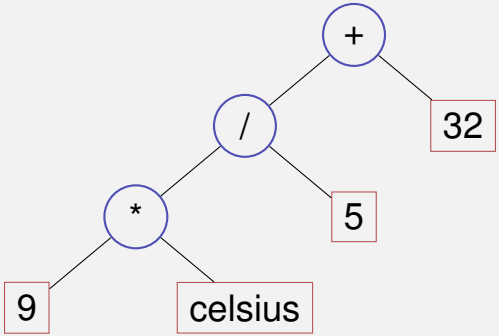
```
9 * celsius / 5 + 32
```



Ausdrucksbäume – Notation

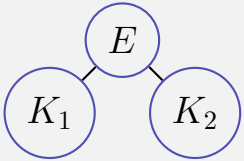
Üblichere Notation: Wurzel oben

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



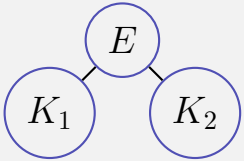
Auswertungsreihenfolge – formaler

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



Auswertungsreihenfolge – formaler

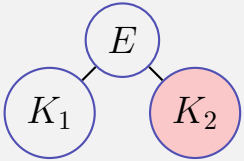
- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

Auswertungsreihenfolge – formaler

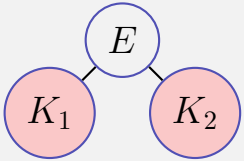
- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

Auswertungsreihenfolge – formaler

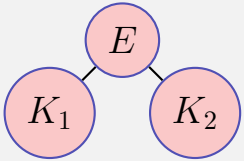
- Gültige Reihenfolge: Jeder Knoten wird erst *nach* seinen Kindern ausgewertet.



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

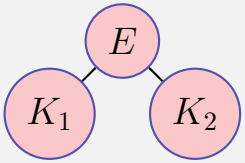
Auswertungsreihenfolge – formaler

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



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

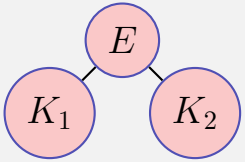
Auswertungsreihenfolge – formaler

- 

C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

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

Auswertungsreihenfolge – formaler

- 

C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

- Beispiel für "schlechten Ausdruck": `(a+b)*(a++)`

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
Modulus	%	2	14	links
Addition	+	2	13	links
Subtraktion	-	2	13	links

Arithmetische Operatoren

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

-a : R-Wert $\rightarrow$ R-Wert

Arithmetische Operatoren

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

Arithmetische Operatoren

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

$$a+b : \text{R-Wert} \times \text{R-Wert} \rightarrow \text{R-Wert}$$

Arithmetische Operatoren

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

```
-a+b+c
```

Arithmetische Operatoren

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Unäres +	+	1	16	rechts
Negation	**−**	**1**	**16**	**rechts**
Addition	**+**	**2**	**13**	**links**
Subtraktion	−	2	13	links

```
-a+b+c
```

Arithmetische Operatoren

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Unäres +	+	1	16	rechts
Negation	−	**1**	**16**	**rechts**
Addition	+	**2**	**13**	**links**
Subtraktion	−	2	13	links

-a+b+c = ((-a) + b) + c

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

Zuweisungsausdruck – nun genauer

- Bereits bekannt: `a = b` bedeutet
  Zuweisung von `b` (R-Wert) an `a` (L-Wert).
  Rückgabe: L-Wert

Zuweisungsausdruck – nun genauer

- Bereits bekannt: `a = b` bedeutet
  Zuweisung von `b` (R-Wert) an `a` (L-Wert).
  Rückgabe: L-Wert
- Was bedeutet `a = b = c` ?

Zuweisungsausdruck – nun genauer

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

```
a = b = c        ⟺        a = (b = c)
```

Zuweisungsausdruck – nun genauer

`a = b = c` $\iff$ `a = (b = c)`

Beispiel Mehrfachzuweisung:

`a = b = 0` $\Longrightarrow$ `b=0; a=0`

Division und Modulus

- Operator / realisiert ganzzahlige Division

  `5 / 2` hat Wert 2

Division und Modulus

- Operator / realisiert ganzzahlige Division

  `5 / 2` hat Wert 2

- In `fahrenheit.cpp`

```
9 * celsius / 5 + 32

15 degrees Celsius are 59 degrees Fahrenheit
```

Division und Modulus

- In `fahrenheit.cpp`

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

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

Division und Modulus

- In `fahrenheit.cpp`

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

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

- Mathematisch äquivalent...

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

Division und Modulus

- In `fahrenheit.cpp`

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

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

- Mathematisch äquivalent...

```
1 * celsius + 32
```

Division und Modulus

- In `fahrenheit.cpp`

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

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

- Mathematisch äquivalent...

```
15 + 32
```

Division und Modulus

- In `fahrenheit.cpp`

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

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

- Mathematisch äquivalent...

```
47
```

Division und Modulus

- 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
```

Division und Modulus

- Modulus-Operator berechnet Rest der ganzzahligen Division

`5 / 2` hat Wert 2, `5 % 2` hat Wert 1.

Division und Modulus

- Modulus-Operator berechnet Rest der ganzzahligen Division

  `5 / 2` hat Wert 2, `5 % 2` hat Wert 1.

- Es gilt immer:

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

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

Inkrement und Dekrement

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

Nachteile

- relativ lang
- expr wird zweimal ausgewertet (Effekte!)

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

Beispiel

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

In-/Dekrement Operatoren

Beispiel

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

In-/Dekrement Operatoren

Beispiel

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

In-/Dekrement Operatoren

Beispiel

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

C++ vs. ++C

Eigentlich sollte unsere Sprache ++C heissen, denn

- sie ist eine Weiterentwicklung der Sprache C,

C++ vs. ++C

Eigentlich sollte unsere Sprache ++C heissen, denn

- sie ist eine Weiterentwicklung der Sprache C,
- während C++ ja immer noch das alte C liefert.

Arithmetische Zuweisungen

```
a += b
```

$\Leftrightarrow$

```
a = a + b
```

Arithmetische Zuweisungen

```
a += b
```

$\Leftrightarrow$

```
a = a + b
```

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 $b_n \cdot 2^n + \cdots + b_1 \cdot 2 + b_0$

Binäre Zahlendarstellungen

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

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

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

Beispiel: `101011`

Binäre Zahlendarstellungen

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

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

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

Beispiel: `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 $b_n \cdot 2^n + \dots + b_1 \cdot 2 + b_0$

Beispiel: `101011` entspricht `43`.

Binäre Zahlendarstellungen

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

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

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

Beispiel: 101011 entspricht 43.

Binäre Zahlen: Zahlen der Computer?

Wahrheit: Computer rechnen mit Binärzahlen.

NEUE ZÜRCHER ZEITUNG

TECHNIK

Mittwoch, 30. August 1950 Blatt 5
Mittagausgabe Nr. 1736 (50)

Das programmgesteuerte Rechengerät an der Eidgenössischen Technischen Hochschule in Zürich

Die Entwicklung programmgesteuerter Rechenmaschinen in den Vereinigten Staaten von Amerika wurde in den Artikeln „Elektronische Rechenmaschinen" (vgl. Nr. 2140 der „N. Z. Z." vom 13. Oktober 1948) und „Die neueste elektronische Rechenmaschine" (vgl. Nr. 871 der „N. Z. Z." vom 26. April 1950) behandelt. Nachstehend soll von einem Gerät deutscher Herkunft — Zuse K.-G., Neukirchen — die Rede sein, welches im Juli dieses Jahres am Institut für angewandte Mathematik der Eidgenössischen Technischen Hochschule in Zürich, das unter der Leitung von Prof. Dr. E. Stiefel steht, in Betrieb genommen wurde. Damit ist diesem Institut in der Lage, dem in der Schweiz immer stärker werdenden Bedürfnis nach einer leistungsfähigen Zentralstelle für numerische Rechnungen wenigstens teilweise gerecht zu werden. Bereits sind einige mathematische Probleme behandelt worden, und die Erledigung vieler weiterer Aufgaben ist vorbereitet.

Merkmale des Gerätes

Das Gerät ist ein Glied in dem längeren Entwicklungsprogramm des Ingenieurs *Konrad Zuse*; es wurde im Auftrag des *Institutes für angewandte Mathematik* der E. T. H. unter Berücksichtigung von dessen Wünschen und Ideen von Zuse als *„Modell Z 4"* konstruiert. Die ursprüngliche Entwicklung in Deutschland erfolgte in den Kriegsjahren und verlief völlig unabhängig von den Unternehmungen in den Vereinigten Staaten. Es ist übrigens interessant festzustellen, wie für die meisten wichtigen funktionellen Probleme bedeutsam gleiche Lösung gefunden wurde, wie aber anderseits gewissen Fragen sekundärer Wichtigkeit eine ganz unterschiedliche Bedeutung beigemessen wurde.

Eine kurze technische Charakterisierung lautet wie folgt: Elektromechanisch arbeitendes Gerät mit 2200 Relais, 21 Schreibschaltern und einem Speicher für 64 Zahlen, welcher mit neuartigen, mechanischen Schaltgliedern arbeitet; Verwendung des Dualsystems und der halblogarithmischen Darstellung; Multiplikationszeit 2,5 Sekunden; Programmsteuerung mit Hilfe zweier Lochstreifen, auf die wahlweise umgeschaltet werden kann; Eingabe von Zahlen durch eine Tastatur oder durch einen Lochstreifen; Abgabe der Resultate durch Lampenfeld, Lochstreifen oder Druckwerk.

Das duale Zahlsystem

Allgemein wird programmgesteuerten Rechengeräten häufig das duale Zahlsystem zugrunde gelegt, welches nur die zwei Zahlzeichen 0 und 1 verwendet, während das bekannte Dezimalsystem

Lesen wir eine Dezimalzahl von rechts nach links, so erhält sich das Gewicht von Stelle zu Stelle um den Faktor 10. Im *Dualsystem* ist nun einfach dieser Faktor 10 durch 2 zu ersetzen. Also bedeutet die (nunmehr duale) Zahl abcdefg den Ausdruck:

$$a \cdot 2^6 + b \cdot 2^5 + c \cdot 2^4 + d \cdot 2^3 + e \cdot 2^2 + f \cdot 2^1 + g \cdot 2^0$$

Die Zahl 1 wird in beiden Systemen gleich dargestellt. Um jedoch duale von dezimalen Zahlen deutlich zu trennen, schreiben wir die duale 1 als 1_D. — Dagegen weicht schon die 2 ab, indem sie dual 10_D lautet; denn dies bedeutet $1 \cdot 2^1 + 0 \cdot 2^0 = 2$. Wenn einer Zahl (ohne Stellen nach dem Komma) rechts eine Null zugefügt wird, so vergrößert sie sich um den Faktor 2 (und nicht, wie im Dezimalsystem, um den Faktor 10). Auf diese Weise kann aus $1_D 0 = 2$ auf einfachste Weise gebildet werden: $1_D 00 = 4$, $1_D 000 = 8$, $1_D 0000 = 16$, usw.

Die Dualzahl $1_D 01_D 1_D$ bedeutet nun also:

$$1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 21$$

Ganz analog sind etwaige Stellen nach dem Komma zu interpretieren; so wird $1_D 01_D$ wie folgt übersetzt:

$$1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 1 + {}^1/_4 = {}^5/_4 = 1{,}375$$

Der große Vorteil, der das Dualsystem für Rechenmaschinen so geeignet macht, nämlich die Reduktion der Anzahl der verwendeten Symbole auf nur zwei, wird durch den Nachteil eines Nachteil erkauft: Es braucht mehr Stellen, um eine bestimmte Zahl darzustellen. Die einstellige Zahl 9

Änderung des Maßstabes durchgerechnet werden können.

Die beschriebene Darstellung bringt eine gewisse Komplikation der Rechenoperationen mit sich. So müssen vor einer Addition die beiden Summanden zunächst so verschoben werden, daß ihre Kommata untereinander zu liegen kommen, was an Hand eines Beispiels erläutert werden soll. Damit der Leser nicht durch das ungewohnte duale Zahlsystem verwirrt wird, ist das Beispiel im Dezimalsystem durchgeführt; doch wird darum ersteres, daß das Gerät in Wirklichkeit mit dualen Zahlen rechnet.

Es soll also etwa addiert werden: $2{,}345678 \times 10^5 + 9{,}876543 \times 10^1$ (Man beachte, daß die eigentliche Zahl stets zwischen 1 und 10 liegt, also das Komma nach der ersten Stelle hat.) Nun müssen die beiden Summanden „ausgerichtet" werden, d. h. die beiden Exponenten sind einander gleich zu machen, und zwar erhält der kleinere Exponent den Wert des größeren, also 2. Die Zahlen lauten nun, richtig untereinander geschrieben und addiert, wie folgt:

$$\begin{array}{l} 2{,}345678 \times 10^5 \\ 0{,}009876 \times 10^5 \\ \hline 2{,}355554 \times 10^5 \end{array}$$

Es ist ersichtlich, daß bei der kleineren der beiden Zahlen rechts einige Stellen abgeschnitten werden müssen; dann wenn die Summanden siebenstellig angegeben waren, so soll auch das Resultat nicht mehr als sieben Stellen enthalten.

Abb. 2. Der Schaltpult bei der Fertigung eines Rechenplanes. Die Abtaster für den Lochstreifen sind deutlich sichtbar.

Befehle können „bedingt" gegeben werden, d. h. ihre Ausführung wird von der Natur eines errechneten Resultates abhängig gemacht. Erst dadurch werden die mathematischen Mittel Potenzial

Binäre Zahlen: Zahlen der Computer?

Klischee: Computer reden 0/1-Kauderwelsch.

Überall Windmühlen – Gustl Mollath und die Gerechtigkeit › Die Seite Drei

Süddeutsche Zeitung

NEUESTE NACHRICHTEN AUS POLITIK, KULTUR, WIRTSCHAFT UND SPORT

WWW.SÜDDEUTSCHE.DE | HH2 | MÜNCHEN, SAMSTAG/SONNTAG, 26./27. JULI 2014 | 70. JAHRGANG / 30. WOCHE / NR. 170 / 2,50 EURO

Das Streiflicht

((SZ) Carl Barks, der Erfinder der nach wie vor unumstritten reichsten Ente der Welt, war für viele ein großer Künstler. Für viele andere war der Comicautor und Maler nichts weiter als ein harmloser Entenzeichner. Die im Land der Dichter und Denker schmerzhaft genaue Unterscheidung zwischen Hoch- und Niedrigkultur wird auch bei diesem großen Humoristen unbarmherzig durchgezogen. Amerika macht sich in Sachen Popkultur lockerer. Dort gelten Barks' Geschichten als Meisterstücke der gesellschaftskritischen Satire: Wenn Dagobert Duck Gier und Geiz zu immer neuen Höhen trieb, stand plötzlich auch der amerikanische Turbokapitalismus mit einem aufgeregt wackelnden Entensterz da. Hierzulande sieht man in Barks immer noch eher den Lieferanten kindgerechter Zerstreuung. Allein musste er sich dabei nicht fühlen, gegen dieses Urteil kämpft jeder Comic-Künstler fortwährend an.

Nun haben Barks, alle Comiczeichner und überhaupt alle Menschen, die in einer Gastwirtschaft der Nachwelt auf Bierdeckeln lustig anzuschauende Figürchen hinterlassen, himmlischen Beistand erhalten. Auch das Universum zeichnet Enten. Und nicht die schlechtesten, um das gleich hinzuzufügen. Die Menschheit muss nur mal genau hinschauen beziehungsweise eine russische Raumsonde losschicken, und schon können wir erkennen, dass der Komet 67P/Tschurjumov-Gerasimenko liebevoll kurz „Tschuri" genannt, in Form einer Ente durchs Weltall saust. Das Universum kann auch anders, mit diesem Stoßseufzer dürfen wir fortan zum Himmel blicken. Kometen, das sind nicht zwangsläufig Angst und Schrecken verbreitende Himmelszeichen, die mit loderndem Schweif Pest,

WOCHENENDE

Liebe Nur Patriarchen und Kapuzenpullischluffis unterwegs? Wir Männer sind weiter, als wir glauben. JAKOB SCHRENK

Luxus Während der Finanzkrise verschwand der Kaviar von der Speisekarte. Zu Besuch bei Züchtern, Händlern und Kunden. TIM NESHITOV

Punk Berlins Kulturstaatssekretär Tim Renner im Interview. MARTIN ZIPS

Mit Stellenmarkt

Anklage wegen Verwandtenaffäre

Ex-CSU-Fraktionschef Schmid soll sich vor Gericht verantworten

Augsburg – Die Staatsanwaltschaft Augsburg hat Anklage gegen den ehemaligen CSU-Fraktionschef Georg Schmid erhoben. Der 61-Jährige soll mindestens 340 000 Euro Sozialversicherungsbeiträge hinterzogen haben. Nach Angaben

Binäre Zahlen: Zahlen der Computer?

Klischee: Computer reden 0/1-Kauderwelsch.

01001110 01011010 01011010

01001010 01010110 01001101

Freitag, 8. Juni 2012 · Nr. 131 · 233. Jhg. www.nzz.ch · Fr. 4.00 · €3.50

01000010 01100101
01110010 01101001

01100011 01101000 01110100 01100101

01000110 01101100 11111100

01101000

Rechentricks

- Abschätzung der Grössenordnung von Zweierpotenzen[3]:

$2^{10} = 1024 = 1\text{Ki} \approx 10^3$.
$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi}$.
$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}$.

[3] Dezimal vs. Binäre Einheiten: MB - Megabyte vs. MiB - Megabibyte (etc.)
kilo (K, Ki) – mega (M, Mi) – giga (G, Gi) – tera(T, Ti) – peta(P, Pi) – exa (E, Ei)

Rechentricks

- Abschätzung der Grössenordnung von Zweierpotenzen[3]:

$2^{10} = 1024 = 1\text{Ki} \approx 10^3.$
$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi}.$
$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}.$

[3] Dezimal vs. Binäre Einheiten: MB - Megabyte vs. MiB - Megabibyte (etc.)
kilo (K, Ki) – mega (M, Mi) – giga (G, Gi) – tera(T, Ti) – peta(P, Pi) – exa (E, Ei)

Rechentricks

- Abschätzung der Grössenordnung von Zweierpotenzen[3]:

$2^{10} = 1024 = 1\text{Ki} \approx 10^3.$
$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi}.$
$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}.$

[3] Dezimal vs. Binäre Einheiten: MB - Megabyte vs. MiB - Megabibyte (etc.)
kilo (K, Ki) – mega (M, Mi) – giga (G, Gi) – tera(T, Ti) – peta(P, Pi) – exa (E, Ei)

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`

Beispiel: `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”.

Wozu Hexadezimalzahlen?

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

32-bit Zahlen: `0x00000000` -- `0xffffffff` .
`0x80000000`: höchstes Bit einer 32-bit Zahl gesetzt.
`0xffffffff`: alle Bits einer 32-bit Zahl gesetzt.
„`0x8a20aaf0` ist eine Addresse in den oberen 2G des 32-bit Addressraumes”

Wozu Hexadezimalzahlen?

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

32-bit Zahlen: `0x00000000` -- `0xffffffff` .
`0x80000000`: höchstes Bit einer 32-bit Zahl gesetzt.
`0xffffffff`: alle Bits einer 32-bit Zahl gesetzt.
„`0x8a20aaf0` ist eine Addresse in den oberen 2G des 32-bit Addressraumes”

Wozu Hexadezimalzahlen?

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

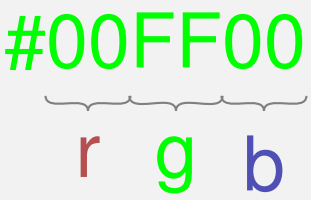
32-bit Zahlen: `0x00000000` -- `0xffffffff` .
`0x80000000`: höchstes Bit einer 32-bit Zahl gesetzt.
`0xffffffff`: alle Bits einer 32-bit Zahl gesetzt.
„`0x8a20aaf0` ist eine Addresse in den oberen 2G des 32-bit Addressraumes”

Wozu Hexadezimalzahlen?

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

32-bit Zahlen: `0x00000000` -- `0xffffffff` .
`0x80000000`: höchstes Bit einer 32-bit Zahl gesetzt.
`0xffffffff`: alle Bits einer 32-bit Zahl gesetzt.
„`0x8a20aaf0` ist eine Addresse in den oberen 2G des 32-bit Addressraumes”

Beispiel: Hex-Farben



Beispiel: Hex-Farben

Beispiel: Hex-Farben

Beispiel: Hex-Farben

Wozu Hexadezimalzahlen?

“Für Programmierer und Techniker”

(Bedienungsanleitung Schachcomputer *Mephisto II*, 1981)

Beispiele:

8200

a) Anzeige ***8200***
MEPHISTO ist mit genau 2 Bauern-Einheiten im Vorteil.

7F00

b) Anzeige ***7F00***
MEPHISTO ist mit genau 1 Bauern-Einheit im Nachteil.

Die Anzeige erfolgt in ***hexadezimaler Schreibweise.*** Im Gegensatz zum gewohnten Dezimalsystem gehen die Ziffern an jeder Stelle von 0 bis F (A = 10, B = 11, ... F = 15).
Für mathematisch Vorgebildete nachstehend die Umrechnungsformel in das dezimale Punktsystem:

$$ABCD = (Ax16^3) + (Bx16^2) + (Cx16^1) + (Dx16^0)$$

Für A gilt: 7 = -1; 8 = 0; 9 = +1 usw.
Eine Bauerneinheit (B) wird ausgedrückt in $16^2 = 256$ Punkten.
Dieses auf den ersten Blick vielleicht etwas komplizierte System dient der Service-Freundlichkeit von MEPHISTO, sowie insbesondere der Entwicklungsarbeit an zukünftigen, noch stärkeren Programmen, ist also mehr für unsere Programmierer und Techniker vorgesehen.

Beispiele:

805E

c) Anzeige ***805E***
(E=14) Umrechnung nach folgendem Verfahren:
$(14x16^0) + (5x16^1) + (0x16^2) + (0x16^3) = 14+80+0+0 =$
$= +94$ ***Punkte.***

7F80

d) Anzeige ***7F80***
(7=-1; F=15) Umrechnung wie folgt:
$(0x16^0) + (8x16^1) + (15x16^2) - (1x16^3) = 0+128+3840-4096 =$

Wozu Hexadezimalzahlen?

Die NZZ hätte viel Platz sparen können...

Wertebereich des Typs `int`

```
// Program: limits.cpp
// 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`

```
// Program: limits.cpp
// 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;
}
```

Zum Beispiel

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

Wertebereich des Typs `int`

```
// Program: limits.cpp
// 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;
}
```

Zum Beispiel

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

Woher kommen diese Zahlen?

Wertebereich des Typs `int`

- Repräsentation mit B Bits. Wertebereich

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

Wertebereich des Typs `int`

- Repräsentation mit B Bits. Wertebereich

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

- Auf den meisten Plattformen $B = 32$

Woher kommt gerade diese Aufteilung?

Wertebereich des Typs `int`

- Repräsentation mit B Bits. Wertebereich

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

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

Woher kommt gerade diese Aufteilung?

Überlauf und Unterlauf

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

`power8.cpp`: $15^8 = -1732076671$

`power20.cpp`: $3^{20} = -808182895$

- Es gibt *keine Fehlermeldung!*

Der Typ `unsigned int`

- Wertebereich
$$\{0, 1, \ldots, 2^B - 1\}$$
- 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
```

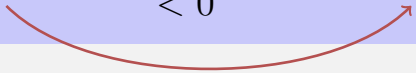
- Solche gemischten Ausdrücke sind vom „allgemeineren” Typ `unsigned int`.
- `int`-Operanden werden *konvertiert* nach `unsigned int`.

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	$x + 2^B$



Bei Zweierkomplementdarstellung passiert dabei intern gar nichts

Rechnen mit Binärzahlen (4 Stellen)

Einfache Addition

$$\begin{array}{r} 2 \\ +3 \\ \hline 5 \end{array} \qquad \begin{array}{r} 0010 \\ +0011 \\ \hline 0101 \end{array}$$

Rechnen mit Binärzahlen (4 Stellen)

Einfache Subtraktion

$$\begin{array}{r} 5 \\ -3 \\ \hline 2 \end{array} \qquad \begin{array}{r} 0101 \\ -0011 \\ \hline 0010 \end{array}$$

Rechnen mit Binärzahlen (4 Stellen)

Addition mit Überlauf

$$\begin{array}{r} 7 \\ +9 \\ \hline 16 \end{array} \qquad \begin{array}{r} 0111 \\ +1001 \\ \hline (1)0000 \end{array}$$

Rechnen mit Binärzahlen (4 Stellen)

Negative Zahlen?

$$\begin{array}{r} 5 \\ +(-5) \\ \hline 0 \end{array} \qquad \begin{array}{r} 0101 \\ ???? \\ \hline (1)0000 \end{array}$$

Rechnen mit Binärzahlen (4 Stellen)

Einfacher: -1

$$\begin{array}{r} 1 \\ +(-1) \\ \hline 0 \end{array} \qquad \begin{array}{r} 0001 \\ 1111 \\ \hline (1)0000 \end{array}$$

Rechnen mit Binärzahlen (4 Stellen)

Nutzen das aus:

$$\begin{array}{r} 3 \\ +? \\ \hline -1 \end{array} \qquad \begin{array}{r} 0011 \\ +???? \\ \hline 1111 \end{array}$$

Rechnen mit Binärzahlen (4 Stellen)

Invertieren!

$$\begin{array}{r} 3 \\ +(-4) \\ \hline -1 \end{array} \qquad \begin{array}{r} 0011 \\ +1100 \\ \hline 1111 \end{array} \widehat{=} 2^B - 1$$

Rechnen mit Binärzahlen (4 Stellen)

$$\begin{array}{r} a \\ +(-a-1) \\ \hline -1 \end{array} \qquad \begin{array}{r} a \\ \bar{a} \\ \hline 1111 \end{array} \widehat{=} 2^B - 1$$

Rechnen mit Binärzahlen (4 Stellen)

- Negation: Inversion und Addition von 1

$$-a \quad \widehat{=} \quad \bar{a} + 1$$

Rechnen mit Binärzahlen (4 Stellen)

- Wrap-around Semantik (Rechnen modulo 2^B)

$$-a \quad \widehat{=} \quad 2^B - a$$

Warum das funktioniert

Modulo-Arithmetik: Rechnen im Kreis[4]

[4] Die 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.

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

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

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

$\mathrm{AND}(x, y)$ $x \wedge y$

- “Logisches Und”

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

- 0 entspricht „falsch”.
- entspricht „wahr”.

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

Logischer Operator &&

`a && b` (logisches Und)

$$\texttt{bool} \times \texttt{bool} \to \texttt{bool}$$

$$\text{R-Wert} \times \text{R-Wert} \to \text{R-Wert}$$

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)
```

$\mathrm{OR}(x, y)$ $x \vee y$

- “Logisches Oder”

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

- 0 entspricht „falsch”.
- entspricht „wahr”.

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

Logischer Operator ||

`a || b` (logisches Oder)

$$\texttt{bool} \times \texttt{bool} \to \texttt{bool}$$

$$\text{R-Wert} \times \text{R-Wert} \to \text{R-Wert}$$

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)
```

$\mathrm{NOT}(x)$ $\neg x$

- “Logisches Nicht”

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

x	$\mathrm{NOT}(x)$
0	1
1	0

- 0 entspricht „falsch”.
- entspricht „wahr”.

Logischer Operator !

`!b` (logisches Nicht)

`bool` $\rightarrow$ `bool`

R-Wert $\rightarrow$ 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)
```

Präzedenzen

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)
```

Präzedenzen

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 $\mathrm{C{+}{+}}$ verfügbaren Booleschen Funktionen.

Vollständigkeit: $\mathrm{XOR}(x, y)$

$x \oplus y$

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

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

Vollständigkeit: $\mathrm{XOR}(x, y)$ $x \oplus y$

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

Vollständigkeit: $\mathrm{XOR}(x, y)$ $x \oplus y$

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

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

Vollständigkeit: $\mathrm{XOR}(x, y)$

$x \oplus y$

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

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

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

Vollständigkeit Beweis

- Identifiziere binäre Boolesche Funktionen mit ihrem charakteristischen Vektor.

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

Charakteristischer Vektor: 0110

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

Vollständigkeit Beweis

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

$$
\begin{aligned}
f_{0001} &= \mathrm{AND}(x, y) \\
f_{0010} &= \mathrm{AND}(x, \mathrm{NOT}(y)) \\
f_{0100} &= \mathrm{AND}(y, \mathrm{NOT}(x)) \\
f_{1000} &= \mathrm{NOT}(\mathrm{OR}(x, y))
\end{aligned}
$$

Vollständigkeit Beweis

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

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

Vollständigkeit Beweis

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

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

- 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`	$\rightarrow$	`int`
true	$\rightarrow$	1
false	$\rightarrow$	0

`bool` vs `int`: Konversion

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

`bool`	→	`int`
true	→	1
false	→	0

`int`	→	`bool`
≠0	→	*true*
0	→	*false*

`bool` vs `int`: Konversion

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

`bool`	→	`int`
true	→	1
false	→	0

`int`	→	`bool`
≠0	→	*true*
0	→	*false*

```
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`	$\rightarrow$	`int`
true	$\rightarrow$	1
false	$\rightarrow$	0

`int`	$\rightarrow$	`bool`
$\neq 0$	$\rightarrow$	*true*
0	$\rightarrow$	*false*

```
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

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)

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

DeMorgansche Regeln

Hinterfrage das Offensichtliche!

```
#include<cassert>

int main()
{
  bool x = ...; // whatever x and y actually are,
  bool y = ...; // De Morgan's laws will hold:
  assert ( !(x && y) == (!x || !y) );
  assert ( !(x || y) == (!x && !y) );
  return 0;
}
```

DeMorgansche Regeln

Hinterfrage das Offensichtliche!

```
#include<cassert>

int main()
{
  bool x = ...; // whatever x and y actually are,
  bool y = ...; // De Morgan's laws will hold:
  assert ( !(x && y) == (!x || !y) );
  assert ( !(x || y) == (!x && !y) );
  return 0;
}
```

DeMorgansche Regeln

Hinterfrage das scheinbar Offensichtliche!

```
#include<cassert>

int main()
{
  bool x = ...; // whatever x and y actually are,
  bool y = ...; // De Morgan's laws will hold:
  assert ( !(x && y) == (!x || !y) );
  assert ( !(x && y) == (!x && !y) ); //assertion failure
  return 0;
}
```

Assertions abschalten

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

int main()
{
  bool x; // whatever x and y actually are,
  bool y; // De Morgan's laws will hold:
  assert ( !(x && y) == (!x || !y) ); // ignored
  assert ( !(x || y) == (!x && !y) ); // ignored
  return 0;
}
```

Div-Mod Identität

`a/b * b + a%b == a`

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

```
std::cout << "Dividend a =? ";
int a;
std::cin >> a;

std::cout << "Divisor b =? ";
int b;
std::cin >> b;

// check input
assert (b != 0);
```

Eingabe der Argumente für die Berechnung

Div-Mod Identität

`a/b * b + a%b == a`

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

```
std::cout << "Dividend a =? ";
int a;
std::cin >> a;

std::cout << "Divisor b =? ";
int b;
std::cin >> b;

// check input
assert (b != 0);
```

Vorbedingung für die weitere Berechnung

Div-Mod Identität

`a/b * b + a%b == a`

. . . und hinterfrage das Offensichtliche!

```
// check input
assert (b != 0);
```

← Vorbedingung für die weitere Berechnung

```
// compute result
int div = a / b;
int mod = a % b;

// check result
assert (div * b + mod == a);
...
```

Div-Mod Identität

`a/b * b + a%b == a`

. . . und hinterfrage das Offensichtliche!

```
// check input
assert (b != 0);

// compute result
int div = a / b;
int mod = a % b;

// check result
assert (div * b + mod == a);
...
```

Div-Mod Identität

4. Kontrollanweisungen I

Auswahlanweisungen, Iterationsanweisungen, Terminierung, Blöcke

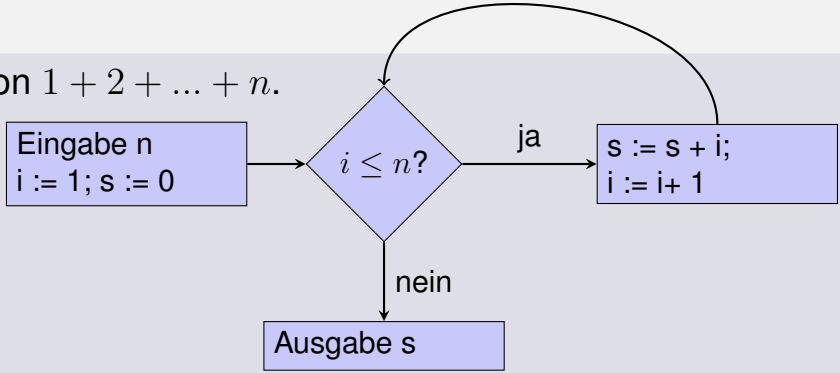
Kontrollfluss

- bisher *linear* (von oben nach unten)
- Für interessante Programme braucht man „Verzweigungen” und „Sprünge”.

Kontrollfluss

- bisher *linear* (von oben nach unten)
- Für interessante Programme braucht man „Verzweigungen” und „Sprünge”.

Berechung von $1 + 2 + ... + n$.



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";
```

- *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";
```

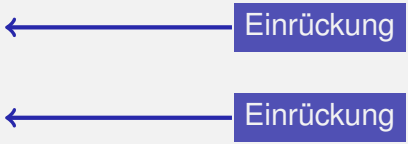
- *condition*: konvertierbar nach `bool`.
- *statement1*: *Rumpf* des `if`-Zweiges
- *statement2*: *Rumpf* des `else`-Zweiges

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";
else
    std::cout << "odd";
```



Iterationsanweisungen

realisieren „Schleifen”:

- `for`-Anweisung
- `while`-Anweisung
- `do`-Anweisung

Berechne $1 + 2 + \ldots + n$

```
// 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 $1 + 2 + \ldots + n$

```
// 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*)
    statement

`for`-Anweisung: Syntax

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

Deklarationsanweisung

auch möglich: Ausdrucksanweisung, Nullanweisung

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

`for`-Anweisung: Syntax

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

Ausdruck vom Typ `bool`

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

`for`-Anweisung: Syntax

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

Ausdruck vom Typ `unsigned int`

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

`for`-Anweisung: Syntax

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

Ausdrucksanweisung

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

Der kleine Gauß (1777 - 1855)

- Mathe-Lehrer wollte seine Schüler mit folgender Aufgabe beschäftigen:

Der kleine Gauß (1777 - 1855)

- Mathe-Lehrer wollte seine Schüler mit folgender Aufgabe beschäftigen:

Berechne die Summe der Zahlen von 1 bis 100 !

Der kleine Gauß (1777 - 1855)

- Mathe-Lehrer wollte seine Schüler mit folgender Aufgabe beschäftigen:

Berechne die Summe der Zahlen von 1 bis 100 !

- Gauß war nach einer Minute fertig.

Die Lösung von Gauß

- Die gesuchte Zahl ist

$$1 + 2 + 3 + \cdots + 98 + 99 + 100.$$

Die Lösung von Gauß

- Die gesuchte Zahl ist

$$1 + 2 + 3 + \cdots + 98 + 99 + 100.$$

- Das ist die Hälfte von

$$\begin{array}{rcrcrcrcr}
 & 1 & + & 2 & + & \cdots & + & 99 & + & 100 \\
+ & 100 & + & 99 & + & \cdots & + & 2 & + & 1 \\
\hline
= & 101 & + & 101 & + & \cdots & + & 101 & + & 101
\end{array}$$

Die Lösung von Gauß

- Die gesuchte Zahl ist

$$1 + 2 + 3 + \cdots + 98 + 99 + 100.$$

- Das ist die Hälfte von

$$\begin{array}{rcrcrcrcr} & 1 & + & 2 & + & \cdots & + & 99 & + & 100 \\ + & 100 & + & 99 & + & \cdots & + & 2 & + & 1 \\ \hline = & 101 & + & 101 & + & \cdots & + & 101 & + & 101 \end{array}$$

- Antwort: $100 \cdot 101/2 = 5050$

`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 ( e; v; e) r;
```

Halteproblem

Unentscheidbarkeit des Halteproblems

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

[5] Alan 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.[5]

[5] Alan 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 $d \in \{2, \ldots, n-1\}$ ein Teiler von n ist.

Beispiel: Primzahltest

Def.: Eine natürliche Zahl $n \geq 2$ ist eine Primzahl, wenn kein $d \in \{2, \ldots, n-1\}$ 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 $d \in \{2, \ldots, n-1\}$ 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)

Beispiel: Primzahltest

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

Eine Schleife, die das testet:

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

- Beobachtung 1:
  Nach der `for`-Anweisung gilt $d \leq n$.

Beispiel: Primzahltest

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

Eine Schleife, die das testet:

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

- Beobachtung 2:
  n ist Primzahl genau wenn am Ende $d = n$.

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";
}
```

5. 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:  undeklarierter Name
    return 0;
}
```

Sichtbarkeit

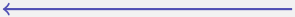
Deklaration in einem Block ist ausserhalb des Blocks nicht „sichtbar”.

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

main block

block

„Blickrichtung”



Kontrollanweisung definiert Block

Kontrollanweisungen verhalten sich in diesem Zusammenhang wie Blöcke.

```
int main()
{
    for (unsigned int i = 0; i < 10; ++i)
        s += i;
    std::cout << i; // Fehler:  undeklarierter Name
    return 0;
}
```

Kontrollanweisung definiert Block

Kontrollanweisungen verhalten sich in diesem Zusammenhang wie Blöcke.

```
int main()
{
  for (unsigned int i = 0; i < 10; ++i)
      s += i;
  std::cout << i; // Fehler:  undeklarierter Name
  return 0;
}
```

block

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

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

scope

Im Funktionsrumpf

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

scope

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
```

`while`-Anweisung: Warum?

- Bei `for`-Anweisung ist oft expression allein für den Fortschritt zuständig („Zählschleife”)

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

`while`-Anweisung: Warum?

- Bei `for`-Anweisung ist oft expression allein für den Fortschritt zuständig („Zählschleife”)

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

- Falls der Fortschritt nicht so einfach ist, kann `while` besser lesbar sein.

Beispiel: Die Collatz-Folge $(n \in \mathbb{N})$

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

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)

Die Collatz-Folge in C++

```
// Input
std::cout << "Compute Collatz sequence, n =? ";
unsigned int n;
std::cin >> n;

// Iteration
while (n > 1)       // stop when 1 reached
{
    if (n % 2 == 0) // n is even
        n = n / 2;
    else            // n is odd
        n = 3 * n + 1;
    std::cout << n << " ";
}
```

Die Collatz-Folge in C++

```
// Input
std::cout << "Compute Collatz sequence, n =? ";
unsigned int n;
std::cin >> n;

// Iteration
while (n > 1)       // stop when 1 reached
{
    if (n % 2 == 0) // n is even
        n = n / 2;
    else            // n is odd
        n = 3 * n + 1;
    std::cout << n << " ";
}
```

Die Collatz-Folge in C++

```
n = 27:
82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242,
121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233,
700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336,
668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276,
638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429,
7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232,
4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488,
244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20,
10, 5, 16, 8, 4, 2, 1
```

`do` Anweisung

```
do
  statement
while ( expression );
```

`do` Anweisung

```
do
  statement
while ( expression );
```

ist äquivalent zu

```
statement
while ( expression )
  statement
```

Beispiel Taschenrechner

Summiere ganze Zahlen (bei 0 ist Schluss):

```
int a;    // next input value
int s = 0; // sum of values so far
do {
    std::cout << "next number =? ";
    std::cin >> a;
    s += a;
    std::cout << "sum = " << s << "\n";
} while (a != 0);
```

Sprunganweisungen

- `break;`
- `continue;`

Taschenrechner mit `break`

Summiere ganze Zahlen (bei 0 ist Schluss):

```
int a;
int s = 0;
do {
    std::cout << "next number =? ";
    std::cin >> a;
    // irrelevant in letzter Iteration:
    s += a;
    std::cout << "sum = " << s << "\n";
}  while (a != 0);
```

Taschenrechner mit `break`

Unterdrücke irrelevante Addition von 0:

```
int a;
int s = 0;
do {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a == 0) break; // Abbruch in der Mitte
    s += a;
    std::cout << "sum = " << s << "\n";
} while (a != 0)
```

Taschenrechner mit `break`

Äquivalent und noch etwas einfacher:

```
int a;
int s = 0;
for (;;) {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a == 0) break; // Abbruch in der Mitte
    s += a;
    std::cout << "sum = " << s << "\n";
}
```

Taschenrechner mit `continue`

Ignoriere alle negativen Eingaben:

```
for (;;)
{
    std::cout << "next number =? ";
    std::cin >> a;
    if (a < 0) continue; // springe zu }
    if (a == 0) break;
    s += a;
    std::cout << "sum = " << s << "\n";
}
```

Kontrollfluss

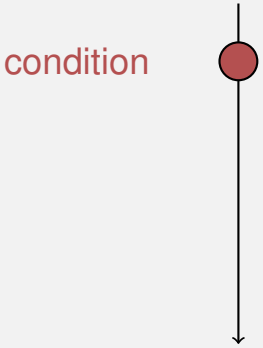
Reihenfolge der (wiederholten) Ausführung von Anweisungen

- Grundsätzlich von oben nach unten...

Kontrollfluss

Reihenfolge der (wiederholten) Ausführung von Anweisungen

- Grundsätzlich von oben nach unten...
- ...ausser in Auswahl- und Kontrollanweisungen

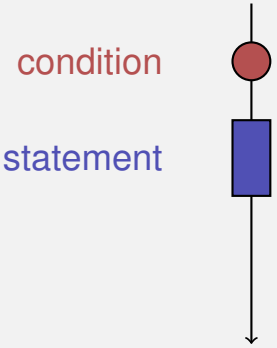


`if` (*condition*)
statement

Kontrollfluss

Reihenfolge der (wiederholten) Ausführung von Anweisungen

- Grundsätzlich von oben nach unten...
- ...ausser in Auswahl- und Kontrollanweisungen

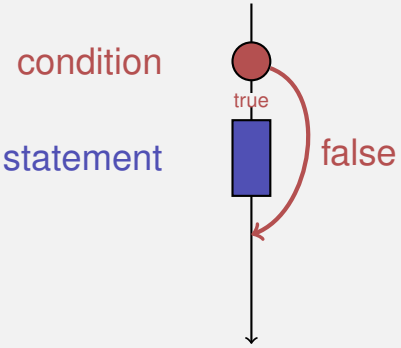


`if` (*condition*)
statement

Kontrollfluss

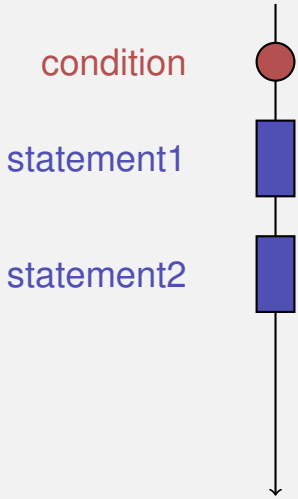
Reihenfolge der (wiederholten) Ausführung von Anweisungen

- Grundsätzlich von oben nach unten...
- ...ausser in Auswahl- und Kontrollanweisungen



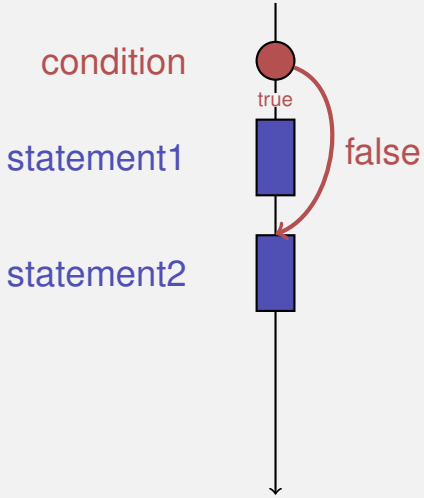
`if` (*condition*)
statement

Kontrollfluss `if else`



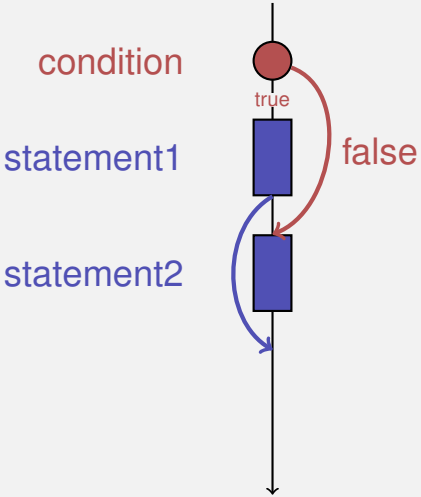

```
if ( condition )
    statement1
else
    statement2
```

Kontrollfluss `if else`




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

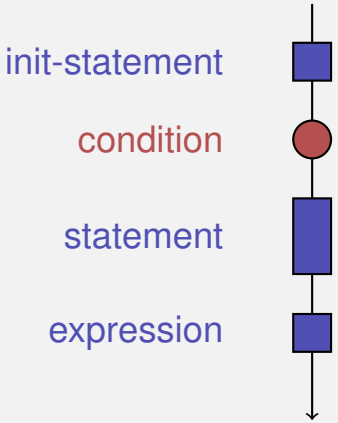
Kontrollfluss `if else`




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

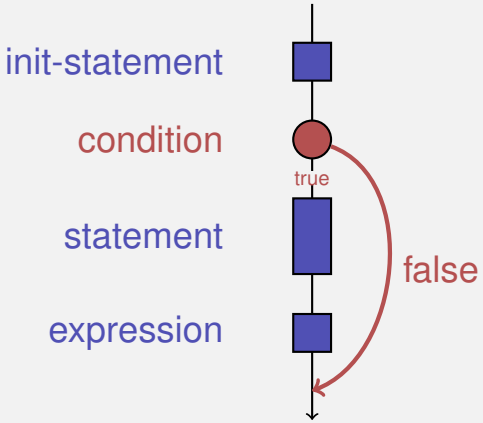
Kontrollfluss `for`

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



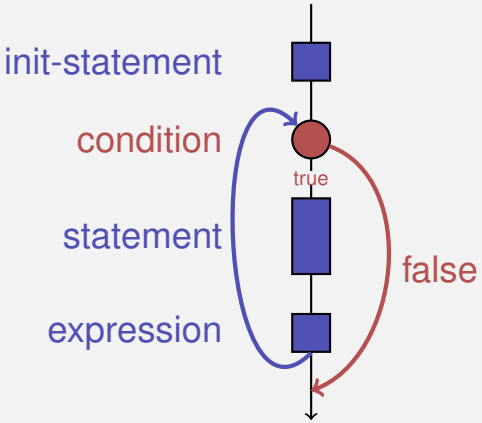
Kontrollfluss `for`

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

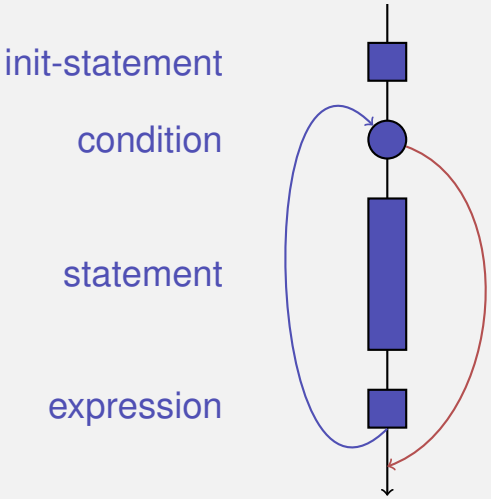


Kontrollfluss `for`

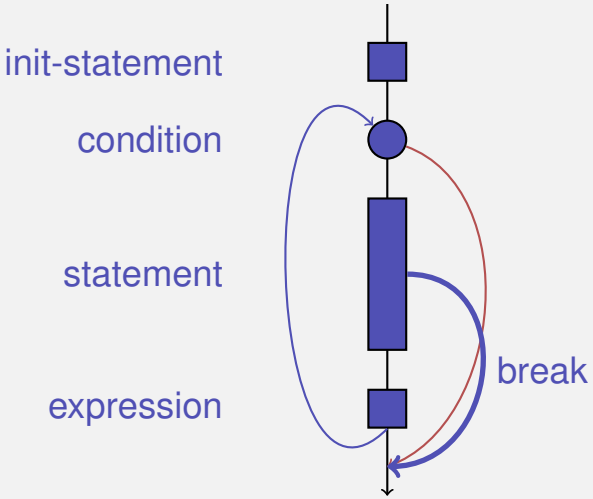
`for` (*init statement* *condition* `;` *expression*)
statement



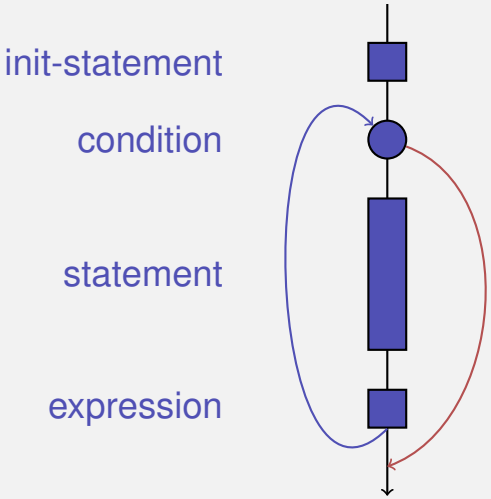
Kontrollfluss `break` und `continue` in for



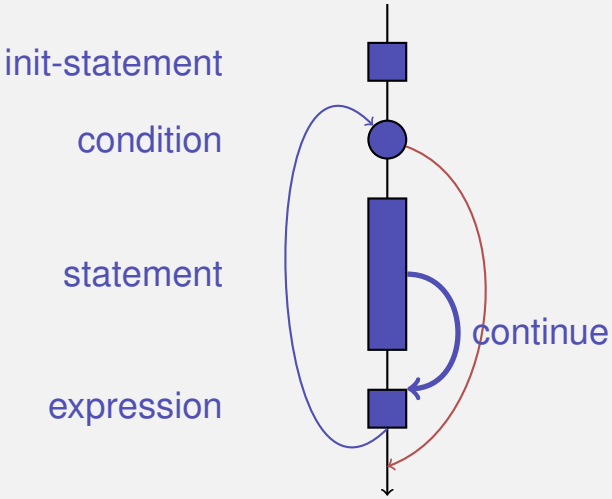
Kontrollfluss `break` und `continue` in for



Kontrollfluss `break` und `continue` in for



Kontrollfluss `break` und `continue` in for



Kontrollfluss: Die guten alten Zeiten?

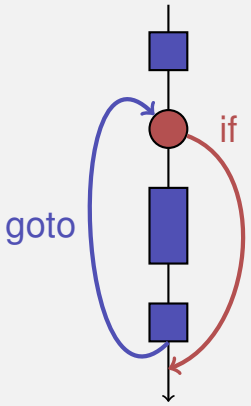
Beobachtung

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`).



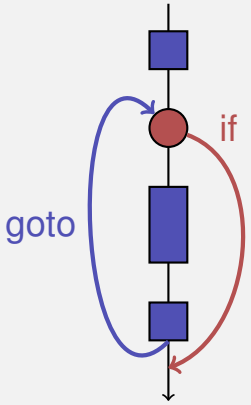
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

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

Modelle:

- Maschinensprache



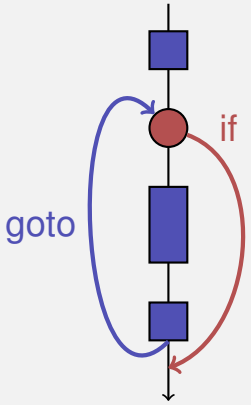
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

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

Modelle:

- Maschinensprache
- Assembler (“höhere” Maschinensprache)



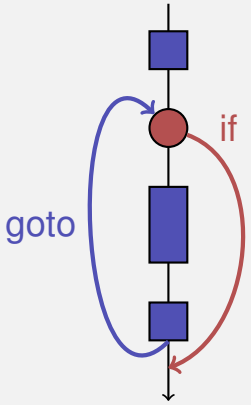
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

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

Modelle:

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



BASIC und die Home-Computer...

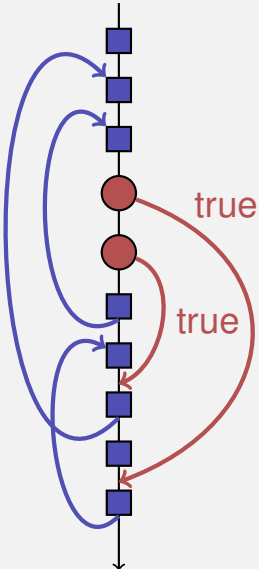
...ermöglichten einer ganzen Generation von Jugendlichen das Programmieren.

Home-Computer Commodore C64 (1982)

Spaghetti-Code mit `goto`

Ausgabe 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 $\{0, \ldots, 100\}$

Erster (korrekter) Versuch:

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

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

Weniger Anweisungen, *weniger* Zeilen:

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

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

Weniger Anweisungen, *einfacherer* Kontrollfluss:

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

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

Weniger Anweisungen, *einfacherer* Kontrollfluss:

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

Das ist hier die “richtige” Iterationsanweisung!

Die `switch`-Anweisung

`switch` (*condition*)
statement

Die `switch`-Anweisung

`switch` (*condition*)
statement

```
int Note;
...
switch (Note) {
    case 6:
        std::cout << "super!";
        break;
    case 5:
        std::cout << "cool!";
        break;
    case 4:
        std::cout << "ok.";
        break;
    default:
        std::cout << "hmm...";
}
```

Die `switch`-Anweisung

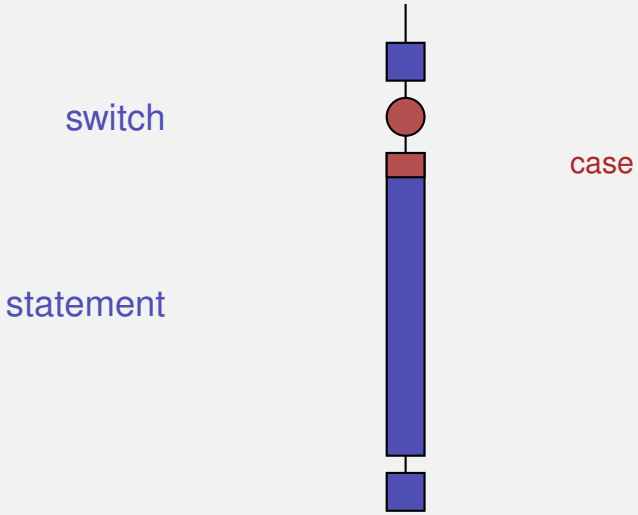
`switch` (*condition*)
statement

Die `switch`-Anweisung

`switch` (*condition*)
statement

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

Kontrollfluss `switch`



Kontrollfluss `switch`

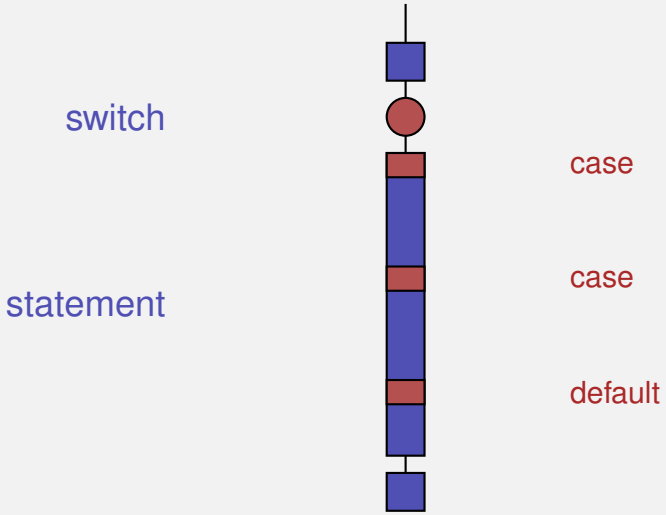
switch

statement

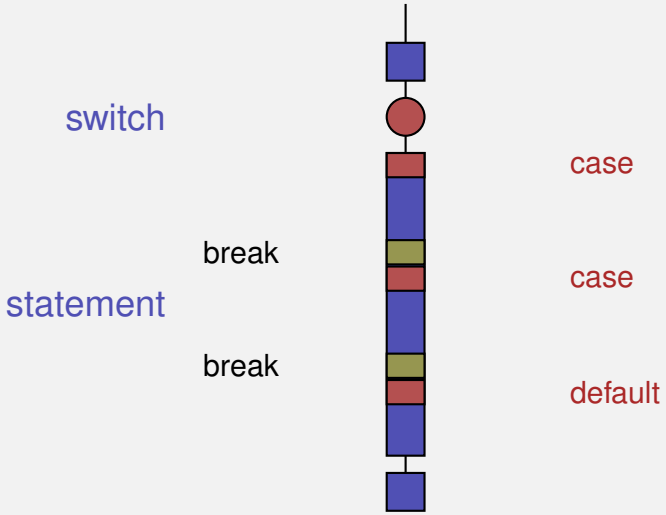
case

case

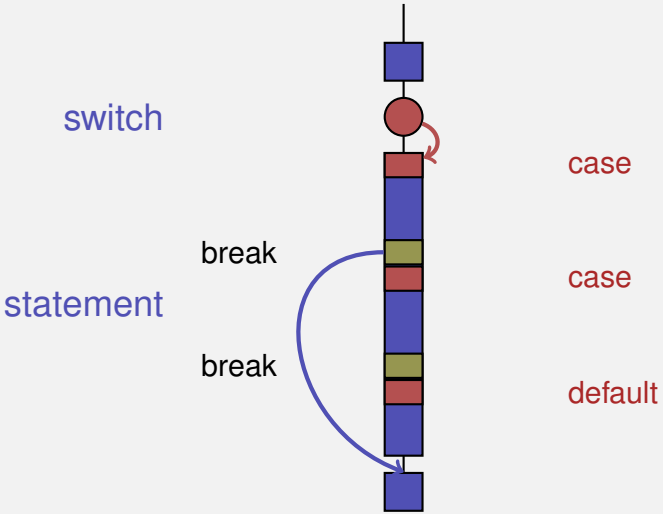
Kontrollfluss `switch`



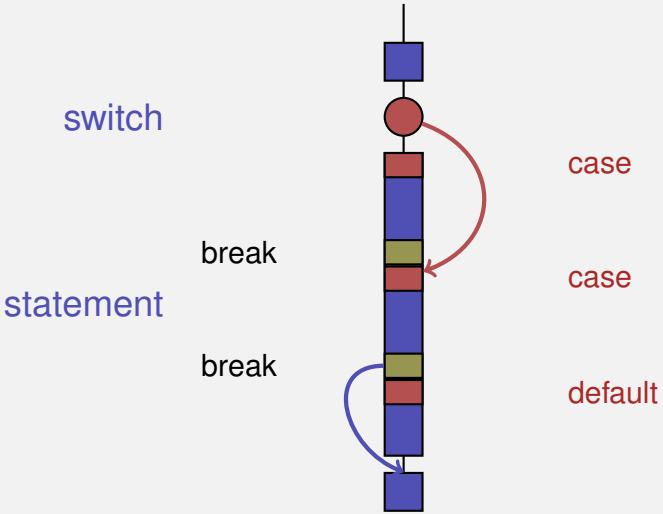
Kontrollfluss `switch`



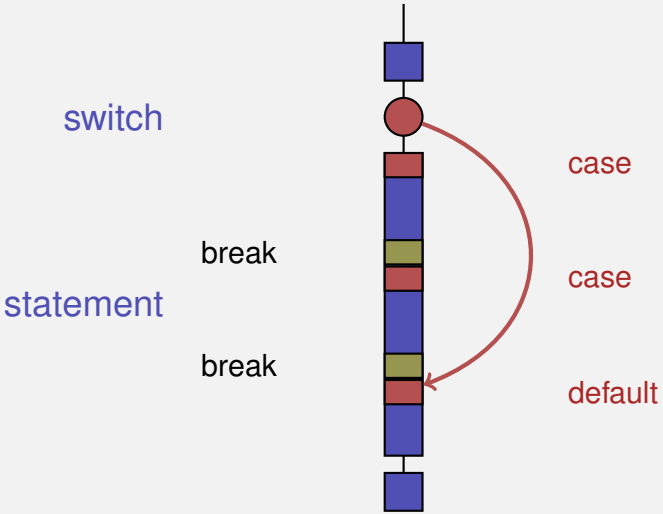
Kontrollfluss `switch`



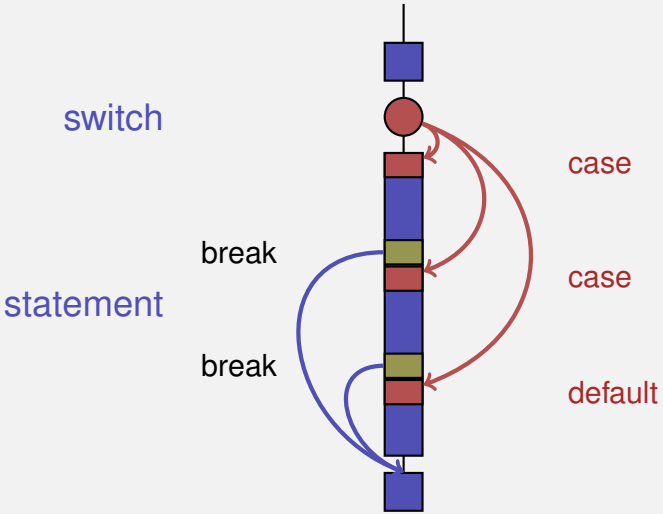
Kontrollfluss `switch`



Kontrollfluss `switch`



Kontrollfluss `switch`



6. Fliesskommazahlen I

Typen `float` und `double`; Gemischte Ausdrücke und Konversionen; Löcher 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; // Fliesskommazahlentyp
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

- feste Anzahl signifikante Stellen (z.B. 10)
- plus Position des Kommas

$$\texttt{82.4} = 824 \cdot 10^{-1}$$

$$\texttt{0.0824} = 824 \cdot 10^{-4}$$

- Zahl ist $\mathit{Signifikand} \times 10^{\mathit{Exponent}}$

Fliesskommazahlen

- feste Anzahl signifikante Stellen (z.B. 10)
- plus Position des Kommas

$$\texttt{82.4} = 824 \cdot 10^{-1}$$

$$\texttt{0.0824} = 824 \cdot 10^{-4}$$

- Zahl ist $\mathit{Signifikand} \times 10^{\mathit{Exponent}}$

Fliesskommazahlen

- feste Anzahl signifikante Stellen (z.B. 10)
- plus Position des Kommas

`82.4` = $824 \cdot 10^{-1}$

`0.0824` = $824 \cdot 10^{-4}$

- Zahl ist *Signifikand* $\times 10^{\textit{Exponent}}$

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 (`double` hat mehr Stellen als `float`)
- sind auf vielen Rechnern sehr schnell

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 (`double` hat mehr Stellen als `float`)
- sind auf vielen Rechnern sehr schnell

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 (`double` hat mehr Stellen als `float`)
- sind auf vielen Rechnern sehr schnell

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 (`double` hat mehr Stellen als `float`)
- sind auf vielen Rechnern sehr schnell

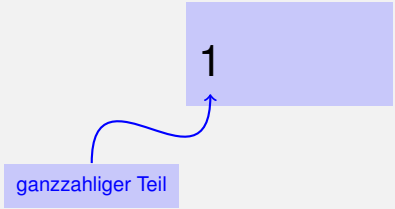
Arithmetische Operatoren

Wie bei `int`, aber ...

- Divisionsoperator `/` modelliert „echte” (reelle, nicht ganzzahlige) Division
- Keine Modulo-Operatoren `%` oder `%=`

Literale

unterscheiden sich von Ganzzahlliteralen

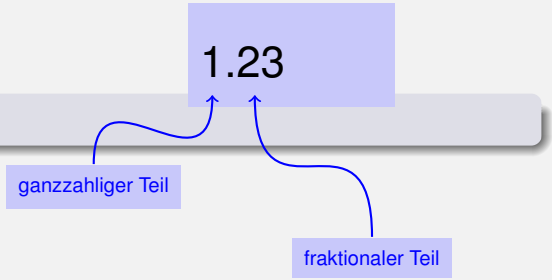


Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma

`1.0` : Typ `double`, Wert 1



Literale

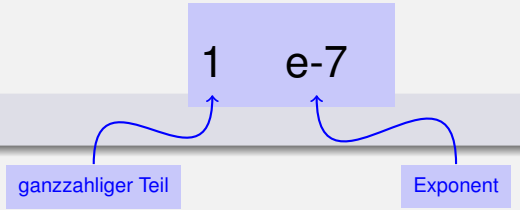
unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma

  `1.0` : Typ `double`, Wert 1

- oder Exponent.

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

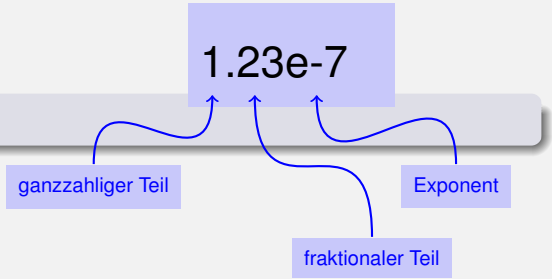


Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma

`1.0` : Typ `double`, Wert 1



- und / oder Exponent.

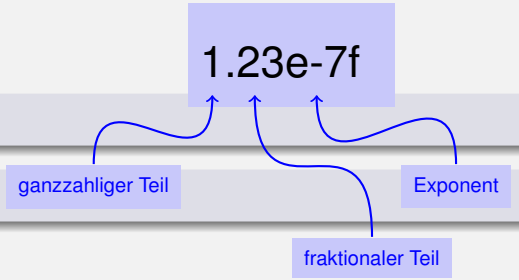
`1e3` : Typ `double`, Wert 1000

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

Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma
- und / oder Exponent.



`1.0` : Typ `double`, Wert 1

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

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

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

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

Rechnen mit `float`: Beispiel

Approximation der Euler-Zahl

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

mittels der ersten 10 Terme.

Rechnen mit `float`: Eulersche Zahl

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

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

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

std::cout << "Approximating the Euler number... \n";
// steps 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.
- 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.

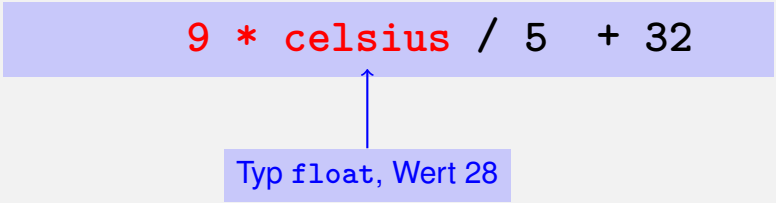
Gemischte Ausdrücke, Konversion

- 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.



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.

Gemischte Ausdrücke, Konversion

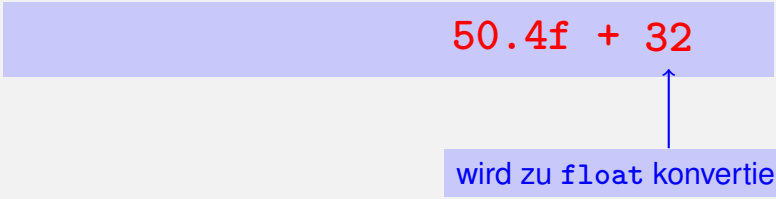
- Fliesskommazahlen sind allgemeiner als ganzzahlige Typen.
- In gemischten Ausdrücken werden ganze Zahlen zu Fliesskommazahlen konvertiert.

```
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.



Gemischte Ausdrücke, Konversion

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

```
82.4f
```

Wertebereich

Ganzzahlige Typen:

- Über- und Unterlauf häufig, aber ...

Wertebereich

Ganzzahlige Typen:

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

Wertebereich

Fliesskommatypen:

- Über- und Unterlauf selten, aber ...

Wertebereich

Fliesskommatypen:

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

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;

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";
```

Eingabe 1.5

Eingabe 1.0

Eingabe 0.5

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";
```

Eingabe 1.5

Eingabe 1.0

Eingabe 0.5

Ausgabe 0

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";
```

Eingabe 1.1

Eingabe 1.0

Eingabe 0.1

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";
```

Eingabe 1.1

Eingabe 1.0

Eingabe 0.1

Ausgabe 2.23517e-8

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";
```

Eingabe 1.1

Eingabe 1.0

Eingabe 0.1

Ausgabe 2.23517e-8

Was ist denn hier los?

7. 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,

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

- $\beta \geq 2$, die Basis,
- $p \geq 1$, die Präzision (Stellenzahl),

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,

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

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

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

Fliesskommazahlensysteme

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

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

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

In Basis-β-Darstellung:

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

Fliesskommazahlensysteme

Beispiel

- $\beta = 10$

Darstellungen der Dezimalzahl 0.1

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

Normalisierte Darstellung

Normalisierte Zahl:

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

Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

Normalisierte Darstellung

Normalisierte Zahl:

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

Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

Normalisierte Darstellung

Normalisierte Zahl:

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

Bemerkung 2

Die Zahl 0 (und alle Zahlen kleiner als $\beta^{e_{\min}}$) haben keine normalisierte Darstellung (werden wir später beheben)!

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 $F^*(2, 3, -2, 2)$ (nur positive Zahlen)

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

Normalisierte Darstellung

Beispiel $F^*(2, 3, -2, 2)$ (nur positive Zahlen)

$d_0{\bullet}d_1d_2$	$\mathbf{e = -2}$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
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

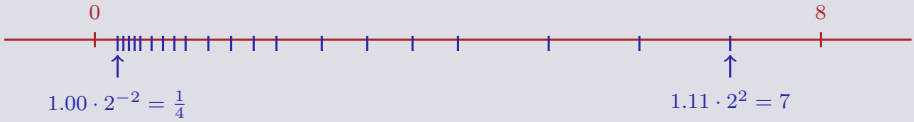
Beispiel $F^*(2, 3, -2, \mathbf{2})$ (nur positive Zahlen)

$d_0 \bullet d_1 d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$\mathbf{e = 2}$
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



Binäre und dezimale Systeme

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

Binäre und dezimale Systeme

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

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

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

Umrechnung ($0 < x < 2$)

Berechnung der *Binärdarstellung*:

$$x = b_0 \bullet b_{-1} b_{-2} b_{-3} \ldots$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$\begin{aligned} x &= b_0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\ &= b_0 + 0 \bullet b_{-1} b_{-2} b_{-3} \ldots \end{aligned}$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$\begin{aligned} x &= b_0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\ &= b_0 + 0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\ &\Longrightarrow \end{aligned}$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$\begin{aligned} x &= b_0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\ &= b_0 + 0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\ &\Longrightarrow \\ (x - b_0) &= 0 \bullet b_{-1} b_{-2} b_{-3} b_{-4} \ldots \end{aligned}$$

Umrechnung $(0 < x < 2)$

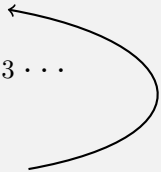
Berechnung der *Binärdarstellung*:

$$\begin{aligned} x &= b_0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\ &= b_0 + 0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\ &\Longrightarrow \\ 2 \cdot (x - b_0) &= b_{-1} \bullet b_{-2} b_{-3} b_{-4} \ldots \end{aligned}$$

Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

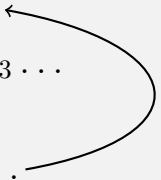
$$
\begin{aligned}
x &= b_0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\
&= b_0 + 0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\
&\Longrightarrow \\
2 \cdot (x - b_0) &= b_{-1} \bullet b_{-2} b_{-3} b_{-4} \ldots
\end{aligned}
$$



Umrechnung $(0 < x < 2)$

Berechnung der *Binärdarstellung*:

$$
\begin{aligned}
x &= b_0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\
&= b_0 + 0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\
&\Longrightarrow \\
2 \cdot (x - b_0) &= b_{-1} \bullet b_{-2} b_{-3} b_{-4} \ldots
\end{aligned}
$$



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

Beispiel (binär)

$$
\begin{aligned}
x &= \mathbf{1}.01011 \\
&= \mathbf{1} + 0.01011 \\
&\Longrightarrow \\
2 \cdot (x - \mathbf{1}) &= 0.1011
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= 1.\mathbf{01011} \\
&= 1 + 0.01011 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= \mathbf{0.1011}
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= \mathbf{0}.1011 \\
&= \mathbf{0} + 0.1011 \\
&\Longrightarrow \\
2 \cdot (x - \mathbf{0}) &= 1.011
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= 0.\mathbf{1011} \\
&= 0 + 0.1011 \\
&\Longrightarrow \\
2 \cdot (x - 0) &= \mathbf{1.011}
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= 1.011 \\
&= 1 + 0.011 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= 0.11
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= 1.\mathbf{011} \\
&= 1 + 0.011 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= \mathbf{0.11}
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= \mathbf{0}.11 \\
&= \mathbf{0} + 0.11 \\
&\Longrightarrow \\
2 \cdot (x - \mathbf{0}) &= 1.1
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= 0.\mathbf{11} \\
&= 0 + 0.11 \\
&\Longrightarrow \\
2 \cdot (x - 0) &= \mathbf{1.1}
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= 1.1 \\
&= 1 + 0.1 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= 1
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= 1.\mathbf{1} \\
&= 1 + 0.1 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= \mathbf{1}
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= 1 \\
&= 1 + 0 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= 0
\end{aligned}
$$

Beispiel (binär)

$$
\begin{aligned}
x &= 1 \\
&= 1 + 0 \\
&\Longrightarrow \\
2 \cdot (x - 1) &= \mathbf{0}
\end{aligned}
$$

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$	0.8	1.6

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$	0.8	1.6
1.6	$b_{-4} = \mathbf{1}$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$	0.8	1.6
1.6	$b_{-4} = \mathbf{1}$	0.6	1.2

Binärdarstellung von 1.1

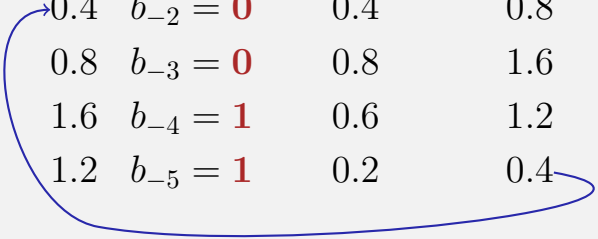
x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$	0.8	1.6
1.6	$b_{-4} = \mathbf{1}$	0.6	1.2
1.2	$b_{-5} = \mathbf{1}$		

Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$	0.8	1.6
1.6	$b_{-4} = \mathbf{1}$	0.6	1.2
1.2	$b_{-5} = \mathbf{1}$	0.2	0.4

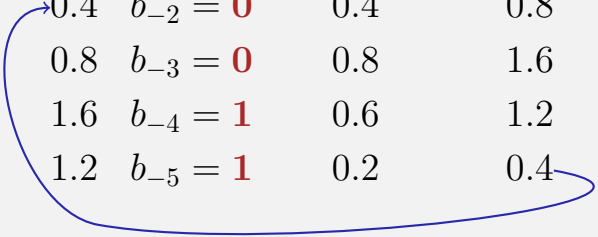
Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$	0.8	1.6
1.6	$b_{-4} = \mathbf{1}$	0.6	1.2
1.2	$b_{-5} = \mathbf{1}$	0.2	0.4



Binärdarstellung von 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$	0.8	1.6
1.6	$b_{-4} = \mathbf{1}$	0.6	1.2
1.2	$b_{-5} = \mathbf{1}$	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
- `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

- 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

- 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:

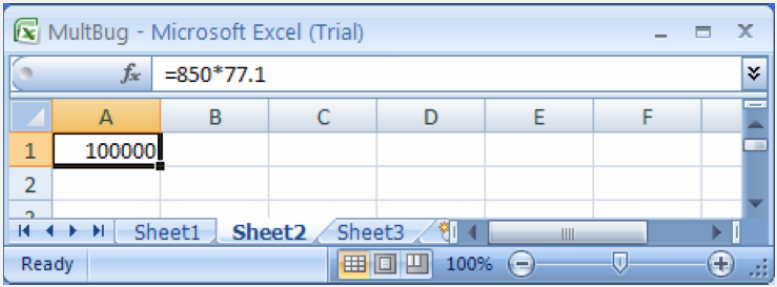
$$\begin{aligned} \texttt{1.1} &= \underline{1.1000000000000000}888178\ldots \\ \texttt{1.1f} &= \underline{1.1000000}238418\ldots \end{aligned}$$

Der Excel-2007-Bug

```
std::cout << 850 * 77.1; // 65535
```

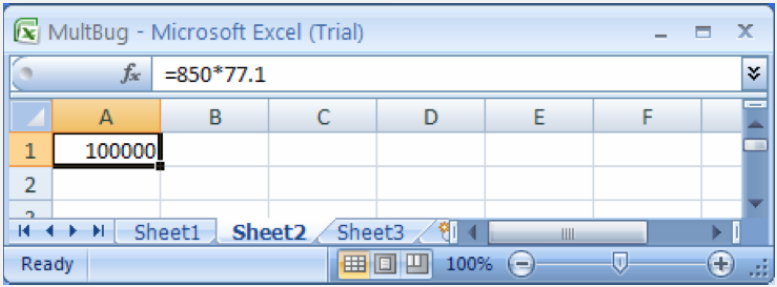
Der Excel-2007-Bug

```
std::cout << 850 * 77.1; // 65535
```



Der Excel-2007-Bug

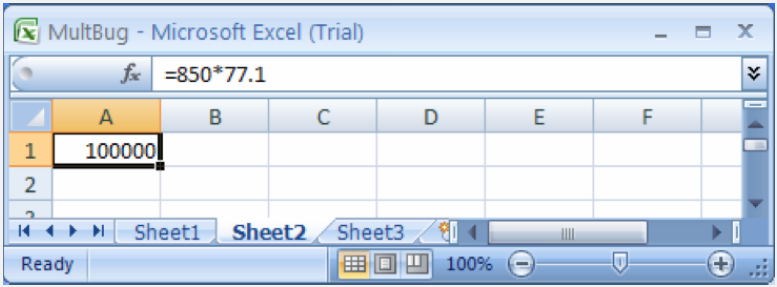
```
std::cout << 850 * 77.1; // 65535
```



- 77.1 hat keine endliche Binärdarstellung, wir erhalten $65534.9999999999927\ldots$

Der Excel-2007-Bug

```
std::cout << 850 * 77.1; // 65535
```



- 77.1 hat keine endliche Binärdarstellung, wir erhalten $65534.9999999999927\ldots$
- Für diese und genau 11 andere “seltene” Zahlen war die Ausgabe (und nur diese) fehlerhaft.

Rechnen mit Fliesskommazahlen

ist fast so einfach wie mit ganzen Zahlen.

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{aligned} & 1.111 \cdot 2^{-2} \\ + \quad & 1.011 \cdot 2^{-1} \end{aligned}$$

1. Exponenten anpassen durch Denormalisieren einer Zahl

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{rl} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \checkmark \end{array}$$

1. Exponenten anpassen durch Denormalisieren einer Zahl

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline \end{array}$$

2. Binäre Addition der Signifikanden

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline & = 100.101 \cdot 2^{-2} \checkmark \end{array}$$

2. Binäre Addition der Signifikanden

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{rr} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline & = 100.101 \cdot 2^{-2} \end{array}$$

3. Renormalisierung

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{aligned} & 1.111 \cdot 2^{-2} \\ + \quad & 10.110 \cdot 2^{-2} \\ \hline & = 1.00101 \cdot 2^{0} \checkmark \end{aligned}$$

3. Renormalisierung

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{array}{rl} & 1.111 \cdot 2^{-2} \\ + & 10.110 \cdot 2^{-2} \\ \hline & = 1.00101 \cdot 2^{0} \end{array}$$

4. Runden auf p signifikante Stellen, falls nötig

Rechnen mit Fliesskommazahlen

Beispiel ($\beta = 2$, $p = 4$):

$$\begin{aligned} & 1.111 \cdot 2^{-2} \\ + \quad & 10.110 \cdot 2^{-2} \\ \hline & = 1.001 \cdot 2^{0} \checkmark \end{aligned}$$

4. Runden auf p signifikante Stellen, falls nötig

Der IEEE Standard 754

- legt Fliesskommazahlensysteme und deren Rundungsverhalten fest
- wird fast überall benutzt

Der IEEE Standard 754

- legt Fliesskommazahlensysteme und deren Rundungsverhalten fest
- wird fast überall benutzt

Der IEEE Standard 754

- Single precision (`float`) Zahlen:

  $F^*(2, 24, -126, 127)$ plus $0, \infty, \ldots$

- Double precision (`double`) Zahlen:

  $F^*(2, 53, -1022, 1023)$ plus $0, \infty, \ldots$

- Alle arithmetischen Operationen runden das *exakte* Ergebnis auf die nächste darstellbare Zahl

Der IEEE Standard 754

- Single precision (`float`) Zahlen:

  $F^*(2, 24, -126, 127)$ plus $0, \infty, \ldots$

- Double precision (`double`) Zahlen:

  $F^*(2, 53, -1022, 1023)$ plus $0, \infty, \ldots$

- Alle arithmetischen Operationen runden das *exakte* Ergebnis auf die nächste darstellbare Zahl

Der IEEE Standard 754

Warum

$$F^*(2, 24, -126, 127)?$$

- 1 Bit für das Vorzeichen
- 23 Bit für den Signifikanden
- 8 Bit für den Exponenten

$\Rightarrow$ insgesamt 32 Bit.

Der IEEE Standard 754

Warum

$$F^*(2, 24, -126, 127)?$$

- 1 Bit für das Vorzeichen
- 23 Bit für den Signifikanden
- 8 Bit für den Exponenten

$\Rightarrow$ insgesamt 32 Bit.

Der IEEE Standard 754

Warum

$$F^*(2, 24, -126, 127)?$$

- 1 Bit für das Vorzeichen
- 23 Bit für den Signifikanden
- 8 Bit für den Exponenten (256 mögliche Werte)

$\Rightarrow$ insgesamt 32 Bit.

Der IEEE Standard 754

Warum

$$F^*(2, 24, -126, 127)?$$

- 1 Bit für das Vorzeichen
- 23 Bit für den Signifikanden
- 8 Bit für den Exponenten (254 mögliche Exponenten, 2 Spezialwerte: 0, ∞,...)

$\Rightarrow$ insgesamt 32 Bit.

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";
```

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!

$$\begin{aligned} & 1.000 \cdot 2^5 \\ +& 1.000 \cdot 2^0 \end{aligned}$$

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

$$\begin{aligned} & 1.000 \cdot 2^5 \\ +& 1.000 \cdot 2^0 \\ =& \, 1.00001 \cdot 2^5 \end{aligned}$$

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

$$
\begin{aligned}
& 1.000 \cdot 2^5 \\
+ & 1.000 \cdot 2^0 \\
= & \ 1.00001 \cdot 2^5 \\
\text{“=”} & \ 1.000 \cdot 2^5 \quad \text{(Rundung auf 4 Stellen)}
\end{aligned}
$$

Regel 2

Addiere keine zwei Zahlen sehr unterschiedlicher Grösse!

$$
\begin{aligned}
& 1.000 \cdot 2^5 \\
+ & 1.000 \cdot 2^0 \\
= & \ 1.00001 \cdot 2^5 \\
\text{“=”} & \ 1.000 \cdot 2^5 \quad \text{(Rundung auf 4 Stellen)}
\end{aligned}
$$

Addition von 1 hat keinen Effekt!

- Die n-te Harmonische Zahl ist

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n.$$

- 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.

Harmonische Zahlen

Regel 2

```
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";
```

Harmonische Zahlen


```
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";
```

Eingabe: **10000000**

Vorwärts: **15.4037**

Rückwärts: **16.686**

Harmonische Zahlen


```
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";
```

Eingabe: **100000000**

Vorwärts: **15.4037**

Rückwärts: **18.8079**

Harmonische Zahlen — Regel 2

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.

Harmonische Zahlen

Regel 2

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

- Bei $1 + 1/2 + 1/3 + \cdots$ sind späte Terme zu klein, um noch beizutragen.
- Problematik wie bei $2^5 + 1$ “=” 2^5

Harmonische Zahlen

Regel 2

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

- Bei $1 + 1/2 + 1/3 + \cdots$ sind späte Terme zu klein, um noch beizutragen.
- Problematik wie bei $2^5 + 1$ “=” 2^5

Beobachtung:

- Die Vorwärtssumme wächst irgendwann nicht mehr und ist “richtig” falsch.
- Die Rückwärtssumme approximiert H_n gut.

Erklärung:

- Bei $1 + 1/2 + 1/3 + \cdots$ 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

8. Funktionen I

Funktionsdefinitionen- und Aufrufe, Auswertung von Funktionsaufrufen, Der Typ `void`, Vor- und Nachbedingungen

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;

std::cout << a << "^" << n << " = " << pow(a,n) << ".\n";
```

"Funktion pow"

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;
}
```

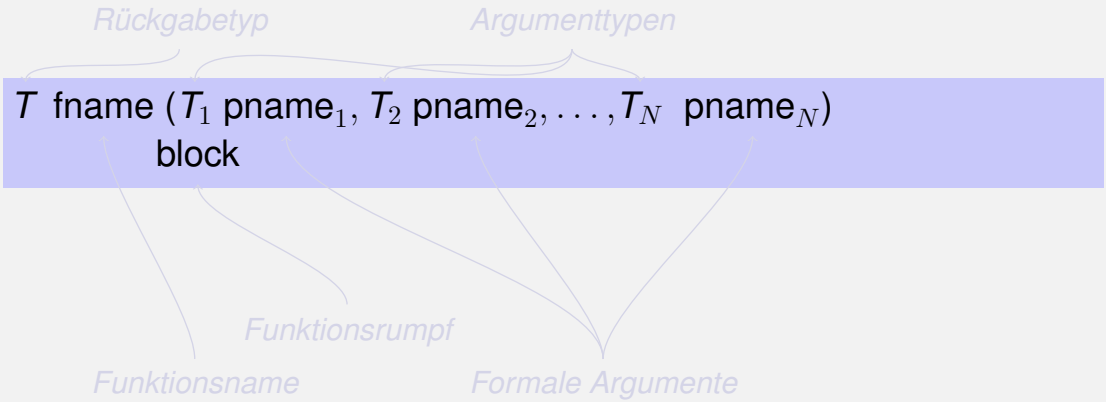
Funktion zur Potenzberechnung

```
double pow(double b, int e){...}
```

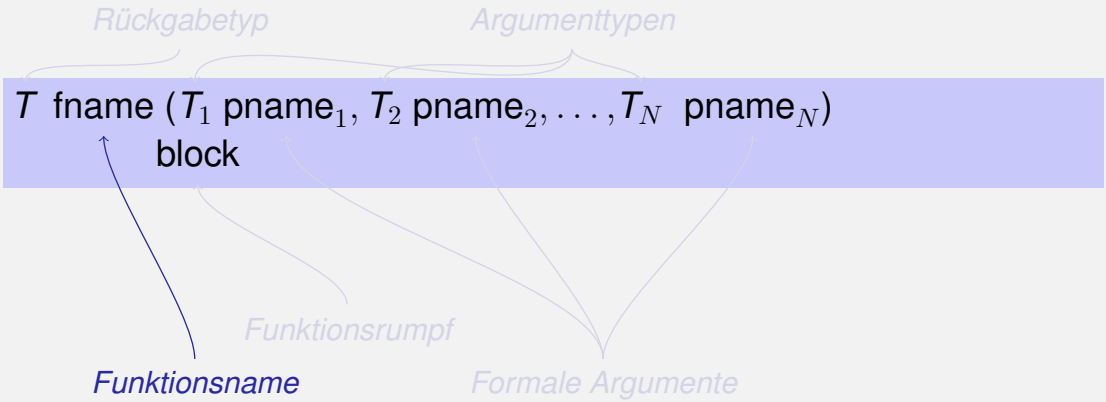
Funktion zur Potenzberechnung

```
// 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;
}
```

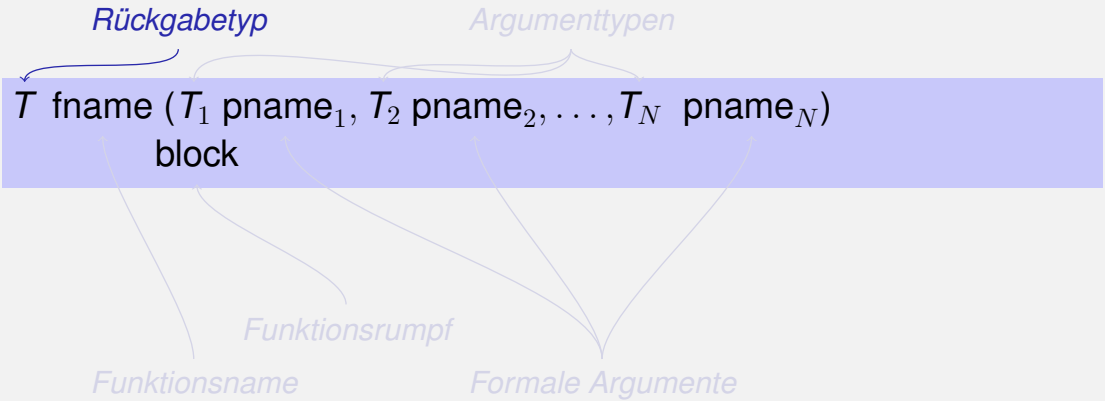
Funktionsdefinitionen



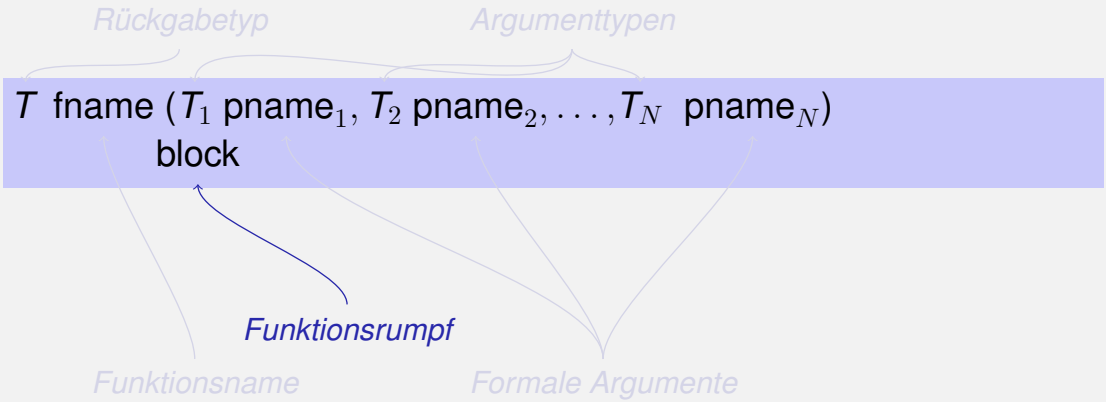
Funktionsdefinitionen



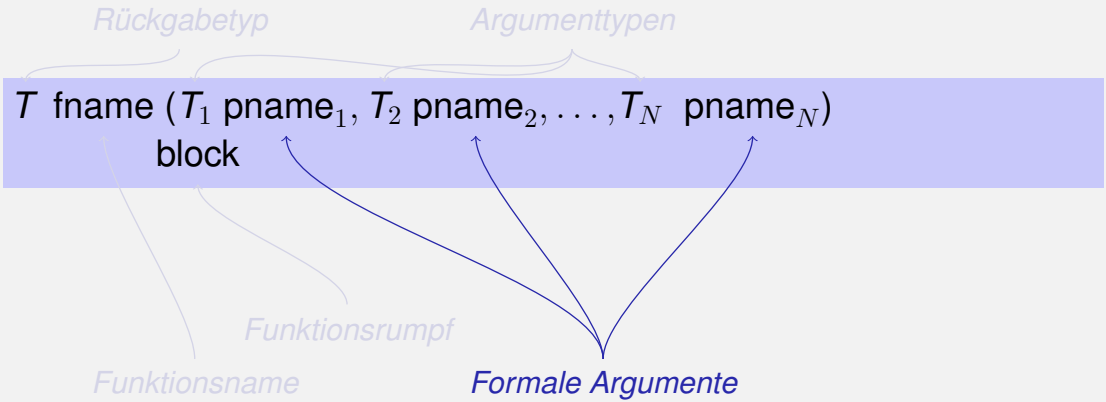
Funktionsdefinitionen



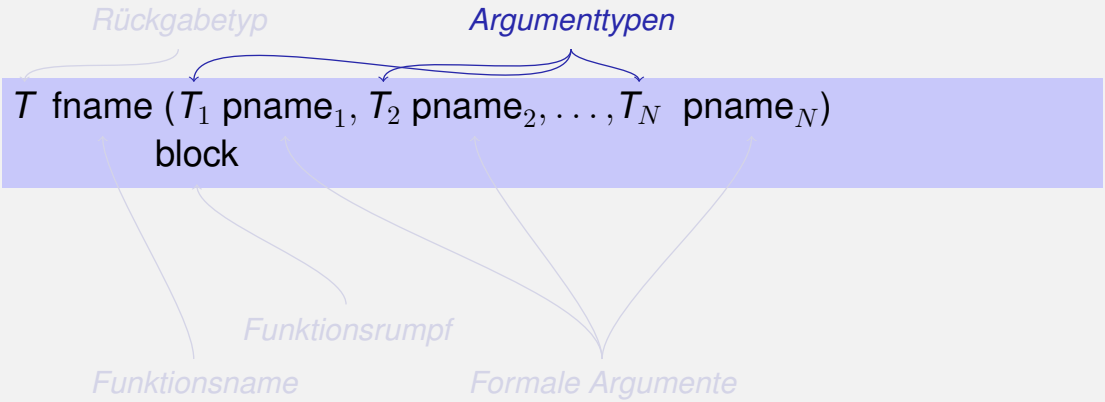
Funktionsdefinitionen



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 ($expression_1$, $expression_2$, ..., $expression_N$)

- Alle Aufrufargumente müssen konvertierbar sein in die entsprechenden Argumenttypen.
- Der Funktionsaufruf selbst ist ein Ausdruck vom Rückgabetyp.

Beispiel: `pow(a,n)`: Ausdruck vom Typ `double`

Funktionsaufrufe

fname ($expression_1$, $expression_2$, ..., $expression_N$)

- Alle Aufrufargumente müssen konvertierbar sein in die entsprechenden Argumenttypen.
- Der Funktionsaufruf selbst ist ein Ausdruck vom Rückgabetyp.

Beispiel: `pow(a,n)`: Ausdruck vom Typ `double`

Funktionsaufrufe

fname ($expression_1$, $expression_2$, ..., $expression_N$)

- Alle Aufrufargumente müssen konvertierbar sein in die entsprechenden Argumenttypen.
- Der Funktionsaufruf selbst ist ein Ausdruck vom Rückgabetyp.

Beispiel: `pow(a,n)`: Ausdruck vom Typ `double`

Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

- Aufrufargumente sind R-Werte
- Funktionsaufruf selbst ist R-Wert.

Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

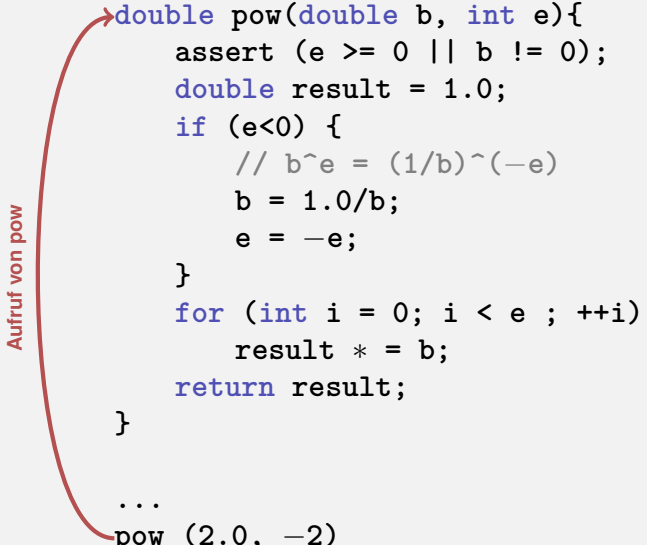
- Aufrufargumente sind R-Werte
- Funktionsaufruf selbst ist R-Wert.

fname: R-Wert $\times$ R-Wert $\times \cdots \times$ R-Wert $\longrightarrow$ R-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




```
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;
}

...
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)
```

b=2.0,e=-2

// ok

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

```
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)
```

e == -2

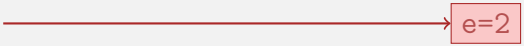
Auswertung Funktionsaufruf

```
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)
```

b=0.5

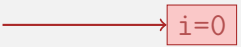
Auswertung Funktionsaufruf

```
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

```
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

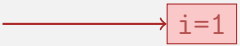
```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

i=0

result=0.5

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

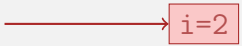
```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
pow (2.0, -2)
```

i=1

result=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)
```



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

Auswertung Funktionsaufruf

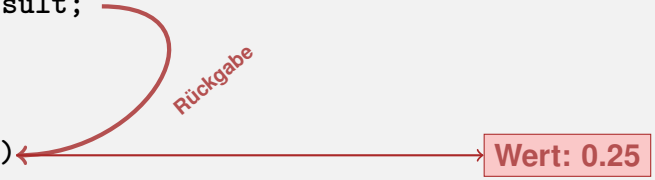
```
double pow(double b, int e){
    assert (e >= 0 || b != 0);
    double result = 1.0;
    if (e<0) {
        // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        result * = b;
    return result;
}

...
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)
```



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`

Der Typ `void`

Pathetic earthlings.
Hurling your bodies out into the void, without the slightest inkling of who or what is out here.

Emperor Ming

Der Typ `void`

- Fundamentaler Typ mit leerem Wertebereich

Der Typ `void`

- Fundamentaler Typ mit leerem Wertebereich
- Verwendung als Rückgabetyp für Funktionen, die *nur* einen Effekt haben

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;
}
```

`void`-Funktionen

- benötigen kein `return`.
- Ausführung endet, wenn Ende des Funktionsrumpfes erreicht wird oder
- `return;` erreicht wird

Vorbedingungen

Vorbedingung (precondition):

- Was muss bei Funktionsaufruf gelten?
- Spezifiziert *Definitionsbereich* der Funktion.

Vorbedingungen

Vorbedingung (precondition):

- Was muss bei Funktionsaufruf gelten?
- Spezifiziert *Definitionsbereich* 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:
- *Wenn* die Vorbedingung beim Funktionsaufruf gilt, *dann* gilt auch die Nachbedingung nach dem Funktionsaufruf.

Funktion `pow`: funktioniert für alle Basen $b \neq 0$

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 $b \neq 0$

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 b^e
```

ist formal inkorrekt:

- Überlauf, falls e oder b zu gross sind
- b^e vielleicht nicht als double Wert darstellbar (Löcher im Wertebereich)

Fromme Lügen...

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
```

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;
}
```

9. Funktionen II

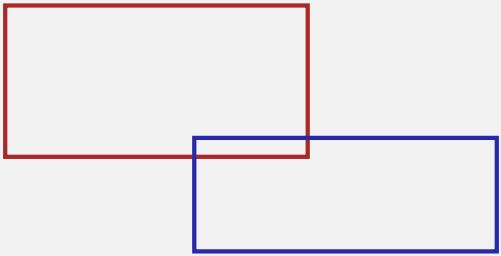
Stepwise Refinement, Gültigkeitsbereich, Bibliotheken, Standardfunktionen

Stepwise Refinement

- Einfache *Programmiertechnik* zum Lösen komplexer Probleme

Beispielproblem

Finde heraus, ob sich zwei Rechtecke schneiden!



Top-Down Ansatz

- Formulierung einer groben Lösung mit Hilfe von
  - Kommentaren
  - fiktiven Funktionen
- Wiederholte Verfeinerung:
  - Kommentare $\longrightarrow$ Programmtext
  - fiktive Funktionen $\longrightarrow$ Funktionsdefinitionen

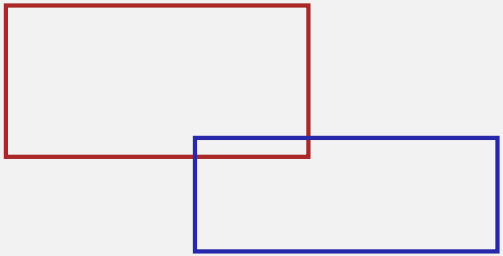
Top-Down Ansatz

- Formulierung einer groben Lösung mit Hilfe von
  - Kommentaren
  - fiktiven Funktionen
- Wiederholte Verfeinerung:
  - Kommentare $\longrightarrow$ Programmtext
  - fiktive Funktionen $\longrightarrow$ Funktionsdefinitionen

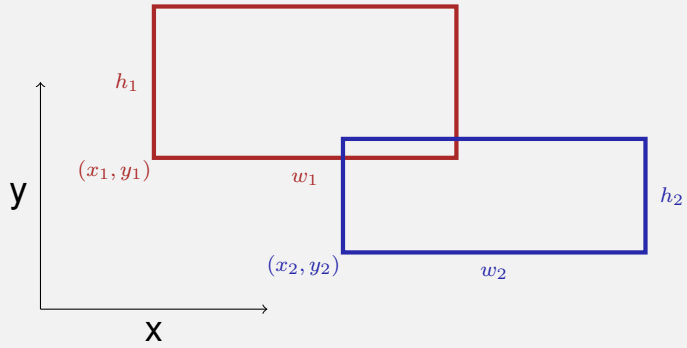
Grobe Lösung

```
int main()
{
    // Eingabe Rechtecke

    // Schnitt?

    // Ausgabe der Loesung

    return 0;
}
```

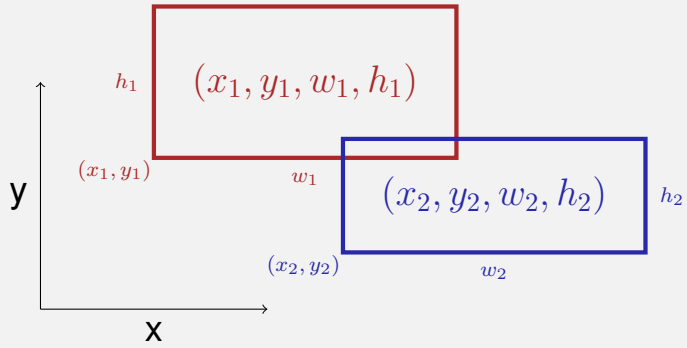
Verfeinerung 1: Eingabe Rechtecke



Verfeinerung 1: Eingabe Rechtecke

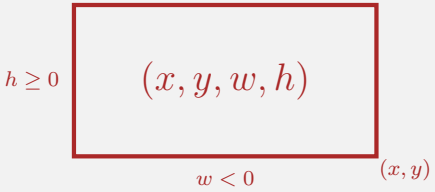


Verfeinerung 1: Eingabe Rechtecke



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
}
```

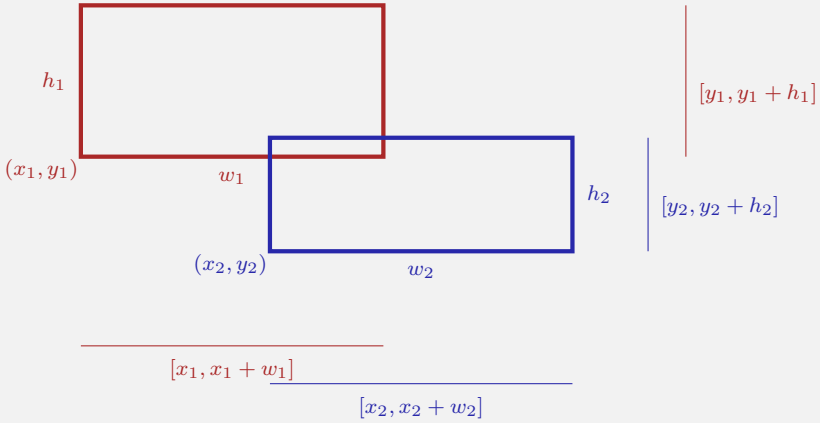
Funktion main ✓

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;
}

// POST: the minimum of x and y is returned
int min (int x, int y){
    if (x<y) return x; else return y;
}
```

gibt es schon in der Standardbibliothek

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

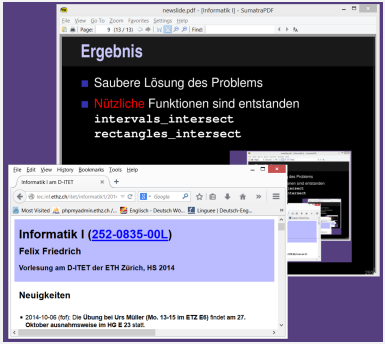
  `intervals_intersect`

  `rectangles_intersect`

Ergebnis

- Saubere Lösung des Problems
- Nützliche Funktionen sind entstanden

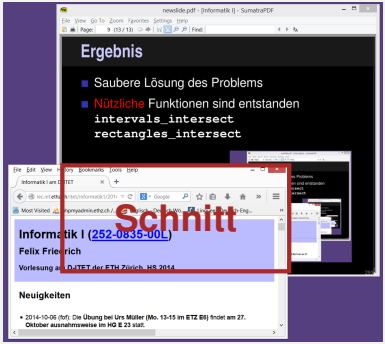
```
intervals_intersect
rectangles_intersect
```



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;
}

int f (int i) // Gueltigkeitsbereich von f ab hier
{
    return i;
}
```

Gültigkeit f

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;
}

int f (int i) // Gueltigkeitsbereich von f ab hier
{
    return i;
}
```

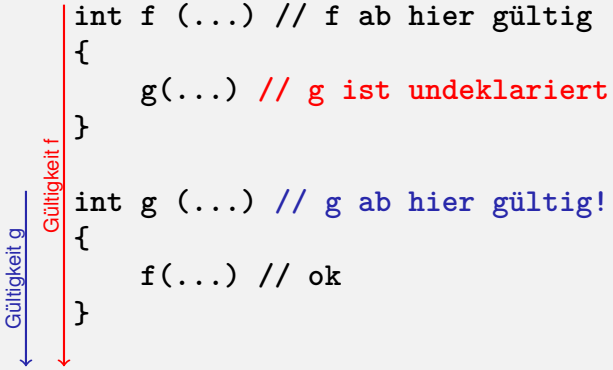
Gültigkeit f

...aber so!

```
#include<iostream>
int f (int i); // Gueltigkeitsbereich von f ab hier

int main()
{
    std::cout << f(1);
    return 0;
}

int f (int i)
{
    return i;
}
```

Forward Declarations, wozu?

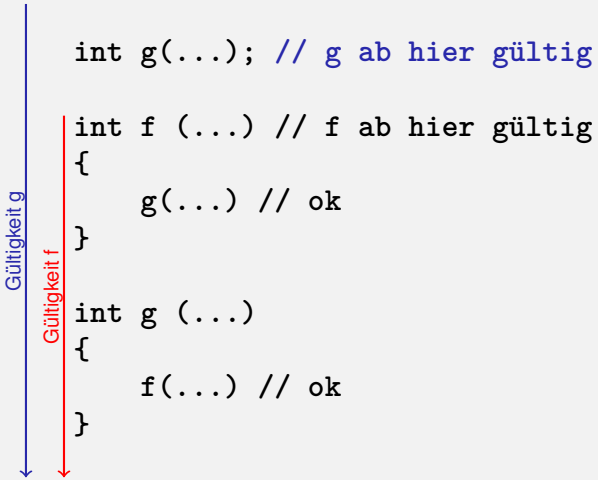
Funktionen, die sich gegenseitig aufrufen:




```
int f (...) // f ab hier gültig
{
    g(...) // g ist undeklariert
}

int g (...) // g ab hier gültig!
{
    f(...) // ok
}
```

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** `math.cpp`

Level 1: Inkludieren der Funktion

```
// Prog: callpow2.cpp
// Call a function for computing powers.

#include <iostream>
#include "math.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 "math.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;
}
```

im Arbeitsverzeichnis

Nachteil des Inkludierens

- `#include` kopiert die Datei (`math.cpp`) in das Hauptprogramm (`callpow2.cpp`).

Nachteil des Inkludierens

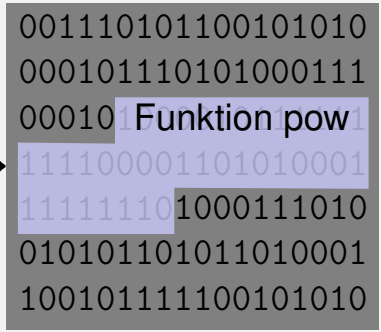
- `#include` kopiert die Datei (`math.cpp`) in das Hauptprogramm (`callpow2.cpp`).
- Der Compiler muss die Funktionsdefinition für jedes Programm neu übersetzen.

Level 2: Getrennte Übersetzung

```
double pow(double b,
           int e)
{
    ...
}
```

math.cpp

g++ -c math.cpp



math.o

Level 2: Getrennte Übersetzung

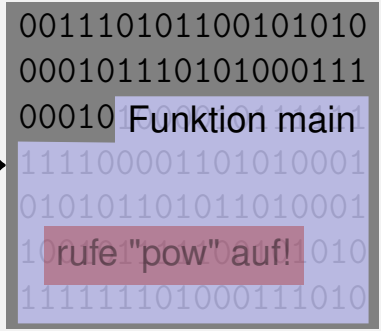
```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow (double b, int e);
```

math.h

Level 2: Getrennte Übersetzung

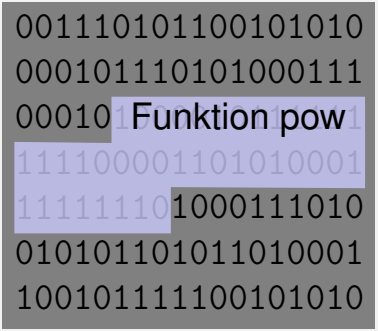
```
#include <iostream>
#include "math.h"
int main()
{
  std::cout << pow(2,-2) << "\n";
  return 0;
}
```

callpow3.cpp



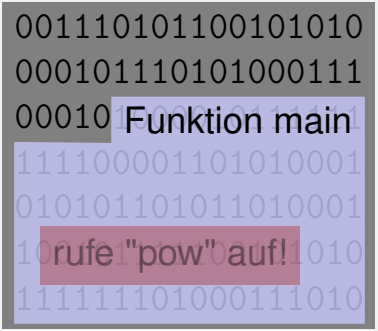
callpow3.o

Der Linker vereint...



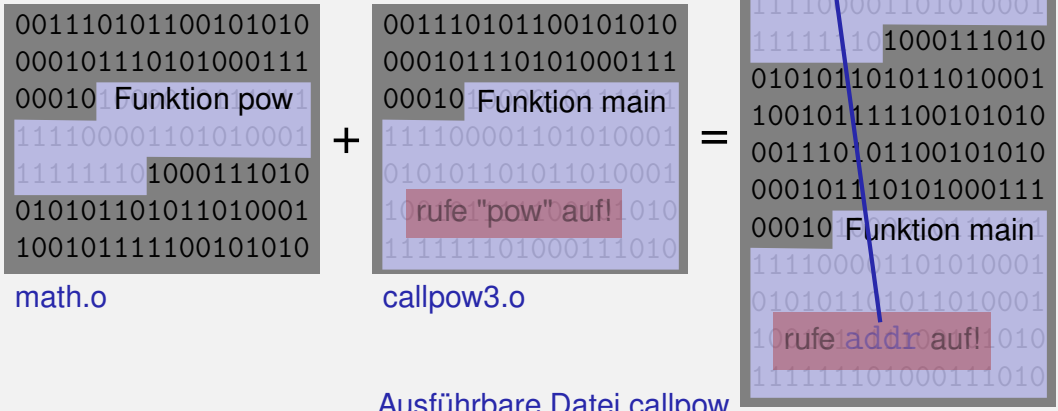
math.o

+



callpow3.o

... was zusammengehört



Verfügbarkeit von Quellcode?

Beobachtung

`math.cpp` (Quellcode) wird nach dem Erzeugen von `math.o` (Object Code) nicht mehr gebraucht.

Verfügbarkeit von Quellcode?

Beobachtung

`math.cpp` (Quellcode) wird nach dem Erzeugen von `math.o` (Object Code) nicht mehr gebraucht.

Viele Anbieter von Funktionsbibliotheken liefern dem Benutzer keinen Quellcode.

Verfügbarkeit von Quellcode?

Beobachtung

`math.cpp` (Quellcode) wird nach dem Erzeugen von `math.o` (Object Code) nicht mehr gebraucht.

Viele Anbieter von Funktionsbibliotheken liefern dem Benutzer keinen Quellcode.

Header-Dateien sind dann die *einzigen* lesbaren Informationen.

„Open Source” Software

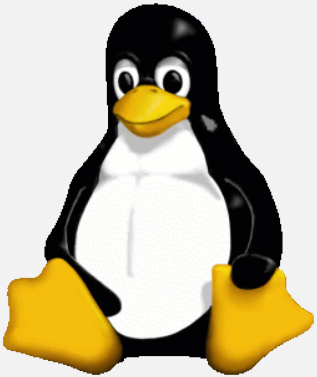
- Alle Quellcodes sind verfügbar.

„Open Source” Software

- Alle Quellcodes sind verfügbar.
- Nur das erlaubt die Weiterentwicklung durch Benutzer und engagierte “Hacker”.

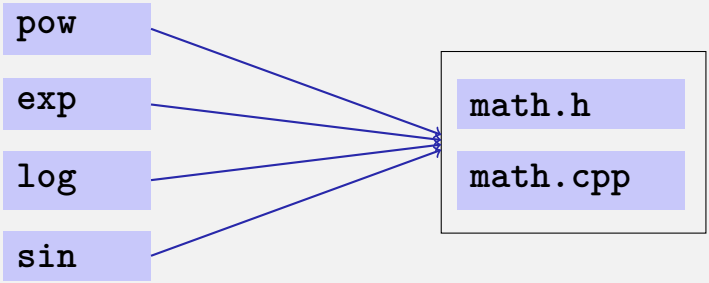
„Open Source” Software

- Alle Quellcodes sind verfügbar.
- Nur das erlaubt die Weiterentwicklung durch Benutzer und engagierte “Hacker”.



Bibliotheken

- Logische Gruppierung ähnlicher Funktionen



Namensräume...

```
// ifeemath.h
// A small library of mathematical functions
namespace ifee {

  // PRE: e >= 0 || b != 0.0
  // POST: return value is b^e
  double pow (double b, int e);

  ....
  double exp (double x);
  ...
}
```

. . . vermeiden Namenskonflikte

```cpp
#include <cmath>
#include "ifeemath.h"

int main()
{
    double x = std::pow (2.0, -2); // <cmath>
    double y = ifee::pow (2.0, -2); // ifeemath.h
}
```

Funktionen aus der Standardbibliothek

- vermeiden die Neuerfindung des Rades (wie bei `ifee::pow`);
- führen auf einfache Weise zu interessanten und effizienten Programmen;

Funktionen aus der Standardbibliothek

- vermeiden die Neuerfindung des Rades (wie bei `ifee::pow`);
- führen auf einfache Weise zu interessanten und effizienten Programmen;
- garantierten einen Qualitäts-Standard, der mit selbstgeschriebenen Funktionen kaum erreicht werden kann.

Primzahltest mit `sqrt`

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \ldots, n-1\}$ 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 $\{2, \ldots, \lfloor \sqrt{n} \rfloor\}$ 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 $\{2, \ldots, \lfloor \sqrt{n} \rfloor\}$ 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).

Primzahltest mit `sqrt`

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \ldots, \lfloor\sqrt{n}\rfloor\}$ 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).
- Andere mathematische Funktionen (`std::pow`,...) sind in der Praxis fast so genau.

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

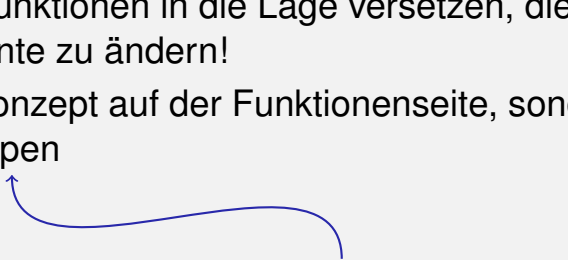
- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern!

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



10. Referenztypen

Referenztypen: Definition und Initialisierung, Call By Value , Call by Reference, Temporäre Objekte, Konstanten, Const-Referenzen

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!
}
```

Referenztypen

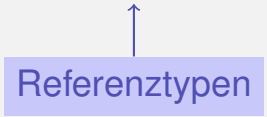
- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern!

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

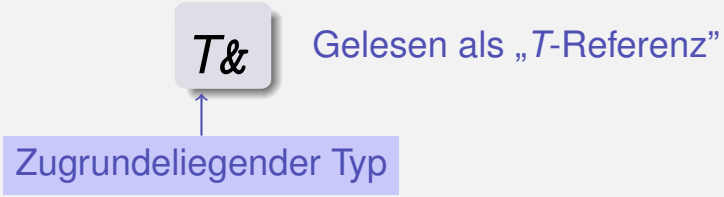
Referenztypen

- 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



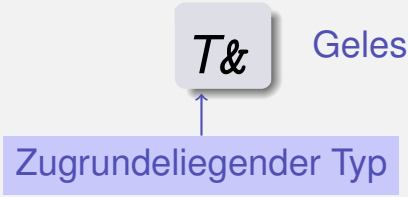
Referenztypen: Definition

Referenztypen: Definition



- *T&* hat den gleichen Wertebereich und gleiche Funktionalität wie *T*, ...

Referenztypen: Definition



- *T&* hat den gleichen Wertebereich und gleiche Funktionalität wie *T*, ...
- nur Initialisierung und Zuweisung funktionieren anders.

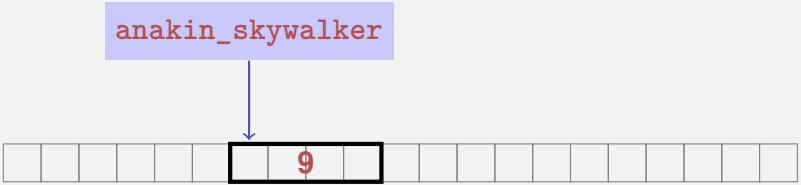
Anakin Skywalker alias Darth Vader

Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker; //  Alias

darth_vader = 22;

std::cout << anakin_skywalker;
```

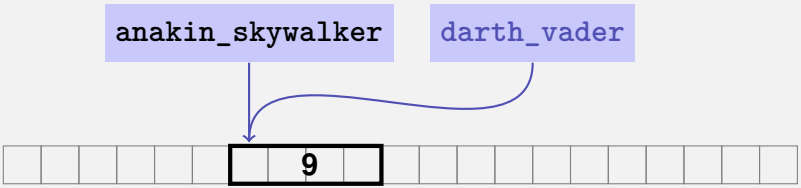
Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker; //  Alias

darth_vader = 22;

std::cout << anakin_skywalker;
```



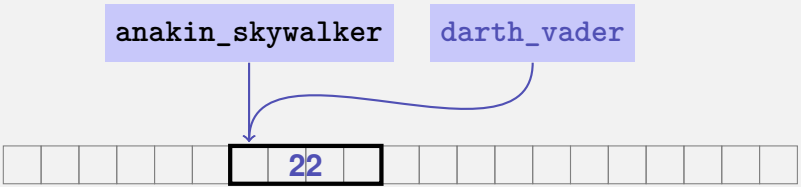
Anakin Skywalker alias Darth Vader

```
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;
```

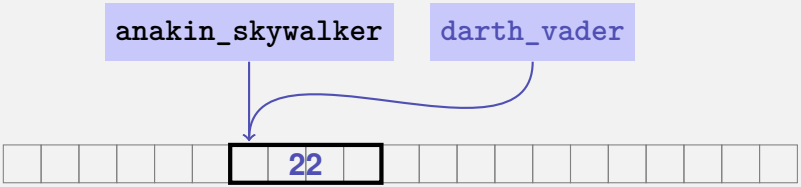


Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker; //  Alias

darth_vader = 22;
```

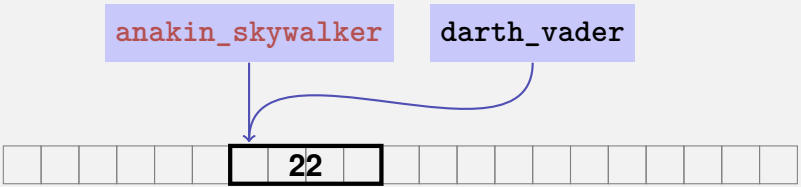
Zuweisung an den L-Wert hinter dem Alias

```
std::cout << anakin_skywalker;
```



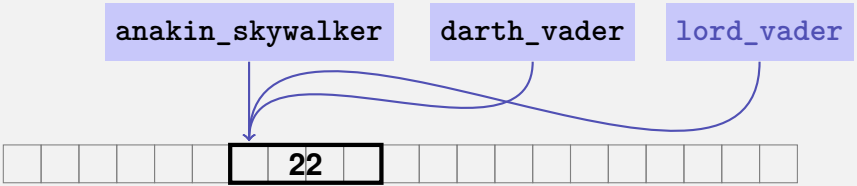
Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker; //  Alias

darth_vader = 22;

std::cout << anakin_skywalker; // 22
```



Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker; //  Alias
int& lord_vader = darth_vader; // noch ein Alias
darth_vader = 22;

std::cout << anakin_skywalker; // 22
```



Referenztypen: Initialisierung & Zuweisung

```
int& darth_vader = anakin_skywalker;
```

- Eine Variable mit Referenztyp (eine *Referenz*) kann nur mit einem L-Wert initialisiert werden.

Referenztypen: Initialisierung & Zuweisung

```
int& darth_vader = anakin_skywalker;
```

- Eine Variable mit Referenztyp (eine *Referenz*) kann nur 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; // anakin_skywalker = 22
```

- Eine Variable mit Referenztyp (eine *Referenz*) kann nur 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: Das Literal 5 hat keine Adresse
```

Call by Reference

```
void increment (int& i)
{
    ++i;
}
...
int j = 5;
increment (j);
std::cout << j << "\n"; // 6
```

Call by Reference

```
void increment (int& i)
{
    ++i;
}
...
int j = 5;
increment (j);
std::cout << j << "\n"; // 6
```

Call by Reference

```
void increment (int& i) ← Initialisierung der formalen Argumente
{ // i wird Alias des Aufrufarguments
    ++i;
}
...
int j = 5;
increment (j);
std::cout << j << "\n"; // 6
```

Call by Reference

```
void increment (int& i)
{
    ++i;
}
...
int j = 5;
increment (j);
std::cout << j << "\n"; // 6
```

Call by Reference

```
void increment (int& i)
{
    ++i;
}
...
int j = 5;
increment (j);
std::cout << j << "\n"; // 6
```

Call by Reference

Formales Argument hat Referenztyp:

$\Rightarrow$ **Call by Reference**

Formales Argument wird (intern) mit der *Adresse* des Aufrufarguments (L-Wert) initialisiert und wird damit zu einem *Alias*.

Call by Value

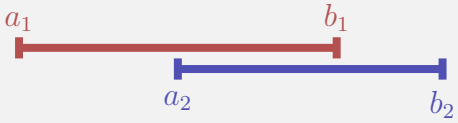
Formales Argument hat keinen Referenztyp:

$\Rightarrow$ **Call by Value**

Formales Argument wird mit dem *Wert* des Aufrufarguments (R-Wert) initialisiert und wird damit zu einer *Kopie*.

Im Kontext: Zuweisung an Referenzen

```
// 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);
  h = std::min (b1, b2);
  return l <= h;
}
...
int lo = 0; int hi = 0;
if (intervals_intersect (lo, hi, 0, 2, 1, 3))
    std::cout << "[" << lo << "," << hi << "]" << "\n"; // [1,2]
```



Im Kontext: Initialisierung von Referenzen

```
// POST: a <= b
void sort (int& a, int& b) {
   if (a > b)
     std::swap (a, b); // 'Durchreichen' der Referenzen a, b
}

bool intervals_intersect (int& l, int& h,
                          int a1, int b1, int a2, int b2) {
  sort (a1, b1); // Erzeugung von Referenzen auf a1, b1
  sort (a2, b2); // Erzeugung von Referenzen auf a2, b2
  l = std::max (a1, a2);
  h = std::min (b1, b2);
  return l <= h;
}
```

Return by Value / Reference

- Auch der Rückgabetyp einer Funktion kann ein Referenztyp sein (return by reference)

Return by Value / Reference

- Auch der Rückgabetyp einer Funktion kann ein Referenztyp sein (return by reference)
- In diesem Fall ist der Funktionsausfruf selbst ein L-Wert

Return by Value / Reference

- Auch der Rückgabetyp einer Funktion kann ein Referenztyp sein (return by reference)
- In diesem Fall ist der Funktionsausfruf selbst ein L-Wert

Return by Value / Reference

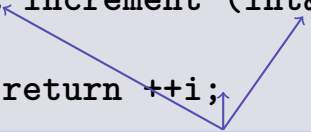
- Auch der Rückgabetyp einer Funktion kann ein Referenztyp sein (return by reference)
- In diesem Fall ist der Funktionsausfruf selbst ein L-Wert

```
int& increment (int& i)
{
    return ++i;
}
```

Return by Value / Reference

- Auch der Rückgabetyp einer Funktion kann ein Referenztyp sein (return by reference)
- In diesem Fall ist der Funktionsausfruf selbst ein L-Wert

```
int& increment (int& i)
{
    return ++i;
}
```



Exakt die Semantik des Prä-Inkrements

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

Rückgabewert vom Typ `int&` wird Alias des formalen Argumentes, dessen Speicherdauer aber nach Auswertung des Funktionsaufrufes endet.

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

Rückgabewert vom Typ `int&` wird Alias des formalen Argumentes, dessen Speicherdauer aber nach Auswertung des Funktionsaufrufes endet.

```
int k = 3;
int& j = foo (k); // j ist Alias einer "Leiche"
std::cout << j << "\n";  // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

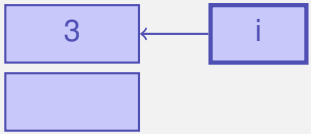
```
int k = 3;
int& j = foo (k); // j ist Alias einer "Leiche"
std::cout << j << "\n";  // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

Wert des Aufrufarguments kommt auf den *Aufrufstapel*



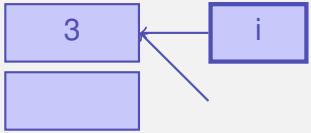

```
int k = 3;
int& j = foo (k); // j ist Alias einer "Leiche"
std::cout << j << "\n";  // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

`i` wird als Referenz zurückgegeben



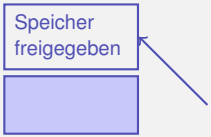

```
int k = 3;
int& j = foo (k); // j ist Alias einer "Leiche"
std::cout << j << "\n";  // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

...und verschwindet vom Aufrufstapel

```
int& foo (int i)
{
    return i;
}
```



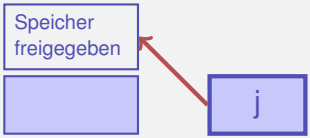

```
int k = 3;
int& j = foo (k); // j ist Alias einer "Leiche"
std::cout << j << "\n";  // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

j wird Alias des freigegebenen Speichers



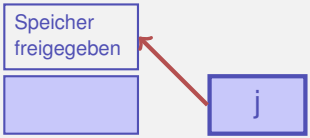

```
int k = 3;
int& j = foo (k); // j ist Alias einer "Leiche"
std::cout << j << "\n";  // undefined behavior
```

Temporäre Objekte

Was ist hier falsch?

```
int& foo (int i)
{
    return i;
}
```

Wert von `j` wird ausgegeben




```
int k = 3;
int& j = foo (k); // j ist Alias einer "Leiche"
std::cout << j << "\n";  // undefined behavior
```

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.

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

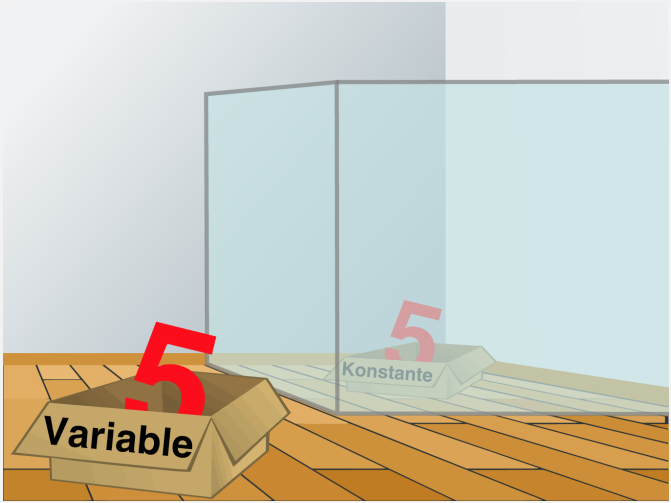
- Compiler kontrolliert Einhaltung des `const`-Versprechens

```
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.

Const-Referenzen

- haben Typ `const` T `&` (= `const` (T `&`))
- können auch mit R-Werten initialisiert werden (Compiler erzeugt temporäres Objekt ausreichender Lebensdauer)

Const-Referenzen

- haben Typ `const` *T* `&` (= `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* `&` (= `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 (flexibel)

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& i = n;
i = 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& i = n; // error: const-qualification is discarded
i = 6;
```

Der 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 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 Wert dahinter nicht verändert werden darf.

```
int n = 5;
const int& i = n;// i: Lese-Alias von n
int& j = n;      // j: Lese-Schreib-Alias
i = 6;           // Fehler: i ist Lese-Alias
j = 6;           // ok: n bekommt Wert 6
```

Wann `const` *T*`&` ?

Regel

Argumenttyp `const` *T* `&` (call by *read-only* reference) wird aus Effizienzgründen anstatt *T* (call by value) benutzt, wenn der Typ *T* grossen Speicherbedarf hat. Für fundamentale Typen (`int`, `double`,...) lohnt es sich aber nicht.

Wann `const` *T*`&` ?

Regel

Argumenttyp `const` *T* `&` (call by *read-only* reference) wird aus Effizienzgründen anstatt *T* (call by value) benutzt, wenn der Typ *T* grossen Speicherbedarf hat. Für fundamentale Typen (`int`, `double`,...) lohnt es sich aber nicht.

Beispiele folgen später in der Vorlesung

11. Felder (Arrays) I

Feldtypen, Sieb des Eratosthenes, Speicherlayout, Iteration, Vektoren, Zeichen und Texte, ASCII, UTF-8, Caesar-Code

Felder: Motivation

- Wir können jetzt über Zahlen iterieren

```
for (int i=0; i<n ; ++i) ...
```

Felder: Motivation

- Wir können jetzt über Zahlen iterieren

```
for (int i=0; i<n ; ++i) ...
```

- .. aber noch nicht über Daten!

Felder: Motivation

- Wir können jetzt über Zahlen iterieren

```
for (int i=0; i<n ; ++i) ...
```

- .. aber noch nicht über Daten!
- Felder speichern *gleichartige* Daten.

Felder: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$

Felder: 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

Felder: 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 ...

Felder: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

Streiche alle echten Vielfachen von 2 ...

Felder: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

... und gehe zur nächsten Zahl

Felder: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

Streiche alle echten Vielfachen von 3 ...

Felder: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

Streiche alle echten Vielfachen von 3 ...

Felder: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

... und gehe zur nächsten Zahl

Felder: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

Am Ende des Streichungsprozesses bleiben nur die Primzahlen übrig.

Felder: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

- Frage: wie streichen wir Zahlen aus ??

Felder: erste Anwendung

Das Sieb des Erathostenes

- berechnet alle Primzahlen $< n$
- Methode: Ausstreichen der Nicht-Primzahlen

- Frage: wie streichen wir Zahlen aus ??
- Antwort: mit einem *Feld* (Array).

Sieb des Eratosthenes: Initialisierung

```
const unsigned int n = 1000;
bool crossed_out[n];
for (unsigned int i = 0; i < n; ++i)
    crossed_out[i] = false;
```

Sieb des Eratosthenes: Initialisierung

```
const unsigned int n = 1000;
bool crossed_out[n];
for (unsigned int i = 0; i < n; ++i)
    crossed_out[i] = false;
```

Konstante!

Sieb des Eratosthenes: Berechnung

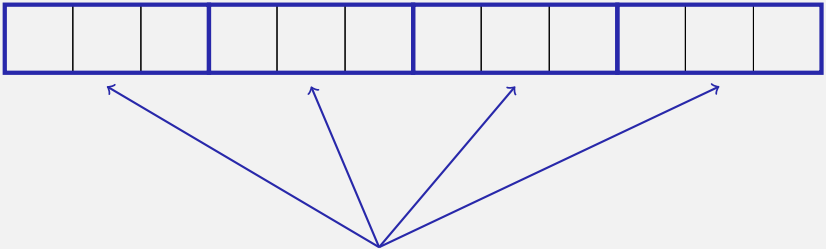
```
for (unsigned int i = 2; i < n; ++i)
    if (!crossed_out[i] ){
      // i is prime
      std::cout << i << " ";
      // cross out all proper multiples of i
      for (unsigned int m = 2*i; m < n; m += i)
        crossed_out[m] = true;
    }
}
```

Speicherlayout eines Feldes

- Ein Feld belegt einen *zusammenhängenden* Speicherbereich

Speicherlayout eines Feldes

- Ein Feld belegt einen *zusammenhängenden* Speicherbereich

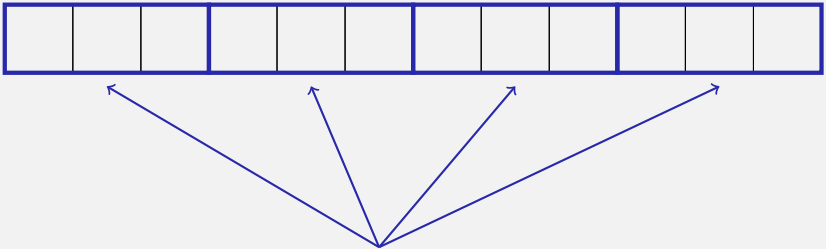


Speicherzellen für jeweils einen Wert vom Typ `T`

Speicherlayout eines Feldes

- Ein Feld belegt einen *zusammenhängenden* Speicherbereich

Beispiel: ein Feld mit 4 Elementen



Speicherzellen für jeweils einen Wert vom Typ `T`

Wahlfreier Zugriff (Random Access)

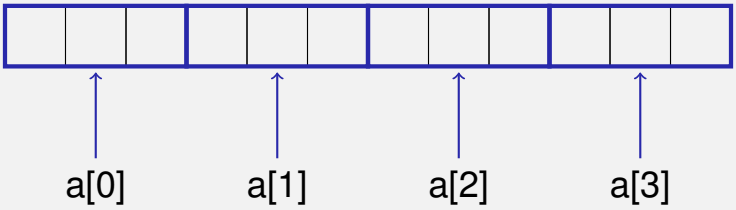
Der L-Wert

hat Typ *T* und bezieht sich auf das i-te Element des Feldes *a* (Zählung ab 0!)

Wahlfreier Zugriff (Random Access)

Der L-Wert

hat Typ *T* und bezieht sich auf das i-te Element des Feldes *a* (Zählung ab 0!)



Wahlfreier Zugriff (Random Access)

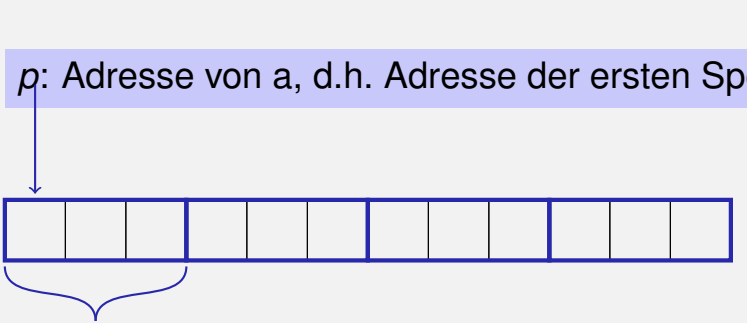
a [*expr*]

Der Wert i von *expr* heisst *Feldindex*.
`[]`: Subskript-Operator

Wahlfreier Zugriff (Random Access)

- Wahlfreier Zugriff ist sehr effizient:

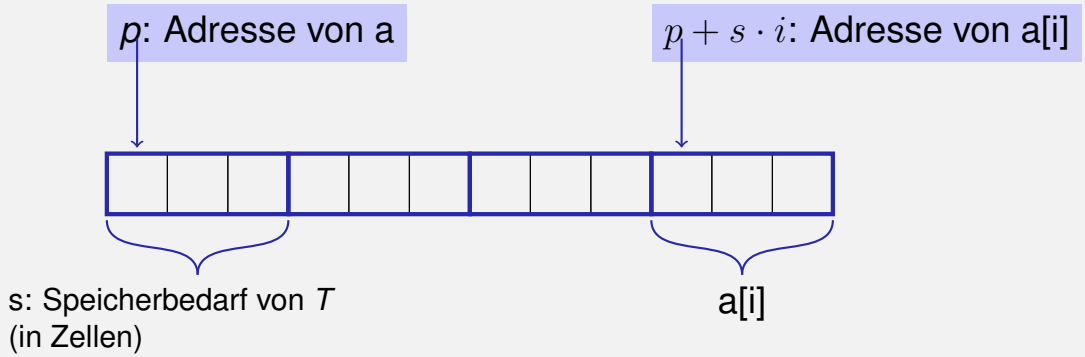
p: Adresse von a, d.h. Adresse der ersten Speicherzelle



s: Speicherbedarf von *T*
(in Zellen)

Wahlfreier Zugriff (Random Access)

- Wahlfreier Zugriff ist sehr effizient:



Feld-Initialisierung

- `int a[5];`

  Die 5 Elemente von `a` bleiben uninitialisiert (können später Werte zugewiesen bekommen)

Feld-Initialisierung

- `int a[5];`

  Die 5 Elemente von `a` bleiben uninitialisiert (können später Werte zugewiesen bekommen)

- `int a[5] = {4, 3, 5, 2, 1};`

  Die 5 Elemente von `a` werden mit einer *Initialisierungsliste* initialisiert.

- `int a[] = {4, 3, 5, 2, 1};`

Feld-Initialisierung

- `int a[5];`

  Die 5 Elemente von `a` bleiben uninitialisiert (können später Werte zugewiesen bekommen)

- `int a[5] = {4, 3, 5, 2, 1};`

  Die 5 Elemente von `a` werden mit einer *Initialisierungsliste* initialisiert.

- `int a[] = {4, 3, 5, 2, 1};`

Auch ok: Länge wird vom Compiler deduziert

Felder sind primitiv

- Der Zugriff auf Elemente ausserhalb der gültigen Grenzen eines Feldes führt zu undefiniertem Verhalten.

```
int arr[10];
for (int i=0; i<=10; ++i)
  arr[i] = 30;
```

Felder sind primitiv

- Der Zugriff auf Elemente ausserhalb der gültigen Grenzen eines Feldes führt zu undefiniertem Verhalten.

```
int arr[10];
for (int i=0; i<=10; ++i)
  arr[i] = 30; // Laufzeit-Fehler: Zugriff auf arr[10]!
```

Felder sind primitiv

Prüfung der Feldgrenzen

In Abwesenheit spezieller Compiler- oder Laufzeitunterstützung ist es die alleinige *Verantwortung des Programmierers*, die Gültigkeit aller Elementzugriffe zu prüfen.

Felder sind primitiv (II)

- Man kann Felder nicht wie bei anderen Typen initialisieren und zuweisen:

```
int a[5] = {4,3,5,2,1};
int b[5];
```

Felder sind primitiv (II)

- Man kann Felder nicht wie bei anderen Typen initialisieren und zuweisen:

```
int a[5] = {4,3,5,2,1};
int b[5];
b = a;            // Fehlermeldung des Compilers!
```

Felder sind primitiv (II)

- Man kann Felder nicht wie bei anderen Typen initialisieren und zuweisen:

```
int a[5] = {4,3,5,2,1};
int b[5];
b = a;          // Fehlermeldung des Compilers!
int c[5] = a;   // Fehlermeldung des Compilers!
```

Felder sind primitiv (II)

- Man kann Felder nicht wie bei anderen Typen initialisieren und zuweisen:

```
int a[5] = {4,3,5,2,1};
int b[5];
b = a;           // Fehlermeldung des Compilers!
int c[5] = a;    // Fehlermeldung des Compilers!
```

Warum?

Felder sind primitiv

- Felder sind „Erblast” der Sprache C und aus heutiger Sicht primitiv.

Felder sind primitiv

- Felder sind „Erblast” der Sprache C und aus heutiger Sicht primitiv.
- In C sind Felder sehr maschinennah und effizient, bieten aber keinen „Luxus” wie eingebautes Initialisieren und Kopieren.

Vektoren

- Offensichtlicher Nachteil statischer Felder: *konstante Feldlänge*

```
const unsigned int n = 1000;
bool crossed_out[n];
```

Vektoren

- Offensichtlicher Nachteil statischer Felder: *konstante Feldlänge*

```
const unsigned int n = 1000;
bool crossed_out[n];
```

- Abhilfe: Verwendung des Typs `Vector` aus der Standardbibliothek

```
#include <vector>
...
std::vector<bool> crossed_out (n, false);
```

Initialisierung mit n Elementen
Initialwert `false`.

Elementtyp, in spitzen Klammern

Eratosthenes mit Vektoren: Initialisierung

```
...
#include <vector>
...
std::vector<bool> crossed_out (n, false);
```

Eratosthenes mit Vektoren: Berechnung

```
for (unsigned int i = 2; i < n; ++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 < n; m += i)
      crossed_out[m] = true;
  }
```

Eratosthenes mit Vektoren: Berechnung

```
for (unsigned int i = 2; i < n; ++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 < n; m += i)
      crossed_out[m] = true;
  }
```

Gleicher Code wie mit “normalen” Feldern!

Zeichen und Texte

- Texte haben wir schon gesehen:

```
std::cout << "Prime numbers in {2,...,999}:\n";
```

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:	Feld mit zugrundeliegendem Typ **`char`**

Der Typ `char` (“character”)

- repräsentiert druckbare Zeichen (z.B. `'a'`) und *Steuerzeichen* (z.B. `'\n'`)

Der Typ `char` (“character”)

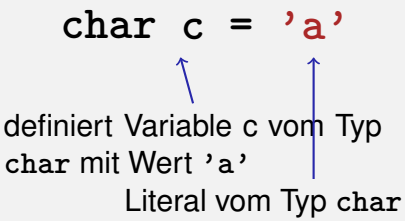
- repräsentiert druckbare Zeichen (z.B. `'a'`) und *Steuerzeichen* (z.B. `'\n'`)

```
char c = 'a'
```

definiert Variable c vom Typ `char` mit Wert `'a'`

Der Typ `char` (“character”)

- repräsentiert druckbare Zeichen (z.B. `'a'`) und *Steuerzeichen* (z.B. `'\n'`)



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

Der Typ `char` (“character”)

ist formal ein ganzzahliger Typ

- Werte konvertierbar nach `int` / `unsigned int`
- Werte belegen meistens 8 Bit

Der Typ `char` (“character”)

ist formal ein ganzzahliger Typ

- Werte konvertierbar nach `int` / `unsigned int`
- Werte belegen meistens 8 Bit

Wertebereich:
$\{-128, \ldots, 127\}$ oder $\{0, \ldots, 255\}$

Der ASCII-Code

- definiert konkrete Konversionsregeln
  `char` $\longrightarrow$ `int / unsigned int`

Der ASCII-Code

- definiert konkrete Konversionsregeln
  `char` $\longrightarrow$ `int / unsigned int`
- wird von fast allen Plattformen benutzt

Zeichen $\longrightarrow \{0, \ldots, 127\}$

`'A', 'B', ... , 'Z'` $\longrightarrow 65, 66, ..., 90$

`'a', 'b', ... , 'z'` $\longrightarrow 97, 98, ..., 122$

`'0', '1', ... , '9'` $\longrightarrow 48, 49, ..., 57$

- `for (char c = 'a'; c <= 'z'; ++c)`
  `std::cout << c;` abcdefghijklmnopqrstuvwxyz

Der ASCII-Code

- definiert konkrete Konversionsregeln
  `char` $\longrightarrow$ `int / unsigned int`
- wird von fast allen Plattformen benutzt

Zeichen $\longrightarrow \{0, \ldots, 127\}$

`'A', 'B', ... , 'Z'` $\longrightarrow 65, 66, ..., 90$

`'a', 'b', ... , 'z'` $\longrightarrow 97, 98, ..., 122$

`'0', '1', ... , '9'` $\longrightarrow 48, 49, ..., 57$

```
for (char c = 'a'; c <= 'z'; ++c)
    std::cout << c;
```

```
abcdefghijklmnopqrstuvwxyz
```

Erweiterung von ASCII: UTF-8

- Internationalisierung von Software $\Rightarrow$ grosse Zeichensätze nötig.
  Heute üblich: Unicode, 100 Schriftsysteme, 110000 Zeichen.

Erweiterung von ASCII: UTF-8

- Internationalisierung von Software $\Rightarrow$ grosse Zeichensätze nötig. Heute üblich: Unicode, 100 Schriftsysteme, 110000 Zeichen.
- ASCII kann mit 7 Bits codiert werden. Ein achtes Bit ist verwendbar

Erweiterung von ASCII: UTF-8

- Internationalisierung von Software $\Rightarrow$ grosse Zeichensätze nötig. Heute üblich: Unicode, 100 Schriftsysteme, 110000 Zeichen.
- ASCII kann mit 7 Bits codiert werden. Ein achtes Bit ist verwendbar, um weitere 128 Zeichen zu codieren – das ist aber Geschichte

Erweiterung von ASCII: UTF-8

- Internationalisierung von Software $\Rightarrow$ grosse Zeichensätze nötig. Heute üblich: Unicode, 100 Schriftsysteme, 110000 Zeichen.
- ASCII kann mit 7 Bits codiert werden. Ein achtes Bit ist verwendbar, um das Vorkommen weiterer Bits festzulegen.

Bits	Encoding
7	`0xxxxxxx`
11	`110xxxxx 10xxxxxx`
16	`1110xxxx 10xxxxxx 10xxxxxx`
21	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`
26	`111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`
31	`1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`

Erweiterung von ASCII: UTF-8

- Internationalisierung von Software ⇒ grosse Zeichensätze nötig. Heute üblich: Unicode, 100 Schriftsysteme, 110000 Zeichen.
- ASCII kann mit 7 Bits codiert werden. Ein achtes Bit ist verwendbar, um das Vorkommen weiterer Bits festzulegen.

Bits	Encoding
7	`0xxxxxxx`
11	`110xxxxx 10xxxxxx`
16	`1110xxxx 10xxxxxx 10xxxxxx`
21	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`
26	`111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`
31	`1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`

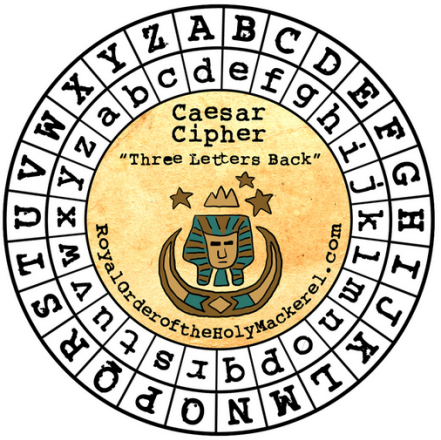
Einige Zeichen in UTF-8

Symbol	Codierung (jeweils 16 Bit)
	11100010 10011000 10100000
	11100010 10011000 10000011
	11100010 10001101 10101000
	11100010 10011000 10011001
	11100011 10000000 10100000
	11101111 10101111 10111001

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)



Caesar-Code: Hauptprogramm

```
// Program: caesar_encrypt.cpp
// encrypts a text by applying a cyclic shift of −3

#include<iostream>
#include<cassert>
#include<ios> // for std::noskipws

// PRE: s < 95 && s > −95
// POST: if c is one of the 95 printable ASCII characters, c is
//       cyclically shifted s printable characters to the right
void shift (char& c, int s);
```

Caesar-Code: Hauptprogramm

```
// Program: caesar_encrypt.cpp
// encrypts a text by applying a cyclic shift of −3

#include<iostream>
#include<cassert>
#include<ios> // for std::noskipws

// PRE: s < 95 && s > −95
// POST: if c is one of the 95 printable ASCII characters, c is
//       cyclically shifted s printable characters to the right
void shift (char& c, int s);
```

Leerzeichen und Zeilenumbrüche sollen *nicht* ignoriert werden

Caesar-Code: Hauptprogramm

```
int main ()
{
  std::cin >> std::noskipws; // don't skip whitespaces!

  // encryption loop
  char next;
  while (std::cin >> next) {
    shift (next, -3);
    std::cout << next;
  }
  return 0;
}
```

Konversion nach `bool`: liefert *false* genau dann, wenn die Eingabe leer ist.

Caesar-Code: Hauptprogramm

```
int main ()
{
  std::cin >> std::noskipws; // don't skip whitespaces!

  // encryption loop
  char next;
  while (std::cin >> next) {
    shift (next, -3);
    std::cout << next;
  }
  return 0;
}
```

Verschiebt nur druckbare Zeichen.

Caesar-Code: shift-Funktion

```
// PRE: s < 95 && s > -95
// POST: if c is one of the 95 printable ASCII characters, c is
//       cyclically shifted s printable characters to the right
void shift (char& c, int s)
{
  assert (s < 95 && s > -95);
  if (c >= 32 && c <= 126) {
    if (c + s > 126)
      c += (s - 95);
    else if (c + s < 32)
      c += (s + 95);
    else
      c += s;
  }
}
```

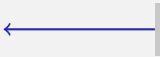
Caesar-Code: shift-Funktion

```
// PRE: s < 95 && s > -95
// POST: if c is one of the 95 printable ASCII characters, c is
//       cyclically shifted s printable characters to the right
void shift (char& c, int s)
{
  assert (s < 95 && s > -95);
  if (c >= 32 && c <= 126) {
    if (c + s > 126)
      c += (s - 95);
    else if (c + s < 32)
      c += (s + 95);
    else
      c += s;
  }
}
```

Call by reference!

Caesar-Code: shift-Funktion

```
// PRE: s < 95 && s > -95
// POST: if c is one of the 95 printable ASCII characters, c is
//       cyclically shifted s printable characters to the right
void shift (char& c, int s)
{
  assert (s < 95 && s > -95);
  if (c >= 32 && c <= 126) {
    if (c + s > 126)
      c += (s - 95);
    else if (c + s < 32)
      c += (s + 95);
    else
      c += s;
  }
}
```



Caesar-Code: shift-Funktion

```
// PRE: s < 95 && s > -95
// POST: if c is one of the 95 printable ASCII characters, c is
//       cyclically shifted s printable characters to the right
void shift (char& c, int s)
{
  assert (s < 95 && s > -95);
  if (c >= 32 && c <= 126) {
    if (c + s > 126)
      c += (s - 95);
    else if (c + s < 32)
      c += (s + 95);
    else
      c += s;
  }
}
```

Caesar-Code: shift-Funktion

```
// PRE: s < 95 && s > -95
// POST: if c is one of the 95 printable ASCII characters, c is
//       cyclically shifted s printable characters to the right
void shift (char& c, int s)
{
  assert (s < 95 && s > -95);
  if (c >= 32 && c <= 126) {
    if (c + s > 126)
      c += (s - 95);
    else if (c + s < 32)
      c += (s + 95);
    else
      c += s;
  }
}
```

Unterlauf – 95 vorwärts!

./caesar_encrypt < power8.cpp

```
„|Moldo^j7|mltbo5+‘mm
„|O^fpb|^|krj_bo|ql|qeb|bfdeqe|mltbo+

 fk‘irab|9flpqob^j;|

fkq|j^fk%&
x
||„|fkmrq
||pqa77‘lrq|99|~@ljmrqb|^[5|clo|^|:<|~8||
||fkq|^8
||pqa77‘fk|;;|^8

||„|‘ljmrq^qflk
||fkq|_|:|^|’|^8|„|_|:|^[/
||_|:|_|’|_8|||||„|_|:|^[1

||„|lrqmrq|_|’|_)|f+b+)|^[5
||pqa77‘lrq|99|^|99|~[5|:|~|99|_|’|_|99|~+Yk~8
||obqrok|-8
z
```

./caesar_encrypt < power8.cpp

```
„|Moldo^j7|mltbo5+‘mm
„|O^fpb|^|krj_bo|ql|qeb|bfdeqe|mltbo+

 fk‘irab|9flpqob^j;|

fkq|j^fk%&
x
||„|fkmrq
||pqa77‘lrq|99|~@ljmrqb|^[5|clo|^|:<|~8||
||fkq|^8
||pqa77‘fk|;;|^8

||„|‘ljmrq^qflk
||fkq|_|:|^|’|^8|„|_|:|^[/
||_|:|_|’|_8|||||„|_|:|^[1

||„|lrqmrq|_|’|_)|f+b+)|^[5
||pqa77‘lrq|99|^|99|~[5|:|~|99|_|’|_|99|~+Yk~8
||obqrok|-8
z
```

Program = Moldo^j

Caesar-Code: Entschlüsselung

```
// decryption loop
char next;
while (std::cin >> next) {
  shift (next, 3);
  std::cout << next;
}
```

Caesar-Code: Entschlüsselung

```
// decryption loop
char next;
while (std::cin >> next) {
  shift (next, 3);
  std::cout << next;
}
```

Jetzt: Verschiebung um 3 nach *rechts*

Caesar-Code: Entschlüsselung

```
// decryption loop
char next;
while (std::cin >> next) {
  shift (next, 3);
  std::cout << next;
}
```

Jetzt: Verschiebung um 3 nach *rechts*

Interessante Art, `power8.cpp` auszugeben:

- `./caesar_encrypt < power8.cpp | ./caeser_decrypt`

12. Felder (Arrays) II

Strings, Lindenmayer-Systeme, Mehrdimensionale Felder, Vektoren von Vektoren, Kürzeste Wege, Felder und Vektoren als Funktionsargumente

Strings als Felder

- sind repräsentierbar mit zugrundeliegendem Typ `char`

```
char text[] = {'b','o','o','l'}
```

Strings als Felder

- sind repräsentierbar mit zugrundeliegendem Typ `char`

```
char text[] = {'b','o','o','l'}
```

definiert ein Feld der Länge 4, das dem Text “bool” entspricht.

Strings als Felder

- sind repräsentierbar mit zugrundeliegendem Typ `char`

```
char text[] = {'b','o','o','l'}
```

- können auch durch String-Literale definiert werden

```
char text[] = "bool"
```

Strings als Felder

- sind repräsentierbar mit zugrundeliegendem Typ `char`

```
char text[] = {'b','o','o','l'}
```

- können auch durch String-Literale definiert werden

```
char text[] = "bool"
```

definiert ein Feld der Länge *5*, das dem Text “bool” entspricht und *null-terminiert* ist. Extrazeichen `'0'` wird am Ende angehängt – Der Text „speichert seine Länge”

Strings als Felder

- sind repräsentierbar mit zugrundeliegendem Typ `char`

```
char text[] = {'b','o','o','l'}
```

- können auch durch String-Literale definiert werden

```
char text[] = "bool"
```

- können nur mit konstanter Grösse definiert werden

Texte

- können mit dem Typ `std::string` aus der Standardbibliothek repräsentiert werden.

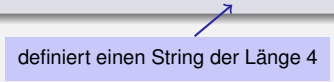
```
std::string text = "bool";
```

definiert einen String der Länge 4

Texte

- können mit dem Typ `std::string` aus der Standardbibliothek repräsentiert werden.

```
std::string text = "bool";
```



Strings: gepimpte `char`-Felder

Ein `std::string`...

- kennt seine Länge

```
text.length()
```

Strings: gepimpte `char`-Felder

Ein `std::string`...

- kann mit variabler Länge initialisiert werden

```
std::string text (n, 'a')
```

Strings: gepimpte `char`-Felder

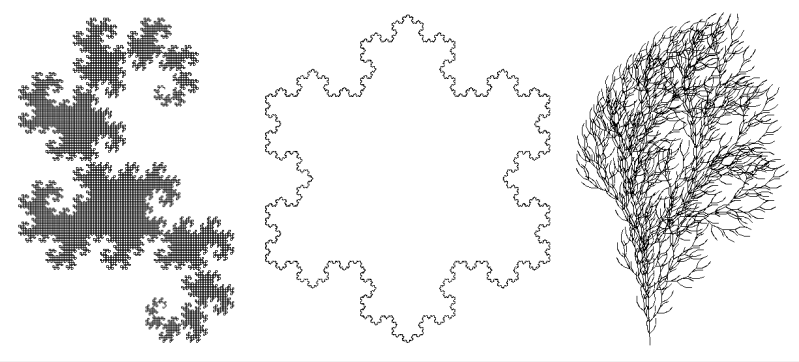
Ein `std::string`...

- „versteht” Vergleiche

```
if (text1 == text2) ...
```

Lindenmayer-Systeme (L-Systeme)

Fraktale aus Strings und Schildkröten



Definition und Beispiel

- Alphabet Σ
- $\{\mathrm{F}, +, -\}$

Definition und Beispiel

- Alphabet Σ
- Σ^*: alle endlichen Wörter über Σ

- $\{\mathrm{F}, +, -\}$

Definition und Beispiel

- Alphabet Σ
- Σ^*: alle endlichen Wörter über Σ
- Produktion $P : \Sigma \to \Sigma^*$

- $\{\mathrm{F}, +, -\}$
-

c	$P(c)$
F	F + F +
+	+
−	−

Definition und Beispiel

- Alphabet Σ
- Σ^*: alle endlichen Wörter über Σ
- Produktion $P : \Sigma \to \Sigma^*$
- Startwort $s_0 \in \Sigma^*$

- $\{\mathrm{F}, +, -\}$
-

c	$P(c)$
F	F + F +
+	+
−	−

- F

Definition und Beispiel

- Alphabet Σ
- Σ^*: alle endlichen Wörter über Σ
- Produktion $P : \Sigma \to \Sigma^*$
- Startwort $s_0 \in \Sigma^*$

- $\{\mathrm{F}, +, -\}$
-

c	$P(c)$
F	F + F +
+	+
−	−

- F

Definition

Das Tripel $\mathcal{L} = (\Sigma, P, s_0)$ ist ein L-System.

Die beschriebene Sprache

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$:

$$w_0 := s_0$$

$$P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$$

$$w_0 := \mathrm{F}$$

Die beschriebene Sprache

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$: $P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$w_0 := s_0$ $\qquad$ $w_0 := \mathrm{F}$

$w_1 := P(w_0)$ $\qquad$ $w_1 := \mathrm{F} + \mathrm{F} +$

Die beschriebene Sprache

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$: $P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$$w_0 := s_0 \qquad w_0 := \mathrm{F}$$

$$w_1 := P(w_0) \qquad w_1 := \mathrm{F} + \mathrm{F} +$$

$$w_2 := P(w_1) \qquad w_2 := \mathrm{F} + \mathrm{F} + + \mathrm{F} + \mathrm{F} + +$$

Definition

$P(c_1 c_2 \ldots c_n) := P(c_1) P(c_2) \ldots P(c_n)$

Die beschriebene Sprache

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$:

$P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$w_0 := s_0$ $\qquad$ $w_0 := \mathrm{F}$

$w_1 := P(w_0)$ $\qquad$ $w_1 := \mathrm{F} + \mathrm{F} +$

$w_2 := P(w_1)$ $\qquad$ $w_2 := \mathrm{F} + \mathrm{F} + + \mathrm{F} + \mathrm{F} + +$

$P(\mathrm{F})\ P(+)\ P(\mathrm{F})\ P(+)$

Definition

$P(c_1 c_2 \ldots c_n) := P(c_1) P(c_2) \ldots P(c_n)$

Die beschriebene Sprache

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$: $P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$$w_0 := s_0 \qquad w_0 := \mathrm{F}$$

$$w_1 := P(w_0) \qquad w_1 := \mathrm{F} + \mathrm{F} +$$

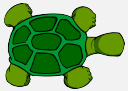
$$w_2 := P(w_1) \qquad w_2 := \mathrm{F} + \mathrm{F} + + \mathrm{F} + \mathrm{F} + +$$

Definition

$P(c_1 c_2 \ldots c_n) := P(c_1) P(c_2) \ldots P(c_n)$

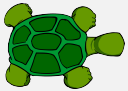
Turtle-Grafik

Schildkröte mit Position und Richtung



Turtle-Grafik

Schildkröte mit Position und Richtung

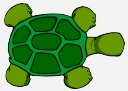


Schildkröte versteht 3 Befehle:

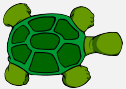
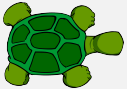
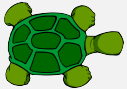
F: Gehe einen Schritt vorwärts	$+$: Drehe dich um 90 Grad	$-$: Drehe dich um -90 Grad

Turtle-Grafik

Schildkröte mit Position und Richtung

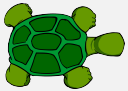


Schildkröte versteht 3 Befehle:

F: Gehe einen Schritt vorwärts 	$+$: Drehe dich um 90 Grad 	$-$: Drehe dich um -90 Grad 

Turtle-Grafik

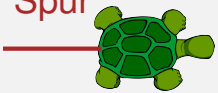
Schildkröte mit Position und Richtung



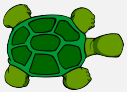
Schildkröte versteht 3 Befehle:

F: Gehe einen Schritt vorwärts ✓

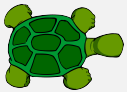
Spur



$+$: Drehe dich um 90 Grad

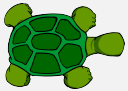


$-$: Drehe dich um -90 Grad



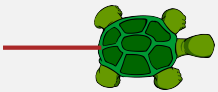
Turtle-Grafik

Schildkröte mit Position und Richtung



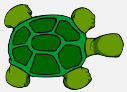
Schildkröte versteht 3 Befehle:

F: Gehe einen Schritt vorwärts ✓



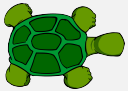
$+$: Drehe dich um 90 Grad ✓

$-$: Drehe dich um -90 Grad



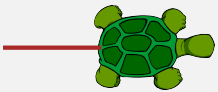
Turtle-Grafik

Schildkröte mit Position und Richtung



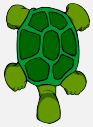
Schildkröte versteht 3 Befehle:

F: Gehe einen Schritt vorwärts ✓



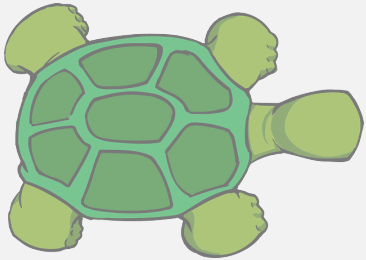
$+$: Drehe dich um 90 Grad ✓

$-$: Drehe dich um -90 Grad ✓



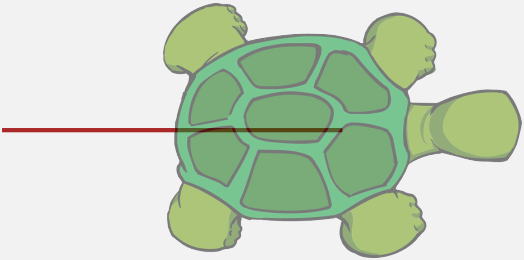
Wörter zeichnen!

$w_1 = \mathrm{F} + \mathrm{F} +$



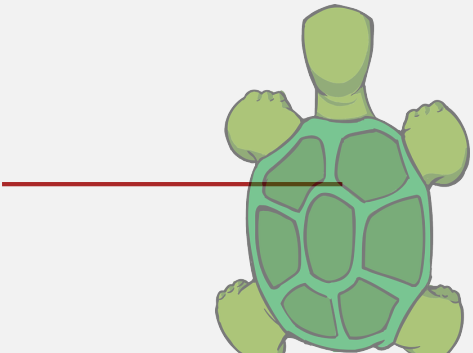
Wörter zeichnen!

$w_1 = \mathrm{F} + \mathrm{F} +$



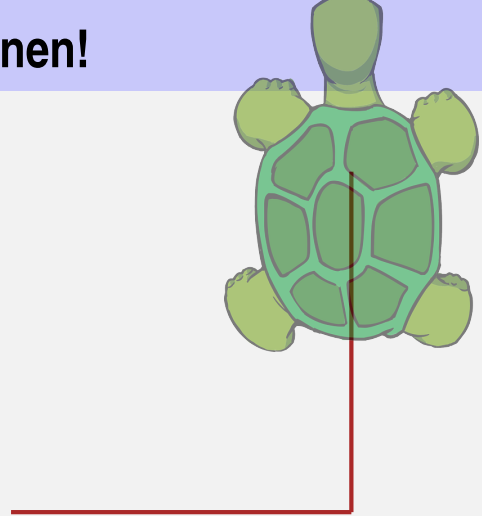
Wörter zeichnen!

$w_1 = \mathrm{F} + \mathrm{F} +$

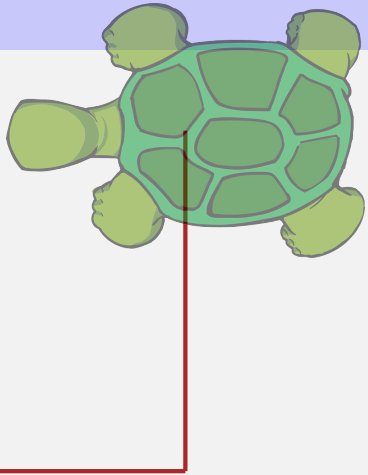


Wörter zeichnen!

$w_1 = \mathrm{F} + \mathrm{F} +$



Wörter zeichnen!



$w_1 = \mathrm{F} + \mathrm{F} +$

Wörter zeichnen!

$w_1 = \mathrm{F} + \mathrm{F} + \checkmark$

lindenmayer.cpp: Hauptprogramm

Wörter $w_0, w_1, w_2, \ldots w_n \in \Sigma^*$: `std::string`

```
...
#include "turtle.h"
...
std::cout << "Number of iterations =? ";
unsigned int n;
std::cin >> n;

std::string w = "F";

for (unsigned int i = 0; i < n; ++i)
  w = next_word (w);

draw_word (w);
```

lindenmayer.cpp: Hauptprogramm

Wörter $w_0, w_1, w_2, \ldots w_n \in \Sigma^*$: `std::string`

```
...
#include "turtle.h"
...
std::cout << "Number of iterations =? ";
unsigned int n;
std::cin >> n;

std::string w = "F";

for (unsigned int i = 0; i < n; ++i)
  w = next_word (w);

draw_word (w);
```

$w = w_0 = \mathrm{F}$

lindenmayer.cpp: Hauptprogramm

Wörter $w_0, w_1, w_2, \ldots w_n \in \Sigma^*$: `std::string`

```
...
#include "turtle.h"
...
std::cout << "Number of iterations =? ";
unsigned int n;
std::cin >> n;

std::string w = "F";

for (unsigned int i = 0; i < n; ++i)
  w = next_word (w);

draw_word (w);
```

$w = w_i \rightarrow w = w_{i+1}$

lindenmayer.cpp: Hauptprogramm

Wörter $w_0, w_1, w_2, \ldots w_n \in \Sigma^*$: `std::string`

```
...
#include "turtle.h"
...
std::cout << "Number of iterations =? ";
unsigned int n;
std::cin >> n;

std::string w = "F";

for (unsigned int i = 0; i < n; ++i)
  w = next_word (w);

draw_word (w);
```

Zeichne $w = w_n$!

lindenmayer.cpp: next_word

```
// POST: replaces all symbols in word according to their
//       production and returns the result
std::string next_word (std::string word) {
  std::string next;
  for (unsigned int k = 0; k < word.length(); ++k)
    next += production (word[k]);
  return next;
}
```

lindenmayer.cpp: next_word

```
// POST: replaces all symbols in word according to their
//       production and returns the result
std::string next_word (std::string word) {
  std::string next;
  for (unsigned int k = 0; k < word.length(); ++k)
    next += production (word[k]);
  return next;
}

// POST: returns the production of c
std::string production (char c) {
  switch (c) {
  case 'F': return "F+F+";
  default: return std::string (1, c); // trivial production c -> c
  }
}
```

lindenmayer.cpp: draw_word

```
// POST: draws the turtle graphic interpretation of word
void draw_word (std::string word)
{
  for (unsigned int k = 0; k < word.length(); ++k)
    switch (word[k]) {
    case 'F':
      turtle::forward();
     break;
    case '+':
      turtle::left(90);
      break;
    case '-':
      turtle::right(90);
    }
}
```

lindenmayer.cpp: draw_word

```
// POST: draws the turtle graphic interpretation of word
void draw_word (std::string word)
{
  for (unsigned int k = 0; k < word.length(); ++k)
    switch (word[k]) {
    case 'F':
      turtle::forward();
     break;
    case '+':
      turtle::left(90);
      break;
    case '-':
      turtle::right(90);
    }
}
```

Springe zum `case`, der `word[k]` entspricht.

lindenmayer.cpp: draw_word

```
// POST: draws the turtle graphic interpretation of word
void draw_word (std::string word)
{
  for (unsigned int k = 0; k < word.length(); ++k)
    switch (word[k]) {
    case 'F':
      turtle::forward();
     break;
    case '+':
      turtle::left(90);
      break;
    case '-':
      turtle::right(90);
    }
}
```

Vorwärts! (Funktion aus unserer Schildkröten-Bibliothek)

lindenmayer.cpp: draw_word

```
// POST: draws the turtle graphic interpretation of word
void draw_word (std::string word)
{
  for (unsigned int k = 0; k < word.length(); ++k)
    switch (word[k]) {
    case 'F':
      turtle::forward();
     break;
    case '+':
      turtle::left(90);
      break;
    case '-':
      turtle::right(90);
    }
}
```

Überspringe die folgenden cases

lindenmayer.cpp: draw_word

```
// POST: draws the turtle graphic interpretation of word
void draw_word (std::string word)
{
  for (unsigned int k = 0; k < word.length(); ++k)
    switch (word[k]) {
    case 'F':
      turtle::forward();
     break;
    case '+':
      turtle::left(90);
      break;
    case '-':
      turtle::right(90);
    }
}
```

Drehe dich um 90 Grad! (Funktion aus unserer Schildkröten-Bibliothek)

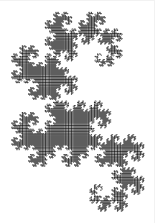
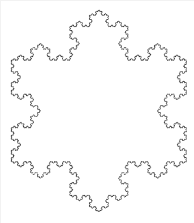
lindenmayer.cpp: draw_word

```
// POST: draws the turtle graphic interpretation of word
void draw_word (std::string word)
{
  for (unsigned int k = 0; k < word.length(); ++k)
    switch (word[k]) {
    case 'F':
      turtle::forward();
     break;
    case '+':
      turtle::left(90);
      break;
    case '-':
      turtle::right(90);
    }
}
```

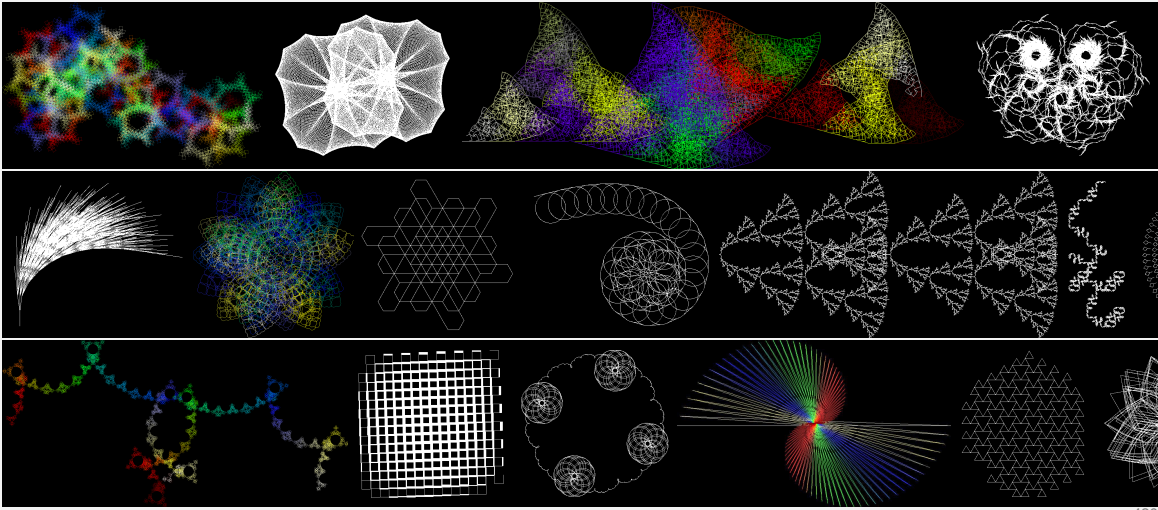
Drehe dich um -90 Grad! (Funktion aus unserer Schildkröten-Bibliothek)

L-Systeme: Erweiterungen

- Beliebige Symbole ohne grafische Interpretation (`dragon.cpp`)
- Beliebige Drehwinkel (`snowflake.cpp`)
- Sichern und Wiederherstellen des Schildkröten-Zustandes $\rightarrow$ Pflanzen (`bush.cpp`)


L-System-Challenge: `amazing.cpp`!



Mehrdimensionale Felder

- sind Felder von Feldern
- dienen zum Speichern von *Tabellen, Matrizen,...*

Mehrdimensionale Felder

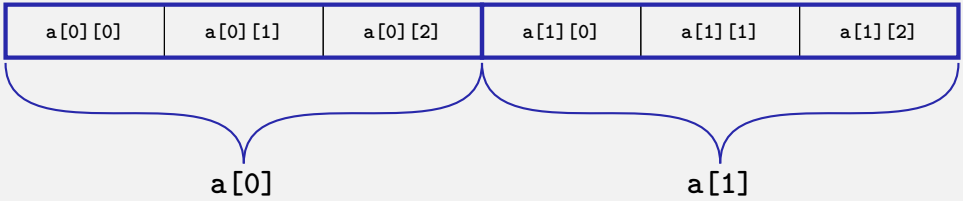
- sind Felder von Feldern

```
int a[2][3]
```


`a` hat zwei Elemente, und jedes von ihnen ist ein Feld der Länge 3 mit zugrundeliegendem Typ `int`

Mehrdimensionale Felder

Im Speicher: flach



Mehrdimensionale Felder

Im Speicher: flach

Im Kopf: Matrix

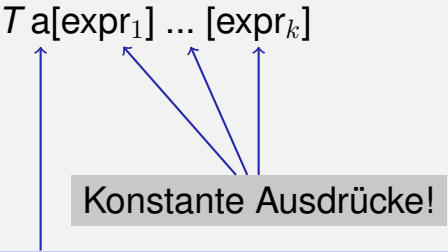
Spalten →

Zeilen ↓

	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]

Mehrdimensionale Felder

- sind Felder von Feldern von Feldern ....



a hat $expr_1$ Elemente und jedes von ihnen ist ein Feld mit $expr_2$ Elementen, von denen jedes ein Feld mit $expr_3$ Elementen ist, ...

Mehrdimensionale Felder

Initialisierung:

```
int a[2][3] =
    {
        {2,4,6},{1,3,5}
    }
```

2	4	6	1	3	5

Mehrdimensionale Felder

Initialisierung:

```
int a[][3] =
    {
        {2,4,6},{1,3,5}
    }
```

Erste Dimension kann weggelassen werden

Vektoren von Vektoren

- Wie bekommen wir mehrdimensionale Felder mit variablen Dimensionen?

Vektoren von Vektoren

- Wie bekommen wir mehrdimensionale Felder mit variablen Dimensionen?
- Lösung: Vektoren von Vektoren

Beispiel: Vektor der Länge n von Vektoren der Länge m:

```
std::vector<?> a (n, ?);
```

Zugrundeliegender Typ des ersten Vektors?

Vektoren von Vektoren

- Wie bekommen wir mehrdimensionale Felder mit variablen Dimensionen?
- Lösung: Vektoren von Vektoren

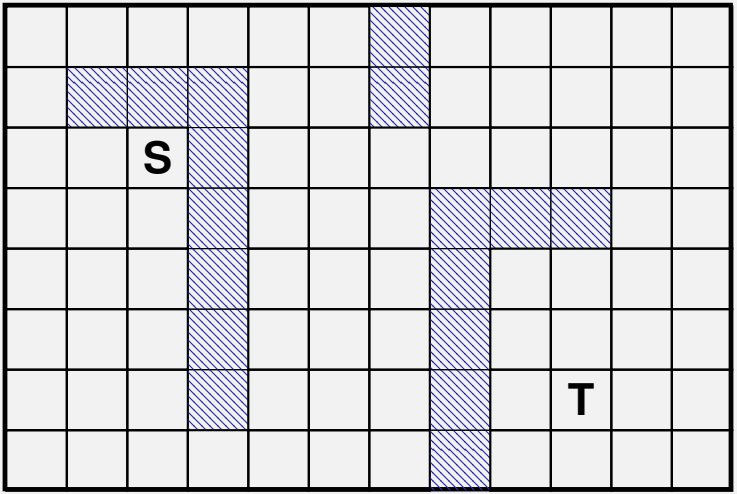
Beispiel: Vektor der Länge n von Vektoren der Länge m:

```
std::vector<std::vector<int> > a (n,
                                  std::vector<int>(m));
```

Zugrundeliegender Typ des ersten Vektors?
`std::vector<int>`, initialisiert mit Länge m:
`std::vector<int>(m)`

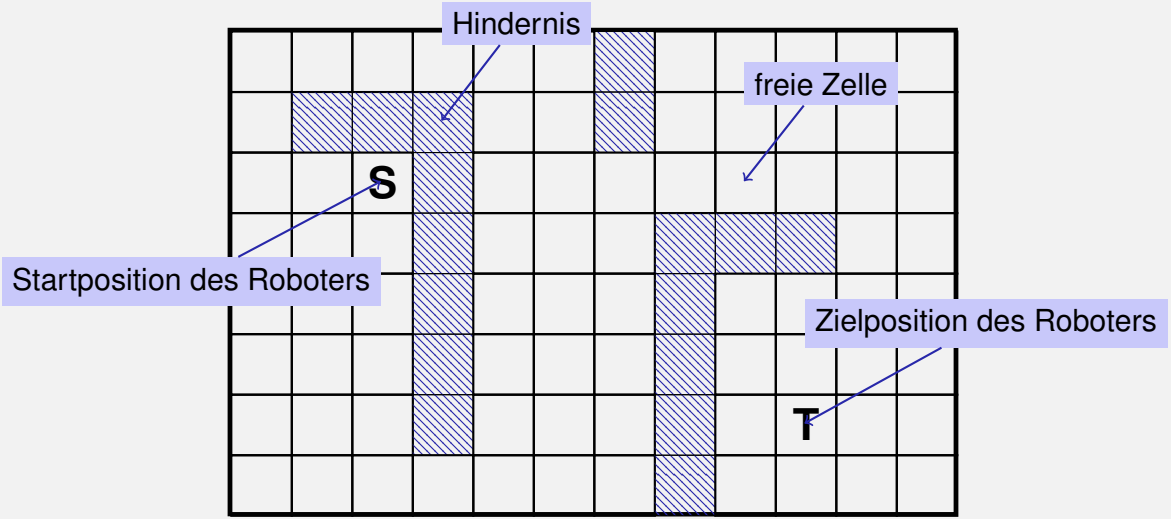
Anwendung: Kürzeste Wege

Fabrik-Halle ($n \times m$ quadratische Zellen)



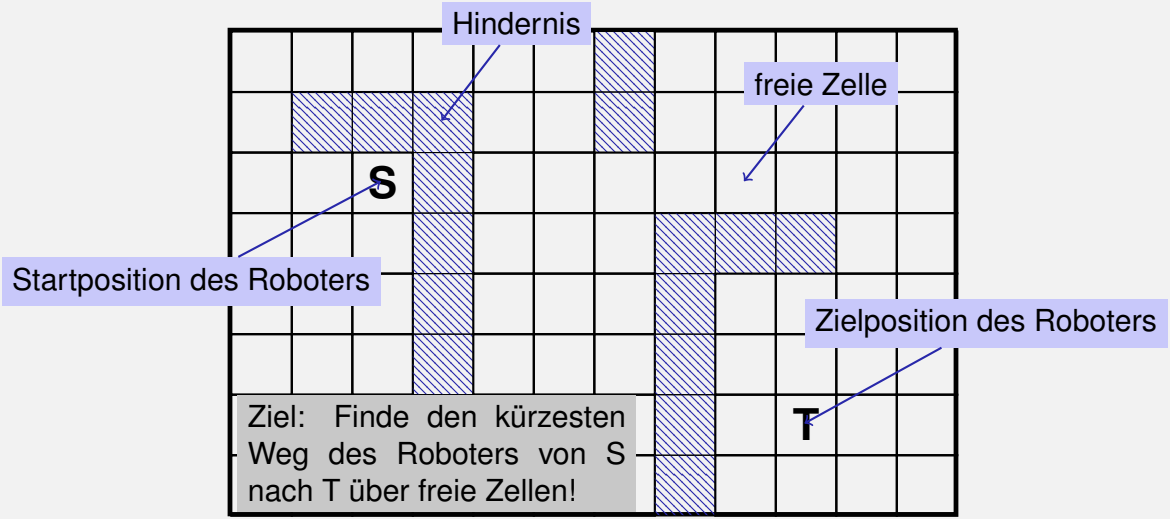
Anwendung: Kürzeste Wege

Fabrik-Halle ($n \times m$ quadratische Zellen)



Anwendung: Kürzeste Wege

Fabrik-Halle ($n \times m$ quadratische Zellen)



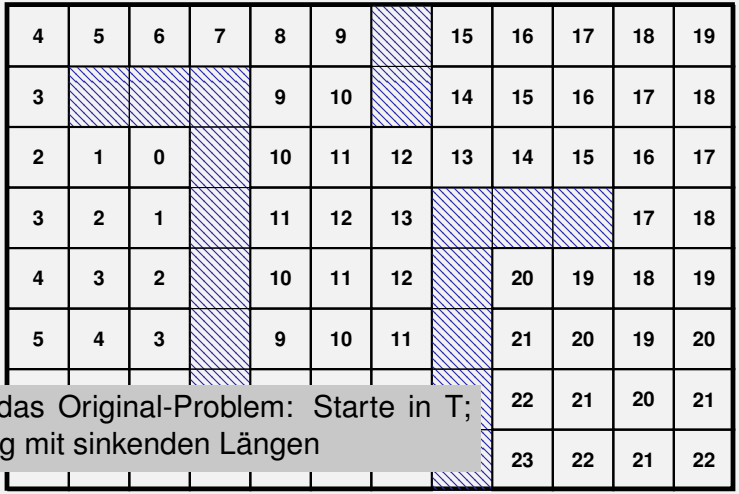
Ein (scheinbar) anderes Problem

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

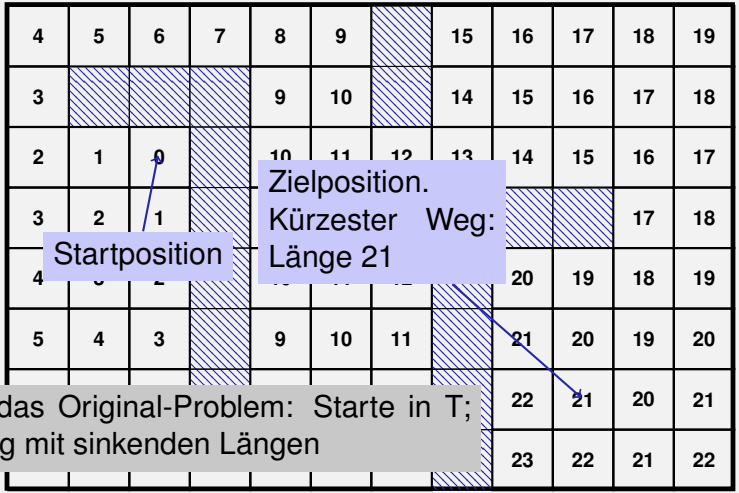
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

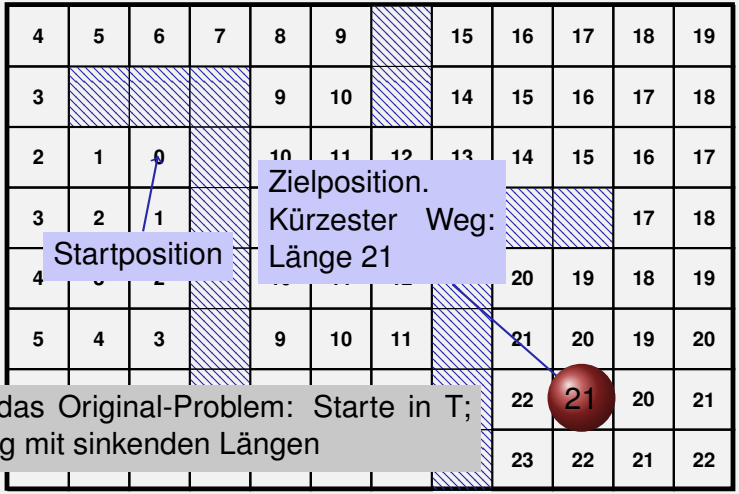
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

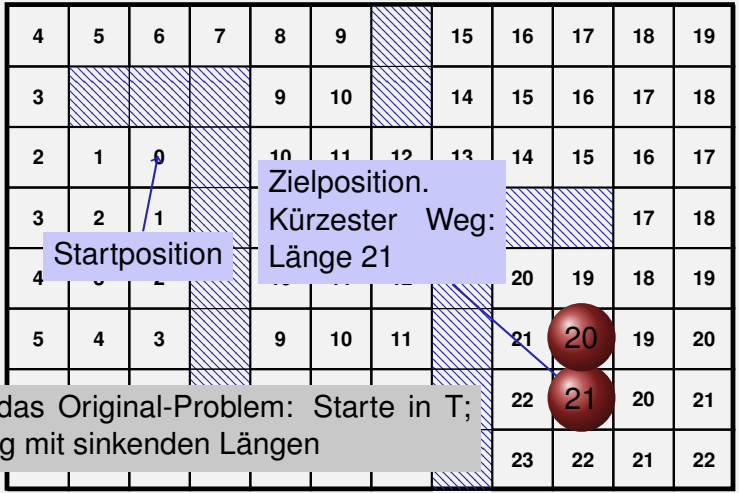
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

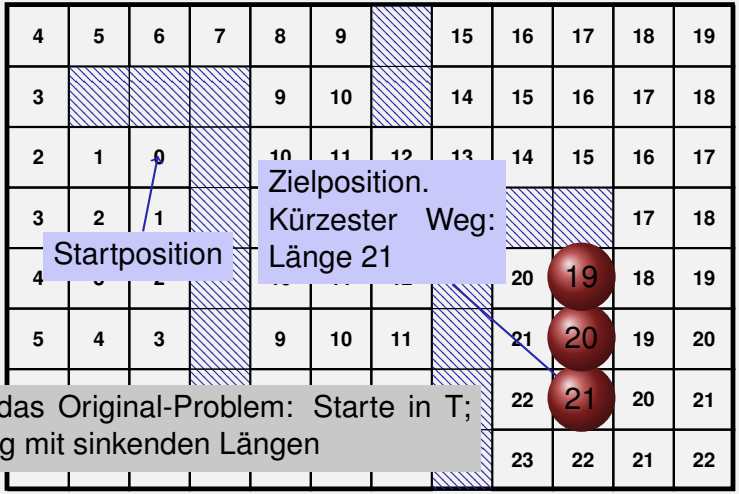
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

Ein (scheinbar) anderes Problem

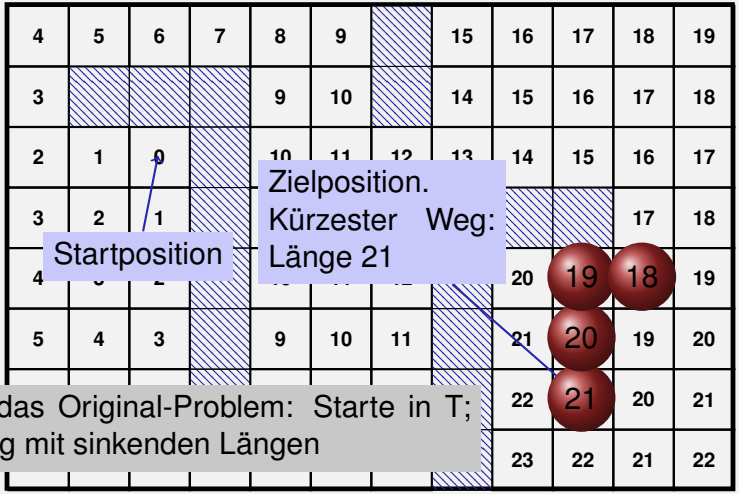
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

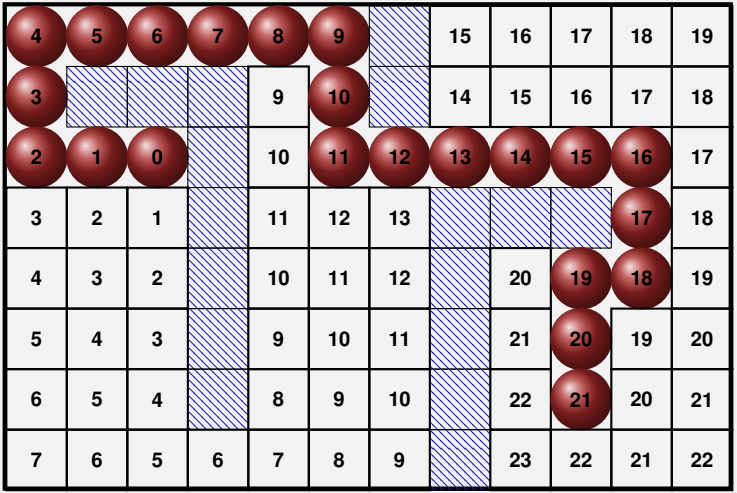
Ein (scheinbar) anderes Problem

Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen

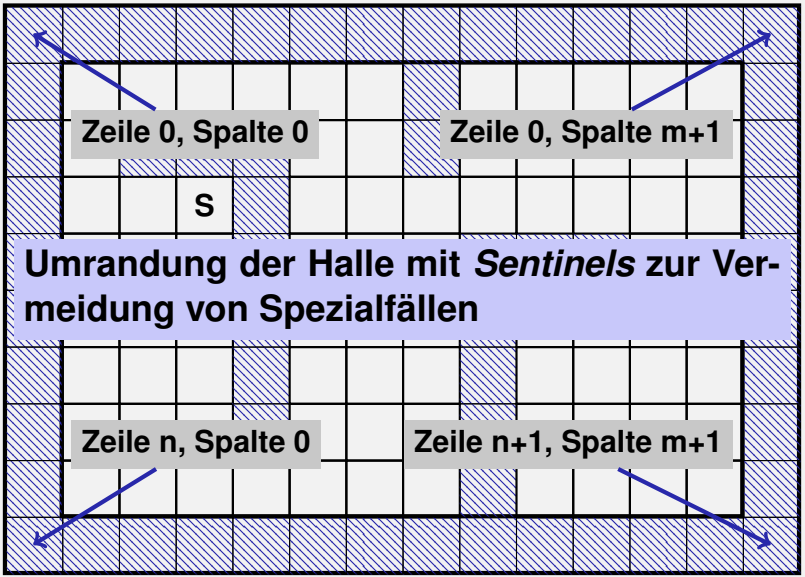


Ein (scheinbar) anderes Problem

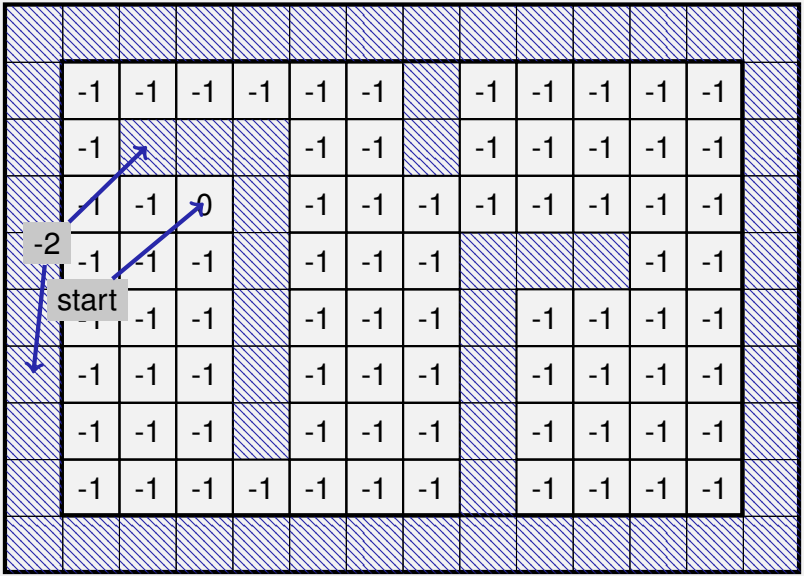
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



Vorbereitung: Wächter (*Sentinels*)



Vorbereitung: Initiale Markierung



Das Kürzeste-Wege-Programm

```
// define a two-dimensional array of dimensions
// (n+2) x (m+2) to hold the floor
// plus extra walls around
std::vector<std::vector<int> >
    floor (n+2, std::vector<int>(m+2));


// Einlesen der Hallenbelegung, initiale Markierung
// (Handout)
...
// Markierung der umschliessenden Waende (Handout)
...
```

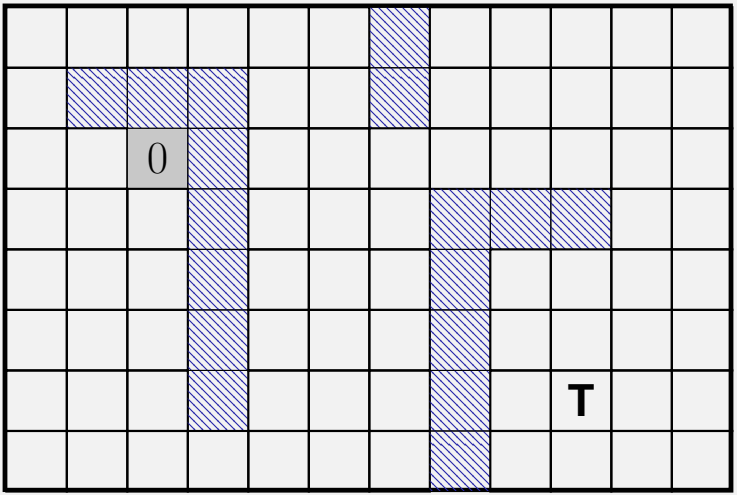
Das Kürzeste-Wege-Programm

```
// define a two-dimensional array of dimensions
// (n+2) x (m+2) to hold the floor
// plus extra walls around
std::vector<std::vector<int> >
    floor (n+2, std::vector<int>(m+2));
```


```
// Einlesen der Hallenbelegung, initiale Markierung
// (Handout)
...
// Markierung der umschliessenden Waende (Handout)
...
```

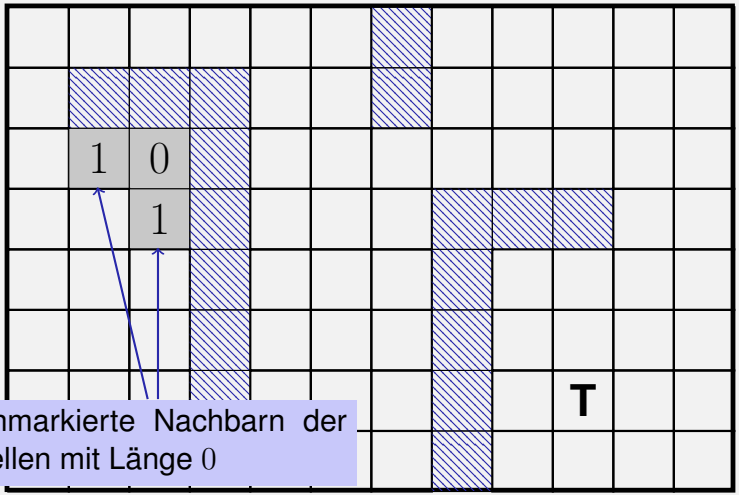
Markierung aller Zellen mit ihren Weglängen

Schritt 0: Alle Zellen mit Weglänge 0



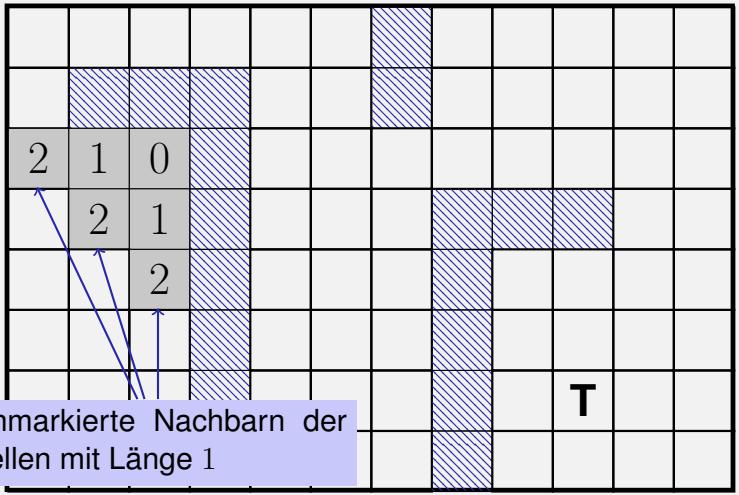
Markierung aller Zellen mit ihren Weglängen

Schritt 1: Alle Zellen mit Weglänge 1



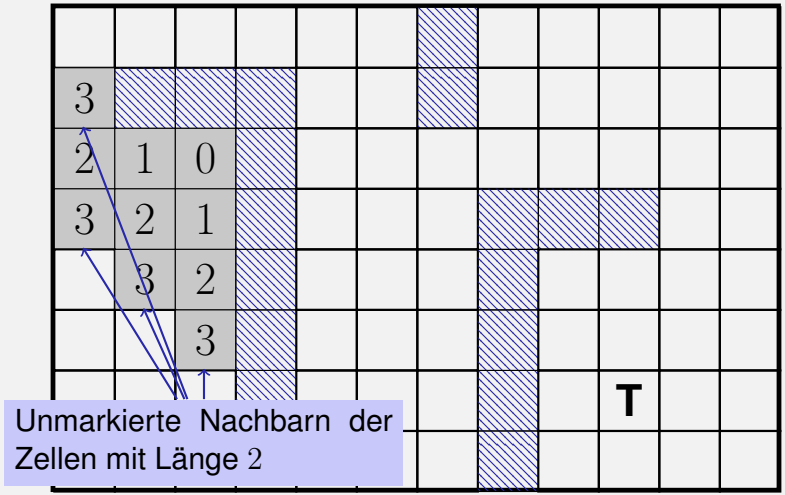
Markierung aller Zellen mit ihren Weglängen

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 $i = 1, 2, 3...$

```
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 $i = 1, 2, 3...$

```
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;
}
```

zeigt an, ob in einem Durchlauf durch alle Zellen Fortschritt gemacht wurde

Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3...$

```
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;
}
```

Gehe über alle Zellen

Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3...$

```
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 $i = 1, 2, 3...$

```
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 $i = 1, 2, 3...$

```
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*

Das Kürzeste-Wege-Programm

- Algorithmus: *Breitensuche*
- Das Programm kann recht langsam sein, weil für jedes i alle Zellen durchlaufen werden

Das Kürzeste-Wege-Programm

- Algorithmus: *Breitensuche*
- 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 $i - 1$

Felder als Funktionsargumente

Felder können auch als *Referenz*-Argumente an eine Funktion übergeben werden. (Hier const, weil nur Lesezugriff nötig).

```
void print_vector(const int (&v)[3]) {
  for (int i = 0; i<3 ; ++i) {
    std::cout << v[i] << " ";
  }
}
```

Felder als Funktionsargumente

Das geht auch für mehrdimensionale Felder.

```
void print_matrix(const int (&m)[3][3]) {
  for (int i = 0; i<3 ; ++i) {
    print_vector (m[i]);
    std::cout << "\n";
  }
}
```

Vektoren als Funktionsargumente

Vektoren können *per value*, aber auch als *Referenz*-Argumente an eine Funktion übergeben werden.

```
void print_vector(const std::vector<int>& v) {
  for (int i = 0; i<v.size() ; ++i) {
    std::cout << v[i] << " ";
  }
}
```

Vektoren als Funktionsargumente

Vektoren können *per value*, aber auch als *Referenz*-Argumente an eine Funktion übergeben werden.

```
void print_vector(const std::vector<int>& v) {
  for (int i = 0; i<v.size() ; ++i) {
    std::cout << v[i] << " ";
  }
}
```

Hier: *Call by Reference* ist effizienter, weil der Vektor sehr lang sein kann.

Vektoren als Funktionsargumente

Das geht auch für mehrdimensionale Vektoren.

```
void print_matrix(const std::vector<std::vector<int> >& m) {
  for (int i = 0; i<m.size() ; ++i) {
    print_vector (m[i]);
    std::cout << "\n";
  }
}
```

13. Zeiger, Algorithmen, Iteratoren und Container I

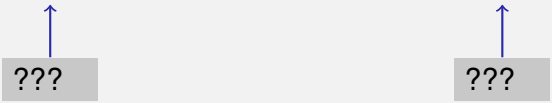
Zeiger, Address- und Dereferenoperator,
Feld-nach-Zeiger-Konversion

Komische Dinge...

```
#include<iostream>
#include<algorithm>

int main(){
  int a[] = {3, 2, 1, 5, 4, 6, 7};

  // gib das kleinste Element in a aus
  std::cout << *std::min_element (a, a + 7);

  return 0;
}
```

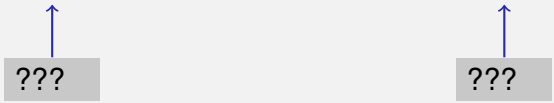
Komische Dinge...

```
#include<iostream>
#include<algorithm>

int main(){
  int a[] = {3, 2, 1, 5, 4, 6, 7};

  // gib das kleinste Element in a aus
  std::cout << *std::min_element (a, a + 7);

  return 0;
}
```



Komische Dinge...

```
#include<iostream>
#include<algorithm>

int main(){
  int a[] = {3, 2, 1, 5, 4, 6, 7};

  // gib das kleinste Element in a aus
  std::cout << *std::min_element (a, a + 7);

  return 0;
}
```



Dafür müssen wir zuerst *Zeiger* verstehen!

Referenzen: Wo ist Anakin?

```
int anakin_skywalker = 9;
```

Referenzen: Wo ist Anakin?


```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker;
```

Referenzen: Wo ist Anakin?

Zuweisung

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker;
darth_vader = 22;

// anakin_skywalker = 22
```

Referenzen: Wo ist Anakin?

“Suche nach Vader, und Anakin finden du wirst.”

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker;
darth_vader = 22;

// anakin_skywalker = 22
```

Zeiger: Wo ist Anakin?

```
int anakin_skywalker = 9;
```

Zeiger: Wo ist Anakin?

```
int anakin_skywalker = 9;
```

“Anakins Adresse ist `0x7fff6bdd1b54`.”

Zeiger: Wo ist Anakin?

Initialisierung

```
int anakin_skywalker = 9;
int* here = &anakin_skywalker;
```

“Anakins Adresse ist `0x7fff6bdd1b54`.”

Zeiger: Wo ist Anakin?

Ausgabe

```
int anakin_skywalker = 9;
int* here = &anakin_skywalker;
std::cout << here; // Adresse
```

“Anakins Adresse ist `0x7fff6bdd1b54`.”

Zeiger: Wo ist Anakin?

Zuweisung

```
int anakin_skywalker = 9;
int* here = &anakin_skywalker;
std::cout << here; // Adresse
*here =  22;

// anakin_skywalker = 22
```

“Anakins Adresse ist `0x7fff6bdd1b54`.”

Swap mit Zeigern

```
void swap(int* x, int* y){
  int t = *x;
  *x = *y;
  *y = t;
}

...
int a = 2;
int b = 1;
swap(&a, &b);
std::cout << "a= " << a << "\n"; // 1
std::cout << "b = " << b << "\n"; // 2
```

Swap mit Zeigern

```
void swap(int* x, int* y){
  int t = *x;
  *x = *y;
  *y = t;
}

...
int a = 2;
int b = 1;
swap(&a, &b);
std::cout << "a= " << a << "\n"; // 1
std::cout << "b = " << b << "\n"; // 2
```

Zeiger Typen

`T*` Zeiger-Typ zum zugrunde liegenden Typ `T`.

Ein Ausdruck vom Typ `T*` heisst *Zeiger* (auf `T`).

Zeiger Typen

Wert eines Zeigers auf `T` ist die Adresse eines Objektes vom Typ `T`.

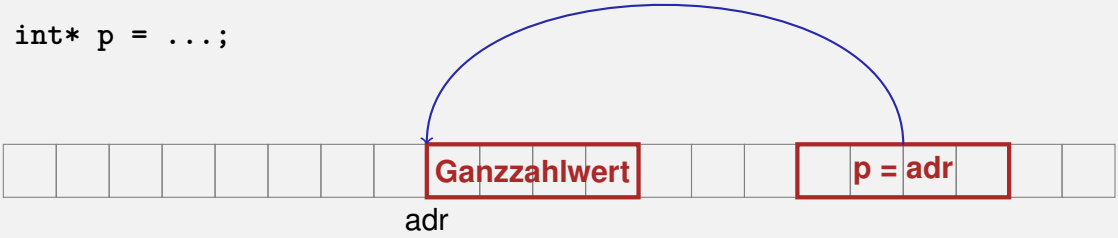
Beispiele

`int* p;` Variable `p` ist Zeiger auf ein `int`.
`float* q;` Variable `q` ist Zeiger auf ein `float`.

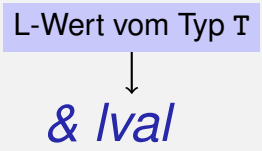
Zeiger Typen

Wert eines Zeigers auf `T` ist die Adresse eines Objektes vom Typ `T`.

```
int* p = ...;
```



Adress-Operator



Typ: *T**

Wert: Adresse von *lval*

Adress-Operator

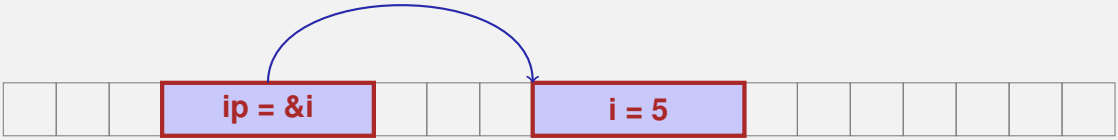
Beispiel

```
int i = 5;
```

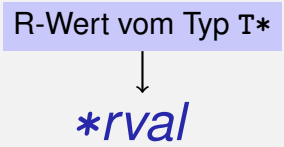
Adress-Operator

Beispiel

```
int i = 5;
int* ip = &i;  // ip initialisiert
               // mit Adresse von i.
```



Dereferenz-Operator



Typ: *T*

Wert: *Wert* des Objekts an Addresse *rval*

Dereferenz-Operator

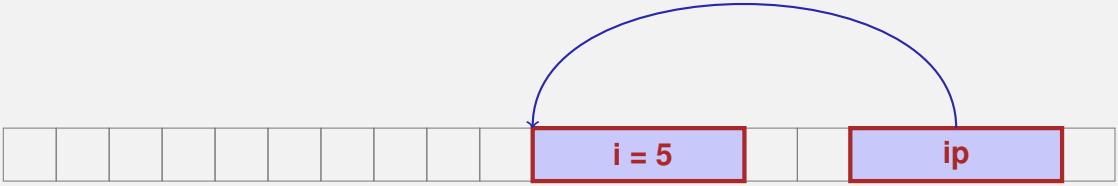
Beispiel

```
int i = 5;
int* ip = &i;  // ip initialisiert
               // mit Adresse von i.
int j = *ip;   // j == 5
```

Dereferenz-Operator

Beispiel

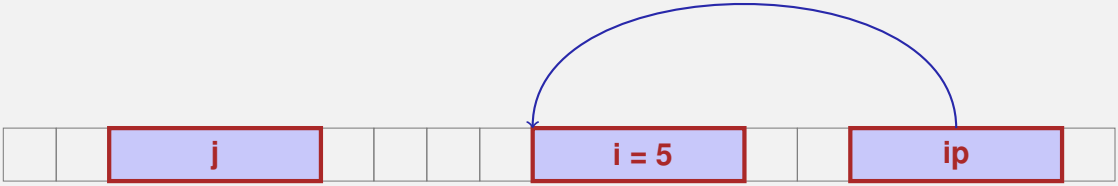
```
int i = 5;
int* ip = &i;  // ip initialisiert
               // mit Adresse von i.
int j = *ip;   // j == 5
```



Dereferenz-Operator

Beispiel

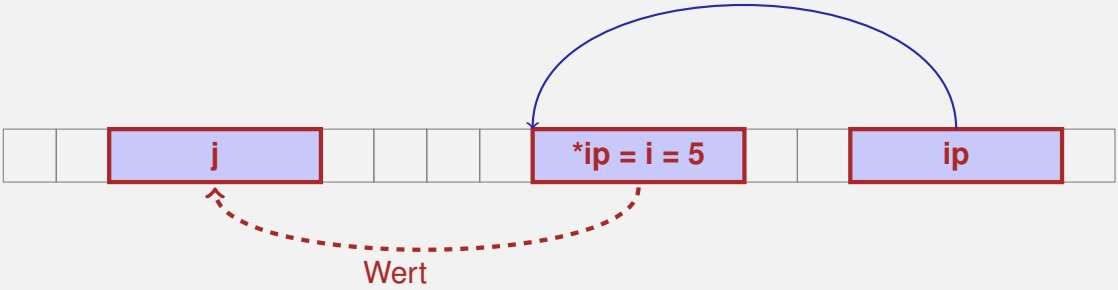
```
int i = 5;
int* ip = &i;  // ip initialisiert
               // mit Adresse von i.
int j = *ip;   // j == 5
```



Dereferenz-Operator

Beispiel

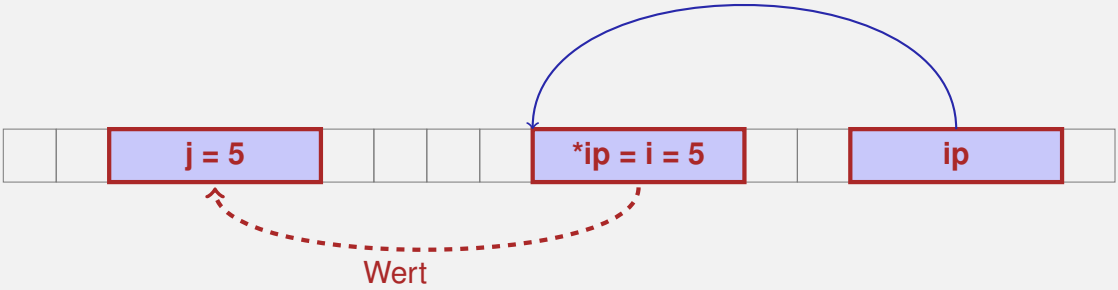
```
int i = 5;
int* ip = &i;  // ip initialisiert
               // mit Adresse von i.
int j = *ip;   // j == 5
```



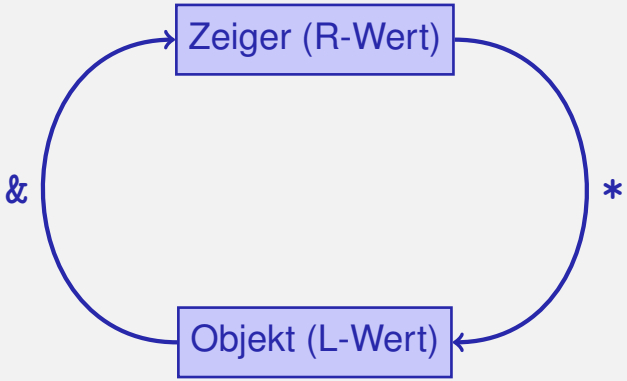
Dereferenz-Operator

Beispiel

```
int i = 5;
int* ip = &i;  // ip initialisiert
               // mit Adresse von i.
int j = *ip;   // j == 5
```



Adress- und Dereferenzoperator



Zeiger-Typen

Man zeigt nicht mit einem `double*` auf einen `int`!

Zeiger-Typen

Man zeigt nicht mit einem `double*` auf einen `int`!

Beispiele

```
int* i = ...;    // an Adresse i "wohnt" ein int...
double* j = i;   //...und an j ein double: Fehler!
```

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
```

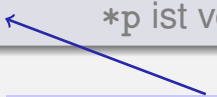
Eselsbrücke

Die Deklaration

```
T* p;        p ist vom Typ “Zeiger auf T”
```

kann gelesen werden als

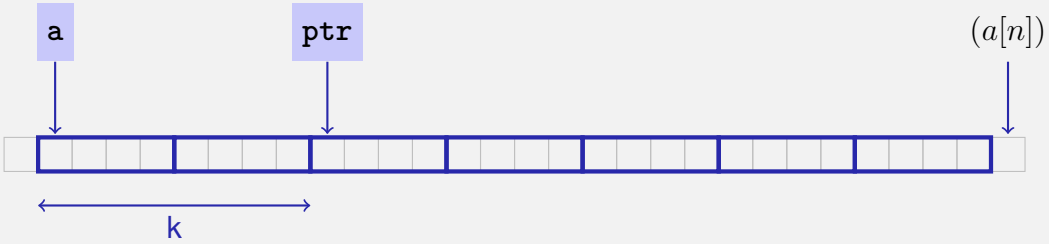
```
T *p;        *p ist vom Typ T
```



Obwohl das legal ist, schreiben wir es nicht so!

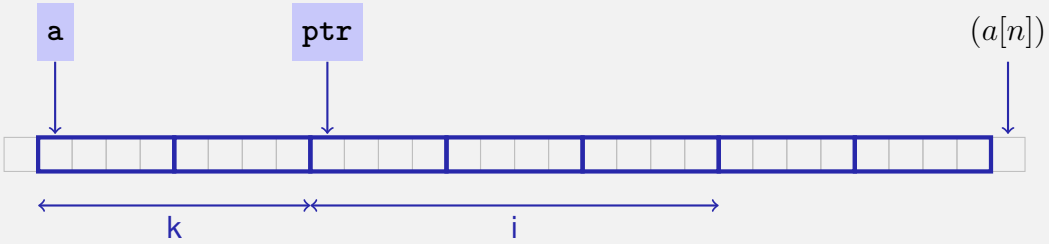
Zeiger-Arithmetik: Zeiger plus `int`

- *ptr*: Zeiger auf Element $a[k]$ des Arrays a mit Länge n



Zeiger-Arithmetik: Zeiger plus `int`

- *ptr*: Zeiger auf Element $a[k]$ des Arrays a mit Länge n
- Wert von *expr*: ganze Zahl i mit $0 \leq k + i \leq n$

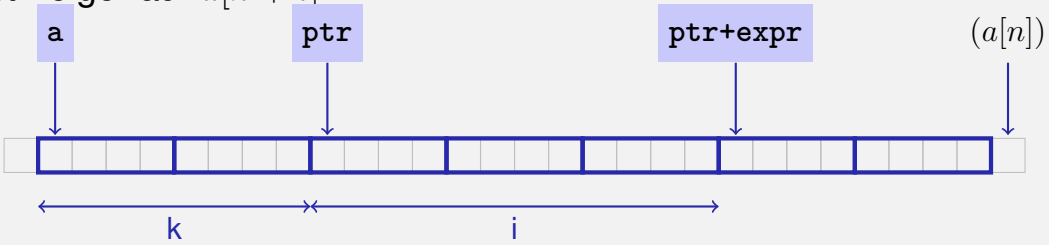


Zeiger-Arithmetik: Zeiger plus `int`

- *ptr*: Zeiger auf Element $a[k]$ des Arrays a mit Länge n
- Wert von *expr*: ganze Zahl i mit $0 \leq k + i \leq n$

ptr + expr

ist Zeiger auf $a[k + i]$.



Konversion Feld $\Rightarrow$ Zeiger

Wie bekommen wir einen Zeiger auf das erste Element eines Feldes?

Konversion Feld $\Rightarrow$ Zeiger

Wie bekommen wir einen Zeiger auf das erste Element eines Feldes?

- Statisches Feld vom Typ $T[\mathrm{n}]$ ist konvertierbar nach $T*$

Beispiel

```
int a[5];
int* begin = a; // begin zeigt auf a[0]
```

Konversion Feld $\Rightarrow$ Zeiger

Wie bekommen wir einen Zeiger auf das erste Element eines Feldes?

- Statisches Feld vom Typ $T[\mathrm{n}]$ ist konvertierbar nach $T*$

Beispiel

```
int a[5];
int* begin = a; // begin zeigt auf a[0]
```

- Längeninformation geht verloren („Felder sind primitiv”).

Iteration über ein Feld mit Zeigern

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

Iteration über ein Feld mit Zeigern

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

- `a+5` ist ein Zeiger direkt hinter das Ende des Feldes (past-the-end), der nicht dereferenziert werden darf.

Iteration über ein Feld mit Zeigern

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

- `a+5` ist ein Zeiger direkt hinter das Ende des Feldes (past-the-end), der nicht dereferenziert werden darf.
- Zeigervergleich (`p < a+5`) bezieht sich auf die Reihenfolge der beiden Adressen im Speicher.

14. Zeiger, Algorithmen, Iteratoren und Container II

Iteration mit Zeigern, Felder: Indizes vs. Zeiger, Felder und Funktionen, Zeiger und const, Algorithmen, Container und Traversierung, Vektor-Iteratoren, Typedef, Mengen, das Iterator-Konzept

Zur Erinnerung: Mit Zeigern übers Feld

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

Zur Erinnerung: Mit Zeigern übers Feld

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

- Ein Feld ist in einen Zeiger auf sein erstes Element konvertierbar.

Zur Erinnerung: Mit Zeigern übers Feld

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

- Ein Feld ist in einen Zeiger auf sein erstes Element konvertierbar.
- Zeiger kennen Arithmetik

Zur Erinnerung: Mit Zeigern übers Feld

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

- Ein Feld ist in einen Zeiger auf sein erstes Element konvertierbar.
- Zeiger kennen Arithmetik und Vergleiche.

Zur Erinnerung: Mit Zeigern übers Feld

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

- Ein Feld ist in einen Zeiger auf sein erstes Element konvertierbar.
- Zeiger kennen Arithmetik und Vergleiche.
- Zeiger können dereferenziert werden.

Zur Erinnerung: Mit Zeigern übers Feld

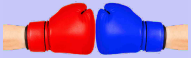
Beispiel

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

- Ein Feld ist in einen Zeiger auf sein erstes Element konvertierbar.
- Zeiger kennen Arithmetik und Vergleiche.
- Zeiger können dereferenziert werden.

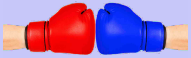
$\Rightarrow$ Mit Zeigern kann man auf Feldern operieren.

Felder: Indizes vs. Zeiger



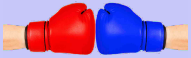
```
int a[n];

// Aufgabe: setze alle Elemente auf 0
```

Felder: Indizes vs. Zeiger



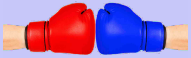
```
int a[n];

// Aufgabe: setze alle Elemente auf 0

// Lösung mit Indizes
for (int i = 0; i < n; ++i)
    a[i] = 0;
```

Felder: Indizes vs. Zeiger



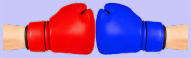
```
int a[n];

// Aufgabe: setze alle Elemente auf 0

// Lösung mit Indizes
for (int i = 0; i < n; ++i)
    a[i] = 0;

// Lösung mit Zeigern
int* begin = a;  // Zeiger aufs erste Element
int* end = a+n;  // Zeiger hinter das letzte Element
for (int* p = begin; p != end; ++p)
   *p = 0;
```

Felder: Indizes vs. Zeiger



```
int a[n];

// Aufgabe: setze alle Elemente auf 0

// Lösung mit Indizes ist lesbarer
for (int i = 0; i < n; ++i)
    a[i] = 0;

// Lösung mit Zeigern
int* begin = a;  // Zeiger aufs erste Element
int* end = a+n;  // Zeiger hinter das letzte Element
for (int* p = begin; p != end; ++p)
   *p = 0;
```

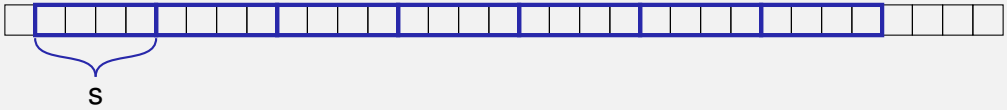
Felder: Indizes vs. Zeiger



```
int a[n];

// Aufgabe: setze alle Elemente auf 0

// Lösung mit Indizes ist lesbarer
for (int i = 0; i < n; ++i)
    a[i] = 0;

// Lösung mit Zeigern ist schneller und allgemeiner
int* begin = a;  // Zeiger aufs erste Element
int* end = a+n;  // Zeiger hinter das letzte Element
for (int* p = begin; p != end; ++p)
   *p = 0;
```

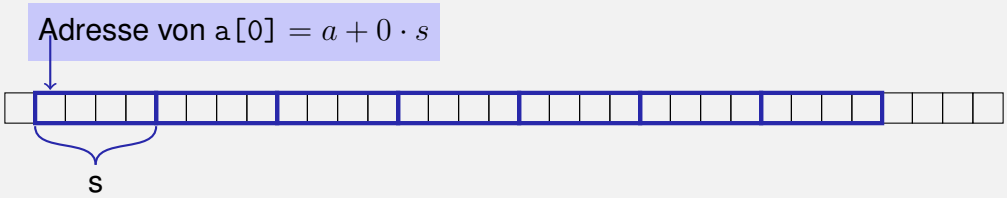
Felder und Indizes

```
// Setze alle Elemente auf value
for (int i = 0; i < n; ++i)
   a[i] = value;
```



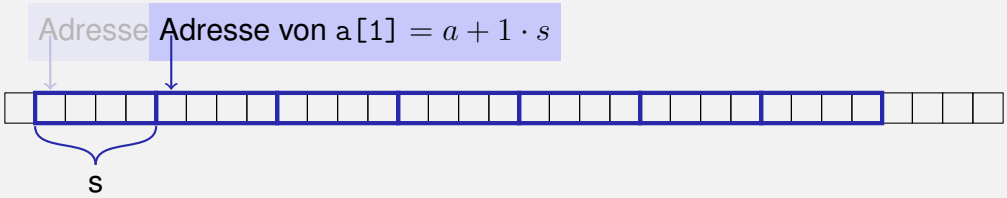
Felder und Indizes

```
// Setze alle Elemente auf value
for (int i = 0; i < n; ++i)
   a[i] = value;
```



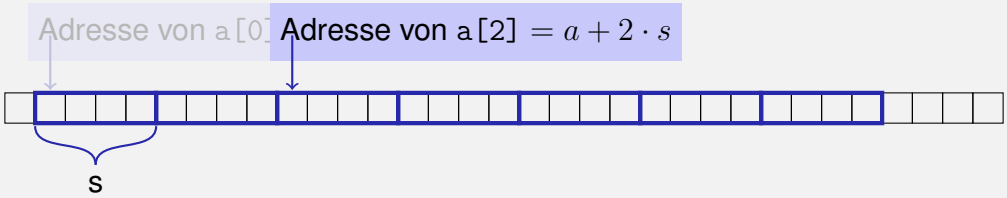
Felder und Indizes

```
// Setze alle Elemente auf value
for (int i = 0; i < n; ++i)
   a[i] = value;
```



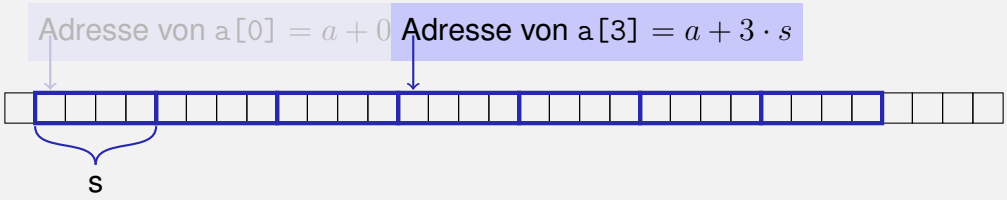
Felder und Indizes

```
// Setze alle Elemente auf value
for (int i = 0; i < n; ++i)
   a[i] = value;
```



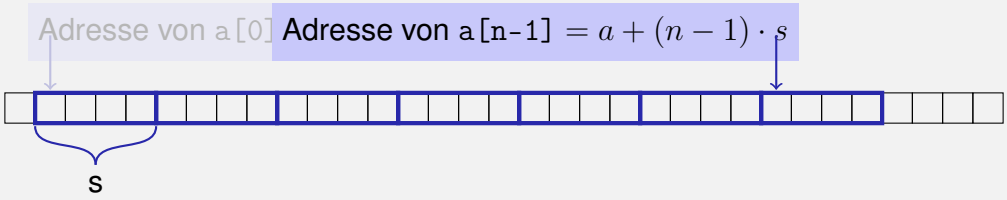
Felder und Indizes

```
// Setze alle Elemente auf value
for (int i = 0; i < n; ++i)
   a[i] = value;
```



Felder und Indizes

```
// Setze alle Elemente auf value
for (int i = 0; i < n; ++i)
   a[i] = value;
```



$\Rightarrow$ Eine **Addition** und eine **Multiplikation** pro Element

Die Wahrheit über wahlfreien Zugriff

Der Ausdruck

```
a[i]
```

ist äquivalent zu

```
*(a + i)
```

$a + i \cdot s$

Die Wahrheit über wahlfreien Zugriff

Der Ausdruck

```
a[i]
```

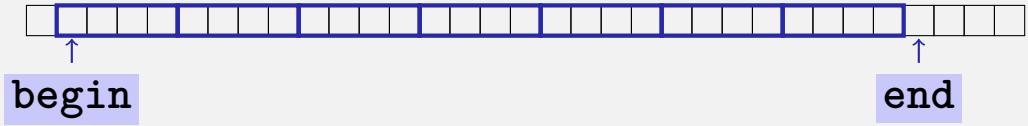
ist äquivalent zu

```
*(a + i)
```

$$a + i \cdot s$$

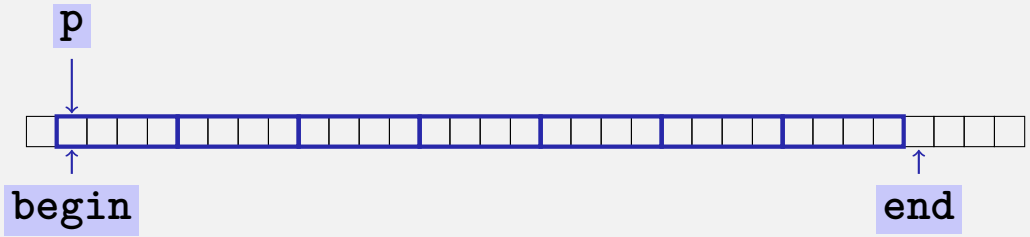
Felder und Zeiger

```
// Setze alle Elemente auf value
for (int* p = begin; p != end; ++p)
   *p = value;
```



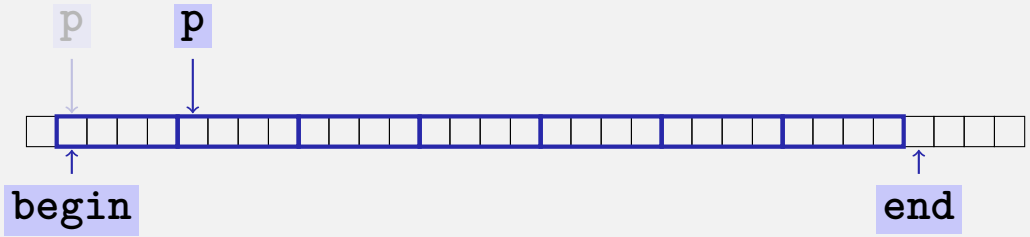
Felder und Zeiger

```
// Setze alle Elemente auf value
for (int* p = begin; p != end; ++p)
   *p = value;
```



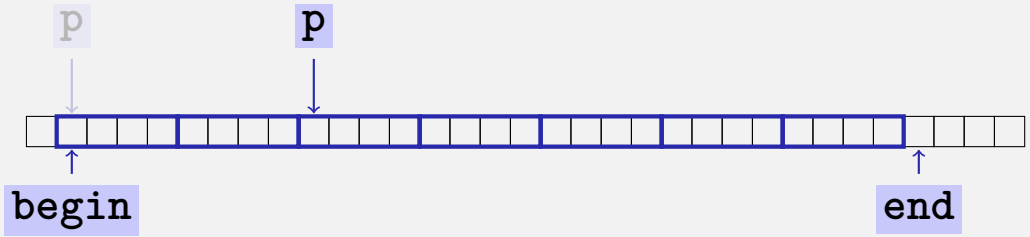
Felder und Zeiger

```
// Setze alle Elemente auf value
for (int* p = begin; p != end; ++p)
   *p = value;
```



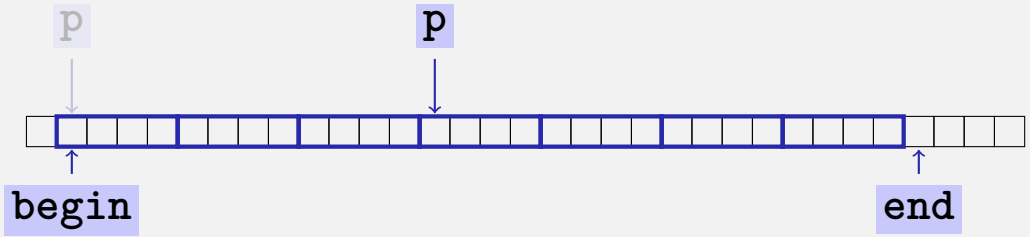
Felder und Zeiger

```
// Setze alle Elemente auf value
for (int* p = begin; p != end; ++p)
   *p = value;
```



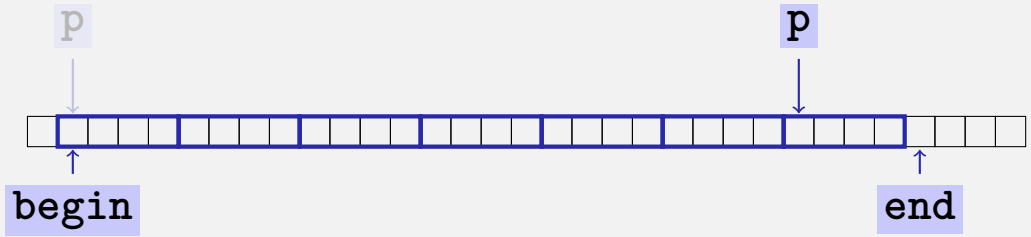
Felder und Zeiger

```
// Setze alle Elemente auf value
for (int* p = begin; p != end; ++p)
   *p = value;
```



Felder und Zeiger

```
// Setze alle Elemente auf value
for (int* p = begin; p != end; ++p)
   *p = value;
```



$\Rightarrow$ eine **Addition** pro Element

Ein Buch lesen . . . mit Indizes

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
- ....

Ein Buch lesen

. . . mit Zeigern

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
- ....

Sequentieller Zugriff

- öffne Buch auf S.1
- blättere um
- blättere um
- blättere um
- blättere um
- blättere um
- ...

Feldargumente: *Call by (const) reference*

```
void print_vector (const int (&v)[3]) {
  for (int i = 0; i<3 ; ++i) {
    std::cout << v[i] << " ";
  }
}
void make_null_vector (int (&v)[3]) {
  for (int i = 0; i<3 ; ++i) {
    v[i] = 0;
  }
}
```

Feldargumente: *Call by value*

```
void make_null_vector (int v[3]) {
  for (int i = 0; i<3 ; ++i) {
    v[i] = 0;
  }
}
...
```

Feldargumente: *Call by value (nicht wirklich...)*

```
void make_null_vector (int v[3]) {
  for (int i = 0; i<3 ; ++i) {
    v[i] = 0;
  }
}
...
int a[10];
make_null_vector (a); // setzt nur a[0], a[1], a[2]
```

Feldargumente: *Call by value (nicht wirklich...)*

```
void make_null_vector (int v[3]) {
  for (int i = 0; i<3 ; ++i) {
    v[i] = 0;
  }
}
...
int a[10];
make_null_vector (a); // setzt nur a[0], a[1], a[2]

int* b;
make_null_vector (b); // kein Feld bei b, Crash!
```

Feldargumente: *Call by value* gibt's nicht

- Formale Argumenttypen *T* `[n]` oder *T* `[]` (Feld über `T`) sind äquivalent zu *T*$*$ (Zeiger auf `T`)

Feldargumente: *Call by value* gibt's nicht

- Formale Argumenttypen *T* `[n]` oder *T* `[]` (Feld über `T`) sind äquivalent zu *T*$*$ (Zeiger auf `T`)
- Bei der Übergabe eines Feldes wird ein Zeiger auf das erste Element übergeben

Feldargumente: *Call by value* gibt's nicht

- Formale Argumenttypen T `[n]` oder T `[]` (Feld über `T`) sind äquivalent zu $T*$ (Zeiger auf `T`)
- Bei der Übergabe eines Feldes wird ein Zeiger auf das erste Element übergeben
- Längeninformation geht verloren

Feldargumente: *Call by value* gibt's nicht

- Formale Argumenttypen T`[n]` oder T`[]` (Feld über `T`) sind äquivalent zu $T*$ (Zeiger auf `T`)
- Bei der Übergabe eines Feldes wird ein Zeiger auf das erste Element übergeben
- Längeninformation geht verloren
- Funktion kann keinen Feldausschnitt verarbeiten (Beispiel: Suche eines Elements nur im hinteren Teil des Feldes)

Felder in Funktionen

Konvention der Standard-Bibliothek: Übergabe eines Feldes (oder eines Feldausschnitts) mit zwei Zeigern:

Felder in Funktionen

Konvention der Standard-Bibliothek: Übergabe eines Feldes (oder eines Feldausschnitts) mit zwei Zeigern:

- `begin`: Zeiger auf das erste Element
- `end`: Zeiger *hinter* das letzte Element

Felder in Funktionen

Konvention der Standard-Bibliothek: Übergabe eines Feldes (oder eines Feldausschnitts) mit zwei Zeigern:

- `begin`: Zeiger auf das erste Element
- `end`: Zeiger *hinter* das letzte Element
- `[begin, end)` bezeichnet die Elemente des Feldausschnitts

Felder in Funktionen

Konvention der Standard-Bibliothek: Übergabe eines Feldes (oder eines Feldausschnitts) mit zwei Zeigern:

- `begin`: Zeiger auf das erste Element
- `end`: Zeiger *hinter* das letzte Element
- `[begin, end)` bezeichnet die Elemente des Feldausschnitts
- *Gültiger* Bereich heisst: hier “leben” wirklich Feldelemente

Felder in Funktionen

Konvention der Standard-Bibliothek: Übergabe eines Feldes (oder eines Feldausschnitts) mit zwei Zeigern:

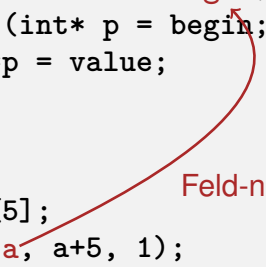
- `begin`: Zeiger auf das erste Element
- `end`: Zeiger *hinter* das letzte Element
- `[begin, end)` bezeichnet die Elemente des Feldausschnitts
- *Gültiger* Bereich heisst: hier “leben” wirklich Feldelemente
- `[begin, end)` ist leer, wenn `begin == end`

Felder in Funktionen: fill

```
// PRE: [begin, end) ist ein gueltiger Bereich
// POST: Jedes Element in [begin, end) wird auf value gesetzt
void fill (int* begin, int* end, int value) {
  for (int* p = begin; p != end; ++p)
     *p = value;
}
  ...

int a[5];
fill (a, a+5, 1);
for (int i=0; i<5; ++i)
  std::cout << a[i] << " ";  // 1 1 1 1 1
```

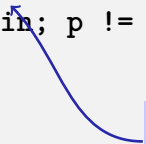
Felder in Funktionen: `fill`

```
// PRE: [begin, end) ist ein gueltiger Bereich
// POST: Jedes Element in [begin, end) wird auf value gesetzt
void fill (int* begin, int* end, int value) {
  for (int* p = begin; p != end; ++p)
     *p = value;
}
  ...

int a[5];
fill (a, a+5, 1);
for (int i=0; i<5; ++i)
  std::cout << a[i] << " ";  // 1 1 1 1 1
```



Felder in Funktionen: fill

```
// PRE: [begin, end) ist ein gueltiger Bereich
// POST: Jedes Element in [begin, end) wird auf value gesetzt
void fill (int* begin, int* end, int value) {
  for (int* p = begin; p != end; ++p)
     *p = value;
}
  ...

int a[5];
fill (a, a+5, 1);
for (int i=0; i<5; ++i)
  std::cout << a[i] << " ";  // 1 1 1 1 1
```



Erwartet Zeiger auf das erste Element eines Bereichs

Felder in Funktionen: fill

```
// PRE: [begin, end) ist ein gueltiger Bereich
// POST: Jedes Element in [begin, end) wird auf value gesetzt
void fill (int* begin, int* end, int value) {
  for (int* p = begin; p != end; ++p)
     *p = value;
}
  ...

int a[5];
fill (a, a+5, 1);
for (int i=0; i<5; ++i)
  std::cout << a[i] << " ";
```

Erwartet Zeiger auf das erste Element eines Bereichs

Übergabe der Adresse (des ersten Elements) von a

Mutierende Funktionen

- Zeiger können (wie auch Referenzen) für Funktionen mit Effekt verwendet werden.

Mutierende Funktionen

- Zeiger können (wie auch Referenzen) für Funktionen mit Effekt verwendet werden.

Beispiel

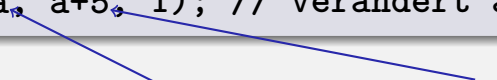
```
int a[5];
fill(a, a+5, 1); // verändert a
```

Mutierende Funktionen

- Zeiger können (wie auch Referenzen) für Funktionen mit Effekt verwendet werden.

Beispiel

```
int a[5];
fill(a, a+5, 1); // verändert a
```



Übergabe der Adresse des Elements hinter `a`

Übergabe der Adresse (des ersten Elements) von `a`

Nicht-mutierende Funktionen

```
// PRE: [begin , end) is a valid and nonempty range
// POST: the smallest value in [begin, end) is returned
int min ( int* begin , int* end)
{
    assert (begin != end);
    int m = *begin; // current minimum candidate
    for ( int* p = ++begin; p != end; ++p)
        if (*p < m) m = *p;
    return m;
}
```

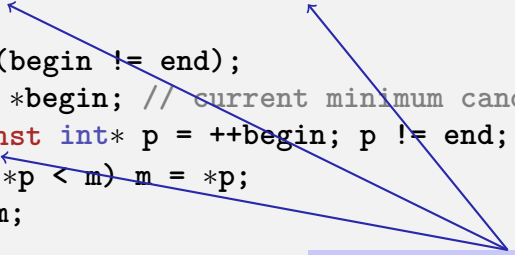
Nicht-mutierende Funktionen

```
// PRE: [begin , end) is a valid and nonempty range
// POST: the smallest value in [begin, end) is returned
int min ( int* begin , int* end)
{
    assert (begin != end);
    int m = *begin; // current minimum candidate
    for ( int* p = ++begin; p != end; ++p)
        if (*p < m) m = *p;
    return m;
}
```

- Kennzeichnung mit `const`

Nicht-mutierende Funktionen

```
// PRE: [begin , end) is a valid and nonempty range
// POST: the smallest value in [begin, end) is returned
int min (const int* begin ,const int* end)
{
    assert (begin != end);
    int m = *begin; // current minimum candidate
    for (const int* p = ++begin; p != end; ++p)
        if (*p < m) m = *p;
    return m;
}
```



`const` bezieht sich auf `int`, nicht auf den Zeiger.

- Kennzeichnung mit `const`

Const und Zeiger

const T ist äquivalent zu *T const* und kann auch so geschrieben werden

```
const int a;    ⟺  int const a;
const int* a;   ⟺  int const *a;
```

Const und Zeiger

Lies Deklaration von rechts nach links

```
int const a;
```

`a` ist eine konstante Ganzzahl

Const und Zeiger

Lies Deklaration von rechts nach links

`int const* a;`	`a` ist ein Zeiger auf eine konstante Ganzzahl

Const und Zeiger

Lies Deklaration von rechts nach links

`int* const a;`	`a` ist ein konstanter Zeiger auf eine Ganzzahl

Const und Zeiger

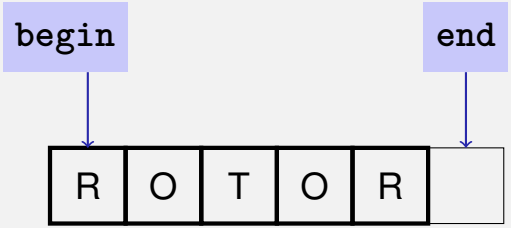
Lies Deklaration von rechts nach links

```
int const* const a;
```

`a` ist ein konstanter Zeiger auf eine konstante Ganzzahl

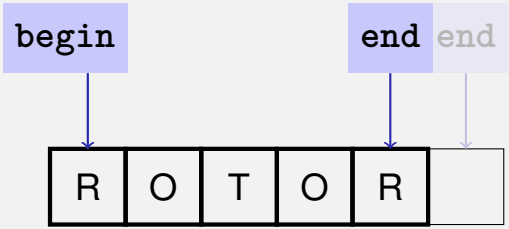
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
  while (begin < end)
    if (*(begin++) != *(--end)) return false;
  return true;
}
```



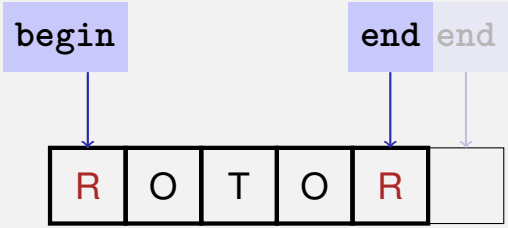
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
  while (begin < end)
    if (*(begin++) != *(--end)) return false;
  return true;
}
```



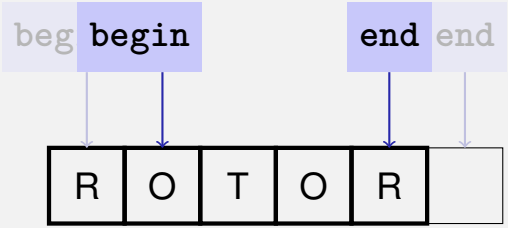
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
  while (begin < end)
    if (*(begin++) != *(--end)) return false;
  return true;
}
```



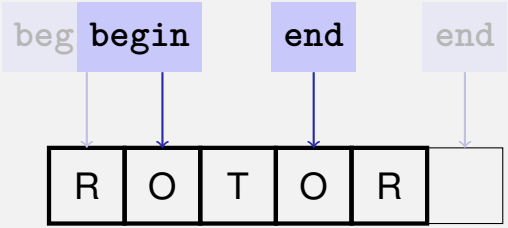
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
  while (begin < end)
    if (*(begin++) != *(--end)) return false;
  return true;
}
```



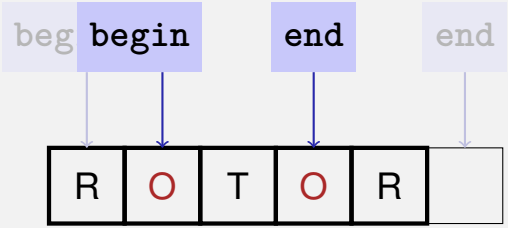
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
  while (begin < end)
    if (*(begin++) != *(--end)) return false;
  return true;
}
```



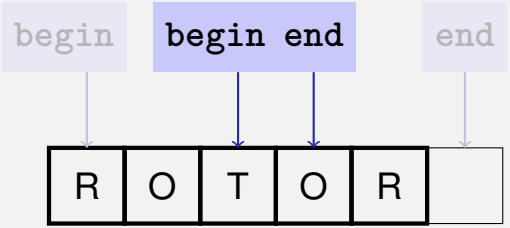
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
  while (begin < end)
    if (*(begin++) != *(--end)) return false;
  return true;
}
```



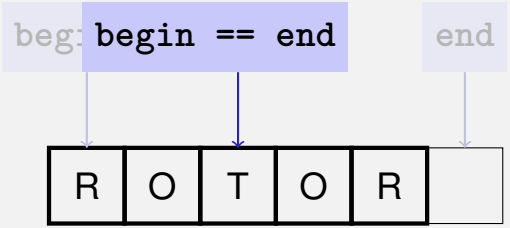
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
  while (begin < end)
    if (*(begin++) != *(--end)) return false;
  return true;
}
```



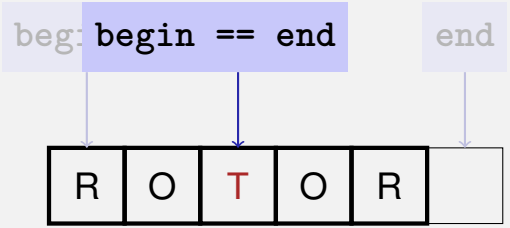
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
  while (begin < end)
    if (*(begin++) != *(--end)) return false;
  return true;
}
```



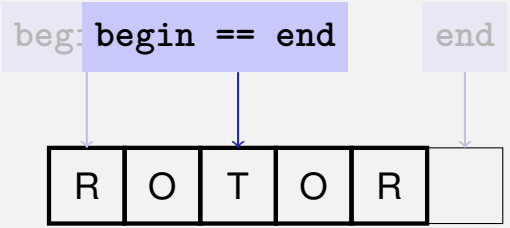
Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
  while (begin < end)
    if (*(begin++) != *(--end)) return false;
  return true;
}
```



Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
  while (begin < end)
    if (*(begin++) != *(--end)) return false;
  return true;
}
```



Algorithmen

Für viele alltägliche Probleme existieren vorgefertigte Lösungen in der Standardbibliothek.

Beispiel: Füllen eines Feldes

```
#include <algorithm> // needed for std::fill
...

int a[5];
std::fill (a, a+5, 1);

for (int i=0; i<5; ++i)
    std::cout << a[i] << " "; // 1 1 1 1 1
```

Algorithmen

Die gleichen vorgefertigten Algorithmen funktionieren für viele verschiedene Datentypen.

Beispiel: Füllen eines Feldes

```
#include <algorithm> // needed for std::fill
...

char c[3];
std::fill (c, c+3, '!');

for (int i=0; i<3; ++i)
    std::cout << c[i]; // !!!
```

Exkurs: Templates

Beispiel: `fill` mit Templates

```
template <typename T>
void fill (T* begin , T* end, T value) {
    for (T* p = begin; p != end; ++p)
        *p = value;
}
int a[5];
fill (a, a+5, 1);   // 1 1 1 1 1

char c[3];
fill (c, c+3, '!'); // !!!
```

Exkurs: Templates

Beispiel: `fill` mit Templates

```
template <typename T>
void fill (T* begin , T* end, T value) {
    for (T* p = begin; p != end; ++p)
        *p = value;
}
int a[5];
fill (a, a+5, 1);   // 1 1 1 1 1

char c[3];
fill (c, c+3, '!'); // !!!
```

Die eckigen Klammern kennen wir schon von `std::vector<int>`. Vektoren sind auch als Templates realisiert.

Exkurs: Templates

Beispiel: `fill` mit Templates

```
template <typename T>
void fill (T* begin , T* end, T value) {
    for (T* p = begin; p != end; ++p)
        *p = value;
}
int a[5];
fill (a, a+5, 1);   // 1 1 1 1 1

char c[3];
fill (c, c+3, '!'); // !!!
```

Die eckigen Klammern kennen wir schon von `std::vector<int>`. Vektoren sind auch als Templates realisiert.

Auch `std::fill` ist als Template realisiert!

Container und Traversierung

- **Container:** Behälter (Feld, Vektor,...) für Elemente

Container und Traversierung

- **Container:** Behälter (Feld, Vektor,...) für Elemente
- **Traversierung:** Durchlaufen eines Containers

Container und Traversierung

- **Container:** Behälter (Feld, Vektor,...) für Elemente
- **Traversierung:** Durchlaufen eines Containers
  - Initialisierung der Elemente (`fill`)

Container und Traversierung

- **Container:** Behälter (Feld, Vektor,...) für Elemente
- **Traversierung:** Durchlaufen eines Containers
  - Initialisierung der Elemente (`fill`)
  - Suchen des kleinsten Elements (`min`)

Container und Traversierung

- **Container:** Behälter (Feld, Vektor,...) für Elemente
- **Traversierung:** Durchlaufen eines Containers
  - Initialisierung der Elemente (`fill`)
  - Suchen des kleinsten Elements (`min`)
  - Prüfen von Eigenschaften (`is_palindrome`)

Container und Traversierung

- **Container:** Behälter (Feld, Vektor,...) für Elemente
- **Traversierung:** Durchlaufen eines Containers
  - Initialisierung der Elemente (`fill`)
  - Suchen des kleinsten Elements (`min`)
  - Prüfen von Eigenschaften (`is_palindrome`)
  - ...

Container und Traversierung

- **Container:** Behälter (Feld, Vektor,...) für Elemente
- **Traversierung:** Durchlaufen eines Containers
  - Initialisierung der Elemente (`fill`)
  - Suchen des kleinsten Elements (`min`)
  - Prüfen von Eigenschaften (`is_palindrome`)
  - ...
- Es gibt noch viele andere Container (Mengen, Listen,...)

Werkzeuge zur Traversierung

- Felder: Indizes (wahlfrei) oder Zeiger (natürlich)

Werkzeuge zur Traversierung

- Felder: Indizes (wahlfrei) oder Zeiger (natürlich)
- Feld-Algorithmen (`std::`) benutzen Zeiger

```
int a[5];
std::fill (a, a+5, 1);   // 1 1 1 1 1
```

Werkzeuge zur Traversierung

- Felder: Indizes (wahlfrei) oder Zeiger (natürlich)
- Feld-Algorithmen (`std::`) benutzen Zeiger

```
int a[5];
std::fill (a, a+5, 1);   // 1 1 1 1 1
```

- Wie traversiert man Vektoren und andere Container?

```
std::vector<int> v (5, 0); // 0 0 0 0 0
std::fill (?, ?, 1);   // 1 1 1 1 1
```

Vektoren: *too sexy for pointers*

- Unser `fill` mit Templates funktioniert für Vektoren nicht...

Vektoren: *too sexy for pointers*

- Unser `fill` mit Templates funktioniert für Vektoren nicht...
- ...und `std::fill` so auch nicht:

```
std::vector<int> v (5, 0);
std::fill (v, v+5, 1);   // Fehlermeldung des Compilers !
```

Vektoren: *too sexy for pointers*

- Unser `fill` mit Templates funktioniert für Vektoren nicht...
- ...und `std::fill` so auch nicht:

```
std::vector<int> v (5, 0);
std::fill (v, v+5, 1);   // Fehlermeldung des Compilers !
```

Vektoren sind was Besseres...

Vektoren: *too sexy for pointers*

- Unser `fill` mit Templates funktioniert für Vektoren nicht...
- ...und `std::fill` so auch nicht:

```
std::vector<int> v (5, 0);
std::fill (v, v+5, 1);   // Fehlermeldung des Compilers !
```

Vektoren sind was Besseres...

- Sie lassen sich weder in Zeiger konvertieren,...

Vektoren: *too sexy for pointers*

- Unser `fill` mit Templates funktioniert für Vektoren nicht...
- ...und `std::fill` so auch nicht:

```
std::vector<int> v (5, 0);
std::fill (v, v+5, 1);   // Fehlermeldung des Compilers !
```

Vektoren sind was Besseres...

- Sie lassen sich weder in Zeiger konvertieren,...
- ...noch mit Zeigern traversieren.

Vektoren: *too sexy for pointers*

- Unser `fill` mit Templates funktioniert für Vektoren nicht...
- ...und `std::fill` so auch nicht:

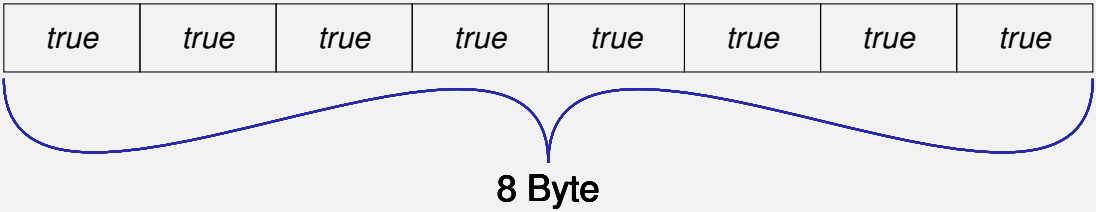
```
std::vector<int> v (5, 0);
std::fill (v, v+5, 1);   // Fehlermeldung des Compilers !
```

Vektoren sind was Besseres...

- Sie lassen sich weder in Zeiger konvertieren,...
- ...noch mit Zeigern traversieren.
- Das ist ihnen viel zu primitiv.☺

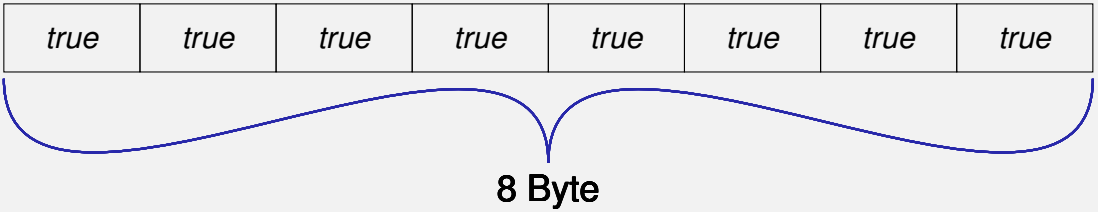
Auch im Speicher: Vektor $\neq$ Feld

```
bool a[8] = {true, true, true, true, true, true, true, true};
```

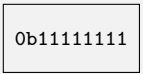


Auch im Speicher: Vektor $\neq$ Feld

```
bool a[8] = {true, true, true, true, true, true, true, true};
```



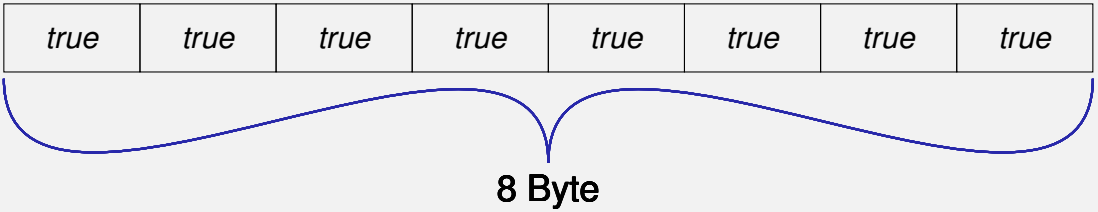

```
std::vector<bool> v (8, true);
```



1 Byte

Auch im Speicher: Vektor $\neq$ Feld

```
bool a[8] = {true, true, true, true, true, true, true, true};
```




```
std::vector<bool> v (8, true);
```

`0b11111111` 1 Byte

`bool*`-Zeiger passt hier nicht, denn er läuft byteweise, nicht bitweise!

Vektor-Iteratoren

Iterator: ein “Zeiger”, der zum Container passt.

Vektor-Iteratoren

Iterator: ein “Zeiger”, der zum Container passt.

Beispiel: Füllen eines Vektors mit `std::fill` – so geht’s!

```
#include <vector>
#include <algorithm> // needed for std::fill

...
std::vector<int> v(5, 0);
std::fill (v.begin(), v.end(), 1);
for (int i=0; i<5; ++i)
    std::cout << v[i] << " "; // 1 1 1 1 1
```

Vektor-Iteratoren

Für jeden Vektor sind zwei *Iterator-Typen* definiert.

- `std::vector<int>::const_iterator`

Vektor-Iteratoren

Für jeden Vektor sind zwei *Iterator-Typen* definiert.

- `std::vector<int>::const_iterator`
  - für nicht-mutierenden Zugriff

Vektor-Iteratoren

Für jeden Vektor sind zwei *Iterator-Typen* definiert.

- `std::vector<int>::const_iterator`
  - für nicht-mutierenden Zugriff
  - analog zu `const int*` für Felder

Vektor-Iteratoren

Für jeden Vektor sind zwei *Iterator-Typen* definiert.

- `std::vector<int>::const_iterator`
  - für nicht-mutierenden Zugriff
  - analog zu `const int*` für Felder
- `std::vector<int>::iterator`

Vektor-Iteratoren

Für jeden Vektor sind zwei *Iterator-Typen* definiert.

- `std::vector<int>::const_iterator`
  - für nicht-mutierenden Zugriff
  - analog zu `const int*` für Felder
- `std::vector<int>::iterator`
  - für mutierenden Zugriff

Vektor-Iteratoren

Für jeden Vektor sind zwei *Iterator-Typen* definiert.

- `std::vector<int>::const_iterator`
  - für nicht-mutierenden Zugriff
  - analog zu `const int*` für Felder
- `std::vector<int>::iterator`
  - für mutierenden Zugriff
  - analog zu `int*` für Felder

Vektor-Iteratoren: `begin()` und `end()`

- `v.begin()` zeigt auf das erste Element von `v`
- `v.end()` zeigt hinter das letzte Element von `v`

Vektor-Iteratoren: `begin()` und `end()`

- `v.begin()` zeigt auf das erste Element von `v`
- `v.end()` zeigt hinter das letzte Element von `v`

Vektor-Iteratoren: `begin()` und `end()`

- Damit können wir einen Vektor traversieren...

```
for (std::vector<int>::const_iterator it = v.begin();
                                      it != v.end(); ++it)
    std::cout << *it << " ";
```

- ...oder einen Vektor füllen.

```
std::fill (v.begin(), v.end(), 1);
```

Vektor-Iteratoren: `begin()` und `end()`

- Damit können wir einen Vektor traversieren...

```
for (std::vector<int>::const_iterator it = v.begin();
                                      it != v.end(); ++it)
    std::cout << *it << " ";
```

- ...oder einen Vektor füllen.

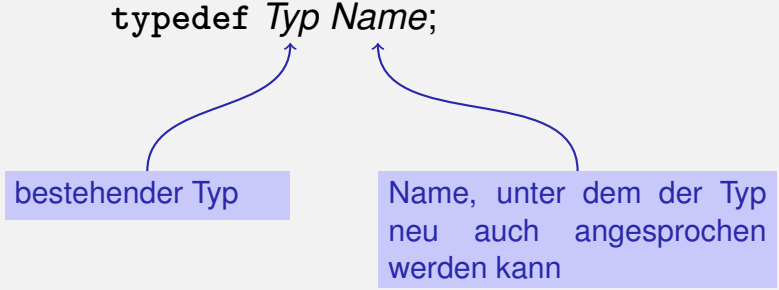
```
std::fill (v.begin(), v.end(), 1);
```

Typnamen in C++ können laaaaaaaang werden

- `std::vector<int>::const_iterator`

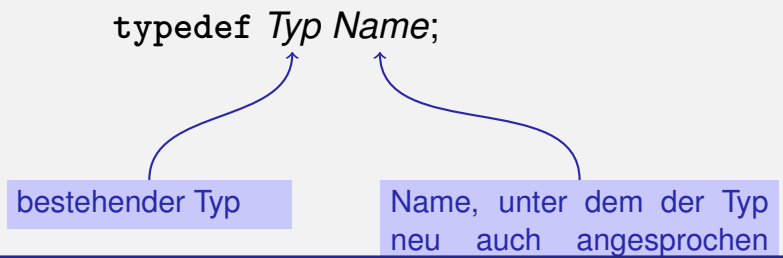
Typnamen in C++ können laaaaaaaang werden

- `std::vector<int>::const_iterator`
- Dann hilft die Deklaration eines *Typ-Alias* mit



Typnamen in C++ können laaaaaaaang werden

- `std::vector<int>::const_iterator`
- Dann hilft die Deklaration eines *Typ-Alias* mit



Beispiele

```
typedef std::vector<int> int_vec;
typedef int_vec::const_iterator Cvit;
```

Vektor-Iteratoren funktionieren wie Zeiger

```
typedef std::vector<int>::const_iterator Cvit;

std::vector<int> v(5, 0); // 0 0 0 0 0

// output all elements of a, using iteration
for (Cvit it = v.begin(); it != v.end(); ++it)
    std::cout << *it << " ";
```

Vektor-Iteratoren funktionieren wie Zeiger

```
typedef std::vector<int>::const_iterator Cvit;

std::vector<int> v(5, 0); // 0 0 0 0 0

// output all elements of a, using iteration
for (Cvit it = v.begin(); it != v.end(); ++it)
    std::cout << *it << " ";
```

Vektor-Element, auf das `it` zeigt

Vektor-Iteratoren funktionieren wie Zeiger

```
typedef std::vector<int>::iterator Vit;

// manually set all elements to 1
for (Vit it = v.begin(); it != v.end(); ++it)
    *it = 1;

// output all elements again, using random access
for (int i=0; i<5; ++i)
    std::cout << v[i] << " ";
```

Vektor-Iteratoren funktionieren wie Zeiger

```
typedef std::vector<int>::iterator Vit;

// manually set all elements to 1
for (Vit it = v.begin(); it != v.end(); ++it)
    *it = 1;

// output all elements again, using random access
for (int i=0; i<5; ++i)
    std::cout << v[i] << " ";
```

Inkrementieren des Iterators

Kurzschreibweise für `*(v.begin()+i)`

Andere Container: Mengen (Sets)

- Eine Menge ist eine ungeordnete Zusammenfassung von Elementen, wobei jedes Element nur einmal vorkommt.

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

Andere Container: Mengen (Sets)

- Eine Menge ist eine ungeordnete Zusammenfassung von Elementen, wobei jedes Element nur einmal vorkommt.

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- C++: `std::set<T>` für eine Menge mit Elementen vom Typ `T`

Mengen: Beispiel einer Anwendung

- Stelle fest, ob ein gegebener Text ein Fragezeichen enthält und gib alle im Text vorkommenden *verschiedenen* Zeichen aus!

Buchstabensalat (1)

```
#include<set>
...
typedef std::set<char>::const_iterator Csit;
...
std::string text =
"What are the distinct characters in this string?";

std::set<char> s (text.begin(),text.end());
```

Buchstabensalat (1)

```
#include<set>
...
typedef std::set<char>::const_iterator Csit;
...
std::string text =
"What are the distinct characters in this string?";

std::set<char> s (text.begin(),text.end());
```

Menge wird mit *String-Iterator-Bereich* `[text.begin(), text.end())` initialisiert

Buchstabensalat (2)

```
// check whether text contains a question mark
if (std::find (s.begin(), s.end(), '?') != s.end())
    std::cout << "Good question!\n";

// output all distinct characters
for (Csit it = s.begin(); it != s.end(); ++it)
    std::cout << *it;
```

Buchstabensalat (2)

Suchalgorithmus, aufrufbar mit beliebigem Iterator-Bereich

```
// check whether text contains a question mark
if (std::find (s.begin(), s.end(), '?') != s.end())
    std::cout << "Good question!\n";

// output all distinct characters
for (Csit it = s.begin(); it != s.end(); ++it)
    std::cout << *it;
```

Buchstabensalat (2)

Suchalgorithmus, aufrufbar mit beliebigem Iterator-Bereich

```
// check whether text contains a question mark
if (std::find (s.begin(), s.end(), '?') != s.end())
    std::cout << "Good question!\n";

// output all distinct characters
for (Csit it = s.begin(); it != s.end(); ++it)
    std::cout << *it;
```

Ausgabe:

```
Good question!
?Wacdeghinrst
```

Mengen und Indizes?

- Kann man Mengen mit wahlfreiem Zugriff traversieren?

Mengen und Indizes?

- Kann man Mengen mit wahlfreiem Zugriff traversieren?

```
for (int i=0; i<s.size(); ++i)
    std::cout << s[i];
```

Mengen und Indizes?

- Kann man Mengen mit wahlfreiem Zugriff traversieren?

```
for (int i=0; i<s.size(); ++i)
    std::cout << s[i];
```


Fehlermeldung: no subscript operator

Mengen und Indizes?

- Kann man Mengen mit wahlfreiem Zugriff traversieren? Nein.

```
for (int i=0; i<s.size(); ++i)
    std::cout << s[i];
```

Fehlermeldung: no subscript operator

- Mengen sind ungeordnet.
  - Es gibt kein "i-tes Element".

Mengen und Indizes?

- Kann man Mengen mit wahlfreiem Zugriff traversieren? Nein.

```
for (int i=0; i<s.size(); ++i)
    std::cout << s[i];
```


Fehlermeldung: no subscript operator

- Mengen sind ungeordnet.
  - Es gibt kein "i-tes Element".
  - Iteratorvergleich `it != s.end()` geht, nicht aber `it < s.end()`!

Das Konzept der Iteratoren

C++kennt verschiedene Iterator-Typen

Das Konzept der Iteratoren

C++kennt verschiedene Iterator-Typen

- Jeder Container hat einen zugehörigen Iterator-Typ

Das Konzept der Iteratoren

C++kennt verschiedene Iterator-Typen

- Jeder Container hat einen zugehörigen Iterator-Typ
- Alle können dereferenzieren (`*it`) und traversieren (`++it`)

Das Konzept der Iteratoren

C++kennt verschiedene Iterator-Typen

- Jeder Container hat einen zugehörigen Iterator-Typ
- Alle können dereferenzieren (`*it`) und traversieren (`++it`)
- Manche können mehr, z.B. wahlfreien Zugriff (`it[k]`, oder äquivalent `*(it + k)`), rückwärts traversieren (`--it`),...

Das Konzept der Iteratoren

Jeder Container-Algorithmus der Standardbibliothek ist *generisch*.
Das heisst:

Das Konzept der Iteratoren

Jeder Container-Algorithmus der Standardbibliothek ist *generisch*. Das heisst:

- Der Container wird per Iterator-Bereich übergeben

Das Konzept der Iteratoren

Jeder Container-Algorithmus der Standardbibliothek ist *generisch*. Das heisst:

- Der Container wird per Iterator-Bereich übergeben
- Der Algorithmus funktioniert für alle Container, deren Iteratoren die Anforderungen des Algorithmus erfüllen

Das Konzept der Iteratoren

Jeder Container-Algorithmus der Standardbibliothek ist *generisch*. Das heisst:

- Der Container wird per Iterator-Bereich übergeben
- Der Algorithmus funktioniert für alle Container, deren Iteratoren die Anforderungen des Algorithmus erfüllen

- `std::find` erfordert z.B. nur `*` und `++`

Das Konzept der Iteratoren

Jeder Container-Algorithmus der Standardbibliothek ist *generisch*. Das heisst:

- Der Container wird per Iterator-Bereich übergeben
- Der Algorithmus funktioniert für alle Container, deren Iteratoren die Anforderungen des Algorithmus erfüllen
- `std::find` erfordert z.B. nur `*` und `++`
- Implementationsdetails des Containers sind nicht von Bedeutung

15. Rekursion 1

Mathematische Rekursion, Terminierung, der Aufrufstapel, Beispiele, Rekursion vs. Iteration

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

Unendliche Rekursion

- ist so schlecht wie eine Endlosschleife...
- ...nur noch schlechter (“verbrennt” Zeit und Speicher)

```
void f()
{
  f(); // f() -> f() -> ... stack overflow
}
```

Mir san mir.

Bayerisches Lebensmotto

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

`fac(n)`:
terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Argument < n aufgerufen.

Rekursive Funktionen: Terminierung

Wie bei Schleifen brauchen wir

- Fortschritt Richtung Terminierung

`fac(n)`:
terminiert sofort für $n \leq 1$, andernfalls wird die Funktion rekursiv mit Argument < n aufgerufen.

„n wird mit jedem Aufruf kleiner."

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Aufruf von `fac(4)`

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 4
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 4
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Auswertung des Rückgabeausdrucks

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 4
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Rekursiver Aufruf mit Argument $n - 1 == 3$

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 3
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Es gibt jetzt zwei n. Das von `fac(4)` und das von `fac(3)`

Initialisierung des formalen Arguments

Rekursive Funktionen: Auswertung

Beispiel: `fac(4)`

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Es wird mit dem n des aktuellen Aufrufs gearbeitet: $n = 3$

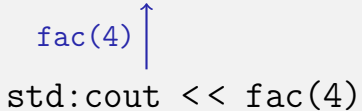
Initialisierung des formalen Arguments

Der Aufrufstapel

```
std:cout << fac(4)
```

Der Aufrufstapel

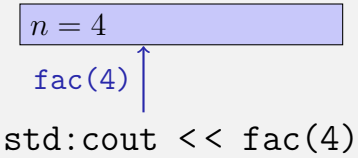
Bei jedem Funktionsaufruf:



Der Aufrufstapel

Bei jedem Funktionsaufruf:

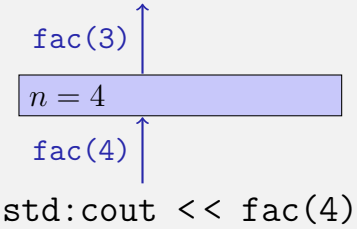
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

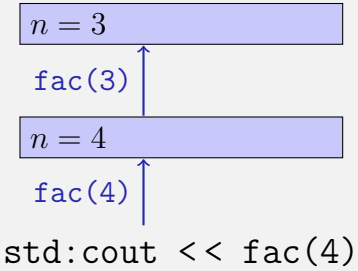
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

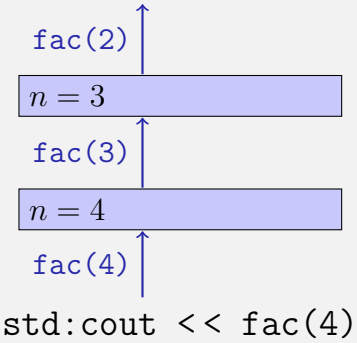
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

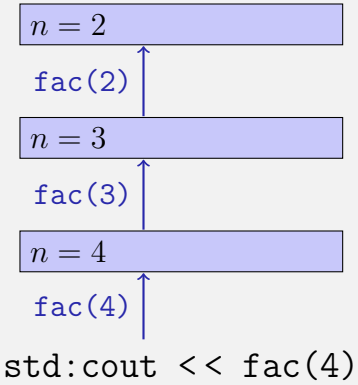
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

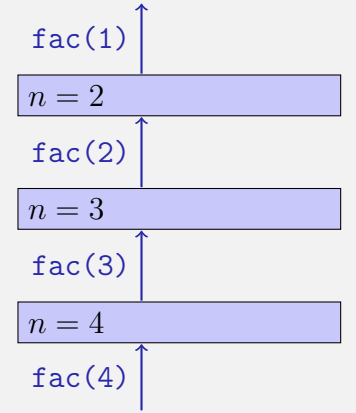
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

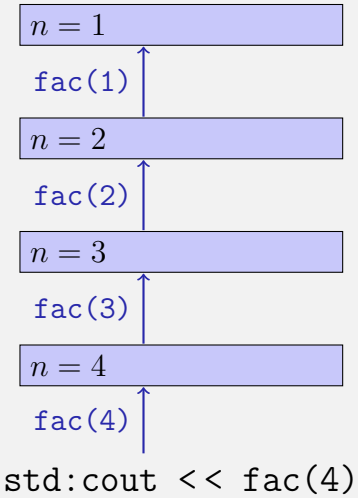
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

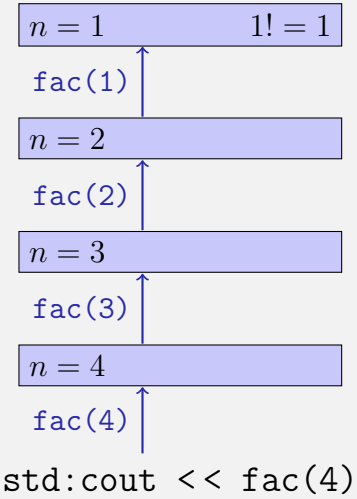
- Wert des Aufrufarguments kommt auf einen Stapel



Der Aufrufstapel

Bei jedem Funktionsaufruf:

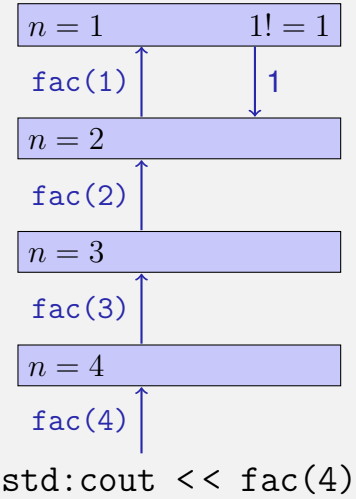
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet



Der Aufrufstapel

Bei jedem Funktionsaufruf:

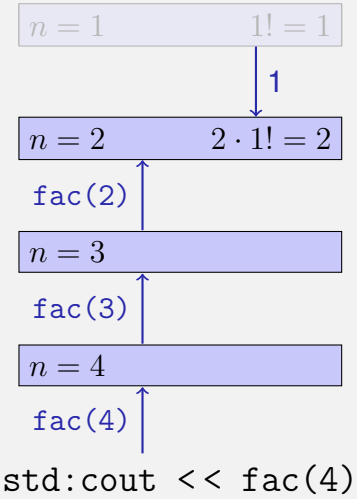
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

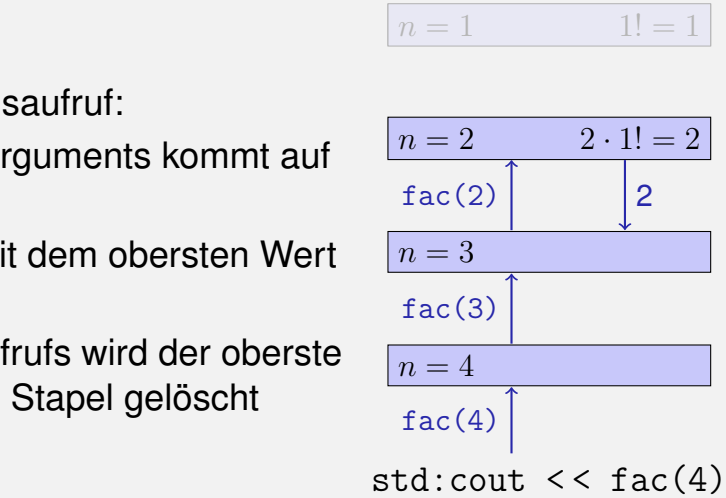
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

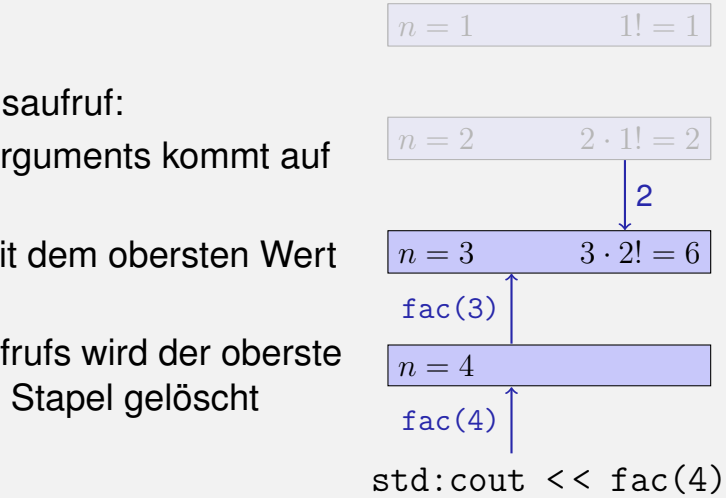
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

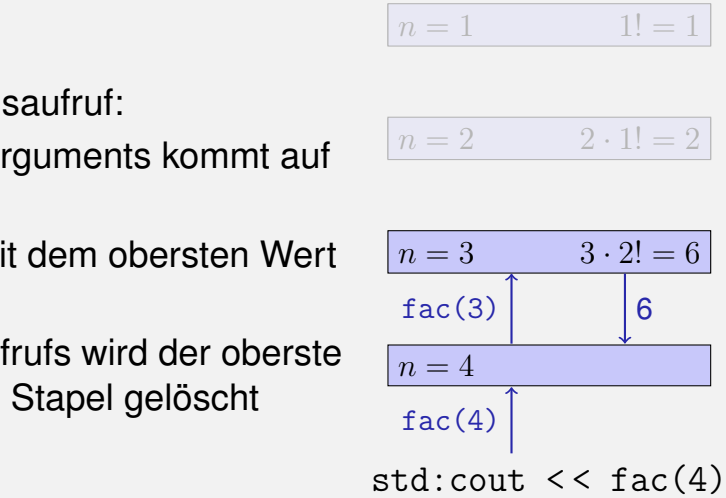
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

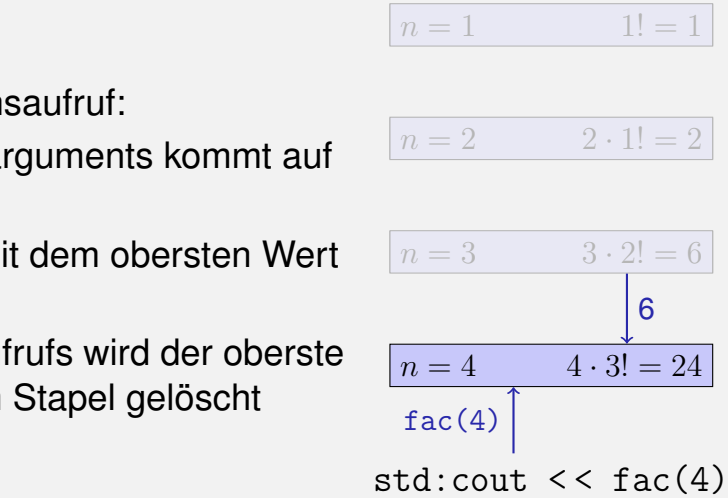
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

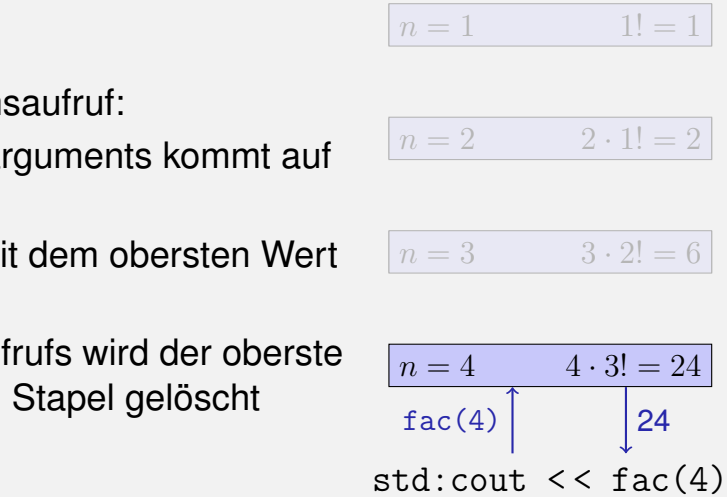
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



Der Aufrufstapel

Bei jedem Funktionsaufruf:

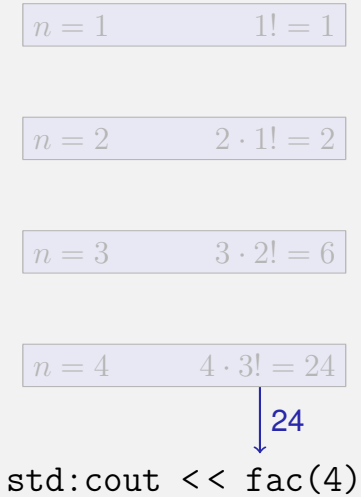
- Wert des Aufrufarguments kommt auf einen Stapel
- Es wird immer mit dem obersten Wert gearbeitet
- Am Ende des Aufrufs wird der oberste Wert wieder vom Stapel gelöscht



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



Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler $\gcd(a, b)$ zweier natürlicher Zahlen a und b

Euklidischer Algorithmus

- findet den grössten gemeinsamen Teiler $\gcd(a, b)$ zweier natürlicher Zahlen a und b
- basiert auf folgender mathematischen Rekursion (Beweis im Skript):

$$\gcd(a, b) = \begin{cases} a, & \text{falls } b = 0 \\ \gcd(b, a \bmod b), & \text{andernfalls} \end{cases}$$

Euklidischer Algorithmus in C++

$$\gcd(a,b) = \begin{cases} a, & \text{falls } b = 0 \\ \gcd(b, a \bmod b), & \text{andernfalls} \end{cases}$$

```
unsigned int gcd
 (unsigned int a, unsigned int b)
{
  if (b == 0)
    return a;
  else
    return gcd (b, a % b);
}
```

Euklidischer Algorithmus in C++

$$\gcd(a,b) = \begin{cases} a, & \text{falls } b = 0 \\ \gcd(b, a \bmod b), & \text{andernfalls} \end{cases}$$

```
unsigned int gcd
 (unsigned int a, unsigned int b)
{
  if (b == 0)
    return a;
  else
    return gcd (b, a % b);
}
```

Terminierung: $a \bmod b < b$, also wird b in jedem rekursiven Aufruf kleiner.

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

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \ldots$

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
}
```

Korrektheit und Terminierung sind klar.

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 $F_0, F_1, F_2, \ldots, F_n$!

Schnelle Fibonacci-Zahlen

Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$!
- Merke dir jeweils die zwei letzten berechneten Zahlen (Variablen `a` und `b`)!

Schnelle Fibonacci-Zahlen

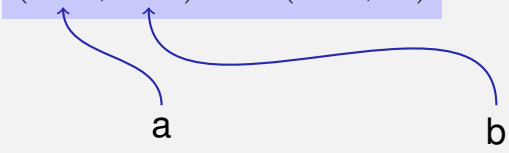
Idee:

- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$!
- Merke dir jeweils die zwei letzten berechneten Zahlen (Variablen `a` und `b`)!
- Berechne die nächste Zahl als Summe von `a` und `b`!

Schnelle Fibonacci-Zahlen in C++

```
unsigned int fib (unsigned int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i){
    unsigned int a_old = a;  // F_i-2
    a = b;                   // F_i-1
    b += a_old;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

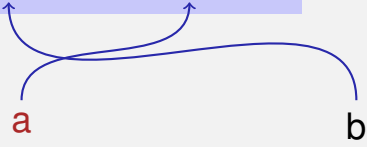
$$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$$



Schnelle Fibonacci-Zahlen in C++

```
unsigned int fib (unsigned int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i){
    unsigned int a_old = a;  // F_i-2
    a = b;                   // F_i-1
    b += a_old;              // F_i-1 += F_i-2 -> 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 <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i){
    unsigned int a_old = a;  // F_i-2
    a = b;                   // F_i-1
    b += a_old;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

$$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$$

a b

Schnelle Fibonacci-Zahlen in C++

```
unsigned int fib (unsigned int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i){
    unsigned int a_old = a;  // F_i-2
    a = b;                   // F_i-1
    b += a_old;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

sehr schnell auch bei `fib(50)`

$$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$$

a

b

16. Rekursion 2

Bau eines Taschenrechners, Ströme, Formale Grammatiken, Extended Backus Naur Form (EBNF), Parsen von Ausdrücken

Motivation: Taschenrechner

Beispiel

Eingabe: `3 + 5`
Ausgabe: `8`

- Binäre Operatoren `+`, `-`, `*`, `/` und Zahlen

Motivation: Taschenrechner

Beispiel

Eingabe: `3 / 5`
Ausgabe: `0.6`

- Binäre Operatoren +, −, ∗, / und Zahlen
- Fliesskommaarithmetik

Motivation: Taschenrechner

Beispiel

Eingabe: `3 + 5 * 20`
Ausgabe: `103`

- Binäre Operatoren `+`, `-`, `*`, `/` und Zahlen
- Fliesskommaarithmetik
- Präzedenzen und Assoziativitäten wie in C++

Motivation: Taschenrechner

Beispiel

Eingabe: `(3 + 5) * 20`
Ausgabe: `160`

- Binäre Operatoren `+`, `-`, `*`, `/` und Zahlen
- Fliesskommaarithmetik
- Präzedenzen und Assoziativitäten wie in C++
- Klammerung

Motivation: Taschenrechner

Beispiel

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

Beispiel

Eingabe:

$$13 + ...$$

Analyse des Problems

Beispiel

Eingabe:

$$13 + 4 * ...$$

Analyse des Problems

Beispiel

Eingabe:

$$13 + 4 * (15 - ...$$

Analyse des Problems

Beispiel

Eingabe:

$$13 + 4 * (15 - 7 * ...$$

Analyse des Problems

Beispiel

Eingabe:

$$\underbrace{13 + 4 * (15 - 7*}\ 3) =$$

Muss gespeichert bleiben, damit jetzt ausgewertet werden kann!

Analyse des Problems

Beispiel

Ergebnis:

$$13 + 4*(15 - 21)$$

Analyse des Problems

Beispiel

Ergebnis:

$$13+4*(-6)$$

Analyse des Problems

Beispiel

Ergebnis:

$$13 + (-24)$$

Analyse des Problems

Beispiel

Ergebnis:

$$-11$$

Analyse des Problems

Beispiel

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Das “Verstehen” eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Analyse des Problems

Beispiel

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Das “Verstehen” eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Beispiel

Diese

Analyse des Problems

Beispiel

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Das “Verstehen” eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Beispiel

Diese Vorlesung

Analyse des Problems

Beispiel

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Das “Verstehen” eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Beispiel

Diese Vorlesung ist

Analyse des Problems

Beispiel

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Das “Verstehen” eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Beispiel

Diese Vorlesung ist insgesamt

Analyse des Problems

Beispiel

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

Das “Verstehen” eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Beispiel

Diese Vorlesung ist insgesamt recht

Analyse des Problems

Beispiel

Ausdruck:

$$13 + 4 * (15 - 7 * 3)$$

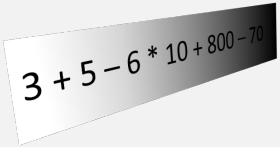
Das “Verstehen” eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Beispiel

Diese Vorlesung ist insgesamt recht rekursiv.

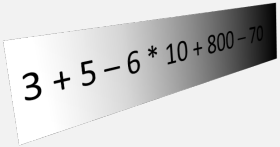
Als Vorbereitung: Ströme

Ein Programm verarbeitet Eingaben von einem konzeptuell unbegrenzten Eingabestrom.



Als Vorbereitung: Ströme

Ein Programm verarbeitet Eingaben von einem konzeptuell unbegrenzten Eingabestrom.



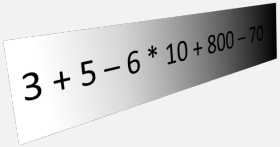
Bisher: Eingabestrom der Kommandozeile `std::cin`

```
while (std::cin >> op && op != '=') { ... }
```

Konsumiere `op` von `std::cin`,
Leseposition schreitet fort.

Als Vorbereitung: Ströme

Ein Programm verarbeitet Eingaben von einem konzeptuell unbegrenzten Eingabestrom.



Bisher: Eingabestrom der Kommandozeile `std::cin`

```
while (std::cin >> op && op != '=') { ... }
```

Konsumiere `op` von `std::cin`, Leseposition schreitet fort.

Wir wollen zukünftig aber auch von Dateien lesen können!

Beispiel: BSD 16-bit Checksum

```
#include <iostream>


int main () {

  char c;
  int checksum = 0;
  while (std::cin >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  std::cout << "checksum = " << std::hex << checksum << "\n";
}
```

Beispiel: BSD 16-bit Checksum

```
#include <iostream>


int main () {

  char c;
  int checksum = 0;
  while (std::cin >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  std::cout << "checksum = " << std::hex << checksum << "\n";
}
```

Eingabe: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Erfordert in der Konsole manuelles Ende der Eingabe

Beispiel: BSD 16-bit Checksum

```
#include <iostream>


int main () {

  char c;
  int checksum = 0;
  while (std::cin >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  std::cout << "checksum = " << std::hex << checksum << "\n";
}
```

Eingabe: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Ausgabe: 67fd

Beispiel: BSD 16-bit Checksum mit Datei

```
#include <iostream>
#include <fstream>

int main () {
  std::ifstream fileStream ("loremispum.txt");
  char c;
  int checksum = 0;
  while (fileStream >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  std::cout << "checksum = " << std::hex << checksum << "\n";
}
```

Beispiel: BSD 16-bit Checksum mit Datei

```
#include <iostream>
#include <fstream>

int main () {
  std::ifstream fileStream ("loremispum.txt");
  char c;
  int checksum = 0;
  while (fileStream >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  std::cout << "checksum = " << std::hex << checksum << "\n";
}
```

Gibt am Dateiende `false` zurück.

Beispiel: BSD 16-bit Checksum mit Datei

```
#include <iostream>
#include <fstream>

int main () {
  std::ifstream fileStream ("loremispum.txt");
  char c;
  int checksum = 0;
  while (fileStream >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  std::cout << "checksum = " << std::hex << checksum << "\n";
}
```

Ausgabe: 67fd

Gibt am Dateiende `false` zurück.

Beispiel: BSD 16-bit Checksum

Wiederverwendung gemeinsam genutzter Funktionalität?

Beispiel: BSD 16-bit Checksum

Wiederverwendung gemeinsam genutzter Funktionalität?

Richtig: mit einer Funktion. Aber wie?

Beispiel: BSD 16-bit Checksum generisch!

```
#include <iostream>
#include <fstream>

int checksum (std::istream& is)
{
  char c;
  int checksum = 0;
  while (is >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  return checksum;
}
```

Beispiel: BSD 16-bit Checksum generisch!

```
#include <iostream>
#include <fstream>

int checksum (std::istream& is)
{
  char c;
  int checksum = 0;
  while (is >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  return checksum;
}
```

Referenz nötig: wir verändern den Strom!

Gleiches Recht für alle!

```
#include <iostream>
#include <fstream>

int checksum (std::istream& is) { ... }

int main () {
  std::ifstream fileStream("loremipsum.txt");

  if (checksum (fileStream) == checksum (std::cin))
        std::cout << "checksums match.\n";
  else
        std::cout << "checksums differ.\n";
}
```

Gleiches Recht für alle!

```
#include <iostream>
#include <fstream>

int checksum (std::istream& is) { ... }

int main () {
  std::ifstream fileStream("loremipsum.txt");

  if (checksum (fileStream) == checksum (std::cin))
        std::cout << "checksums match.\n";
  else
        std::cout << "checksums differ.\n";
}
```

Eingabe: Lorem Yps mit Gimmick

Gleiches Recht für alle!

```
#include <iostream>
#include <fstream>

int checksum (std::istream& is) { ... }

int main () {
  std::ifstream fileStream("loremipsum.txt");

  if (checksum (fileStream) == checksum (std::cin))
        std::cout << "checksums match.\n";
  else
        std::cout << "checksums differ.\n";
}
```

Eingabe: Lorem Yps mit Gimmick
Ausgabe: checksums differ

Warum geht das ?

- `std::cin` ist eine Variable vom Typ `std::istream`. Sie repräsentiert einen Eingabestrom.

Warum geht das ?

- `std::cin` ist eine Variable vom Typ `std::istream`. Sie repräsentiert einen Eingabestrom.
- Unsere Variable `fileStream` ist vom Typ `std::ifstream`. Sie repräsentiert einen Eingabestrom auf einer Datei.

Warum geht das ?

- `std::cin` ist eine Variable vom Typ `std::istream`. Sie repräsentiert einen Eingabestrom.
- Unsere Variable `fileStream` ist vom Typ `std::ifstream`. Sie repräsentiert einen Eingabestrom auf einer Datei.
- Ein `std::ifstream` *ist auch ein* `std::istream`, kann nur etwas mehr.

Warum geht das ?

- `std::cin` ist eine Variable vom Typ `std::istream`. Sie repräsentiert einen Eingabestrom.
- Unsere Variable `fileStream` ist vom Typ `std::ifstream`. Sie repräsentiert einen Eingabestrom auf einer Datei.
- Ein `std::ifstream` *ist auch ein* `std::istream`, kann nur etwas mehr.
- Somit kann `fileStream` überall dort verwendet werden, wo ein `std::istream` verlangt ist.

Nochmal gleiches Recht für alle!

```
#include <iostream>
#include <fstream>
#include <sstream>

int checksum (std::istream& is) { ... }

int main () {
  std::ifstream fileStream ("loremipsum.txt");
  std::stringstream stringStream ("Lorem Yps mit Gimmick");

  if (checksum (fileStream) == checksum (stringStream))
       std::cout << "checksums match.\n";
  else
       std::cout << "checksums differ.\n";
}
```

Nochmal gleiches Recht für alle!

```
#include <iostream>
#include <fstream>
#include <sstream>

int checksum (std::istream& is) { ... }

int main () {
  std::ifstream fileStream ("loremipsum.txt");
  std::stringstream stringStream ("Lorem Yps mit Gimmick");

  if (checksum (fileStream) == checksum (stringStream))
       std::cout << "checksums match.\n";
  else
       std::cout << "checksums differ.\n";
}
```

Eingabe aus `stringStream`

Nochmal gleiches Recht für alle!

```
#include <iostream>
#include <fstream>
#include <sstream>

int checksum (std::istream& is) { ... }

int main () {
  std::ifstream fileStream ("loremipsum.txt");
  std::stringstream stringStream ("Lorem Yps mit Gimmick");

  if (checksum (fileStream) == checksum (stringStream))
       std::cout << "checksums match.\n";
  else
       std::cout << "checksums differ.\n";
}
```

Eingabe aus `stringStream`
Ausgabe: checksums differ

Zurück zu den Ausdrücken

$$13 + 4 * (15 - 7 * 3)$$

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.

Zurück zu den Ausdrücken

$$13 + 4 * (15 - 7 * 3)$$

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.

Zurück zu den Ausdrücken

$$13 + 4 * (15 - 7 * 3)$$

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.

Zurück zu den Ausdrücken

$$13 + 4 * (15 - 7 * 3)$$

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 Σ
- Sätze: endlichen Folgen von Symbolen Σ^*

Formale Grammatiken

- Alphabet: endliche Menge von Symbolen Σ
- Sätze: endlichen Folgen von Symbolen Σ^*

Eine formale Grammatik definiert, welche Sätze gültig sind.

Berge

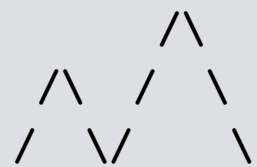
- Alphabet: $\{/, \backslash\}$
- Berge: $\mathcal{M} \subset \{/, \backslash\}^*$ (gültige Sätze)

$m =$ //\/\\

Berge

- Alphabet: $\{/, \backslash\}$
- Berge: $\mathcal{M} \subset \{/, \backslash\}^*$ (gültige Sätze)

$m' =$ //\\//\\\\



Falsche Berge

- Alphabet: $\{/, \backslash\}$
- Berge: $\mathcal{M} \subset \{/, \backslash\}^*$ (gültige Sätze)

$m'' =$ //\\\

Falsche Berge

- Alphabet: $\{/, \backslash\}$
- Berge: $\mathcal{M} \subset \{/, \backslash\}^*$ (gültige Sätze)

$m'' = //\backslash\backslash\backslash \notin \mathcal{M}$

Beide Seiten müssen gleiche Starthöhe haben.

Falsche Berge

- Alphabet: $\{/, \backslash\}$
- Berge: $\mathcal{M} \subset \{/, \backslash\}^*$ (gültige Sätze)

$m''' =$ /\\//\

Falsche Berge

- Alphabet: $\{/, \backslash\}$
- Berge: $\mathcal{M} \subset \{/, \backslash\}^*$ (gültige Sätze)

$m''' = /\backslash\backslash//\backslash \notin \mathcal{M}$

Ein Berg darf nicht unter seine Starthöhe fallen.

Berge in Backus-Naur-Form (BNF)

```
berg = "/\"  |  "/"  berg "\"  |  berg  berg.
```

Berge in Backus-Naur-Form (BNF)

berg = "/\" | "/" berg "\" | berg berg.

Mögliche Berge

Berge in Backus-Naur-Form (BNF)

```
berg = "/\" | "/" berg "\" | berg berg.
```

Mögliche Berge

1. /\

Berge in Backus-Naur-Form (BNF)

```
berg = "/\" | "/" berg "\" | berg berg.
```

Mögliche Berge

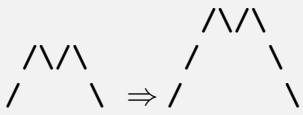
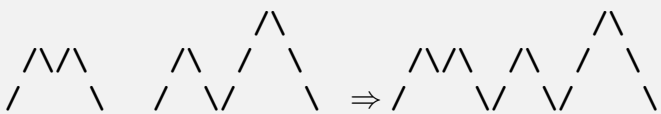
1. `/\`
2. `/\/\` ⇒

```
      /\/\
/\/\ /    \
/    \    \
```

Berge in Backus-Naur-Form (BNF)

berg = "/\" | "/" berg "\" | berg berg.

Mögliche Berge

1. 
2. 
3. 

Berge in Backus-Naur-Form (BNF)

```
berg = "/\"  |  "/"  berg "\"  |  berg  berg.
```

Regel

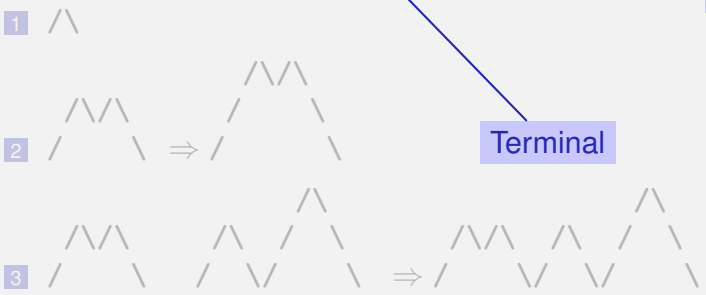
Alternativen

Berge in Backus-Naur-Form (BNF)

berg = "/\" | "/" berg "\" | berg berg.

Nichtterminal

Terminal



Ausdrücke

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

Ausdrücke

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

- Zahl

Ausdrücke

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

- Zahl , (?)

Ausdrücke

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

- Zahl , (?)
  -Zahl, -(?)

Ausdrücke

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

- Zahl , (?)
  -Zahl, -(?)
- ? * ?, ? * ? / ?, ...

Ausdrücke

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

- Zahl , `(` ? `)`
  `-`Zahl, `-(` ? `)`
- ? `*` ?, ? `*` ? `/` ?, ...
- ? `-` ?, ? `+` ?, ...

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

- Zahl , (?)
  -Zahl, -(?)
- ? * ?, ? * ? / ?, ...
- ? - ?, ? + ?, ...

Faktor

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

- Zahl , (?)
  -Zahl, -(?)
- Faktor * Faktor,
  Faktor * Faktor / Faktor , ...
- ? - ?, ? + ?, ...

Faktor

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

- Zahl , (?)
  -Zahl, -(?)
- Faktor * Faktor,
  Faktor * Faktor / Faktor , ...
- ? - ?, ? + ?, ...

Faktor

Term

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

- Zahl , (?)
  -Zahl, -(?)
- Faktor ∗ Faktor, Faktor
  Faktor ∗ Faktor / Faktor , ...
- ? - ?, ? + ?, ...

Faktor

Term

Ausdrücke

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

- Zahl , (?)
  -Zahl, -(?)

  Faktor
- Faktor ∗ Faktor, Faktor
  Faktor ∗ Faktor / Faktor , ...

  Term
- Term + Term,
  Term − Term, ...

Ausdrücke

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

- Zahl , (?)
  -Zahl, -(?)
- Faktor ∗ Faktor, Faktor
  Faktor ∗ Faktor / Faktor , ...
- Term + Term,
  Term − Term, ...

Faktor

Term

Ausdruck

Ausdrücke

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer BNF?

- Zahl , (?)
  −Zahl, −(?)
- Faktor ∗ 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 BNF?

- Zahl , (Ausdruck)
  −Zahl, −(Ausdruck)
- Faktor ∗ Faktor, Faktor
  Faktor ∗ Faktor / Faktor , ...
- Term + Term, Term
  Term − Term, ...

Faktor

Term

Ausdruck

Die BNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck oder
- ein negierter Faktor.

```
factor     = unsigned_number
           | "(" expression ")"
           | "-" factor.
```

Die BNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck oder
- ein negierter Faktor.

```
factor     = unsigned_number
           | "(" expression ")"
           | "-" factor.
```

Die BNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck oder
- ein negierter Faktor.

```
factor    = unsigned_number
          | "(" expression ")"
          | "-" factor.
```

Die BNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck oder
- ein negierter Faktor.

```
factor    = unsigned_number
          | "(" expression ")"
          | "-" factor.
```

Die BNF für Ausdrücke

Ein Term ist

- Faktor,
- Faktor $*$ Faktor, Faktor / Faktor,
- Faktor $*$ Faktor $*$ Faktor, Faktor / Faktor $*$ Faktor, ...
- ...

Wir brauchen Repetition!

Die BNF für Ausdrücke

Ein Term ist

- Faktor,
- Faktor $*$ Faktor, Faktor / Faktor,
- Faktor $*$ Faktor $*$ Faktor, Faktor / Faktor $*$ Faktor, ...
- ...

Wir brauchen Repetition!

Die BNF für Ausdrücke

Ein Term ist

- Faktor,
- Faktor $*$ Faktor, Faktor / Faktor,
- Faktor $*$ Faktor $*$ Faktor, Faktor / Faktor $*$ Faktor, ...
- ...

Wir brauchen Repetition!

EBNF

Extended Backus Naur Form: Erweiterung der BNF um

- Option `[]` und
- Optionale Repetition `{}`

```
term = factor { "*" factor | "/" factor }.
```

Die EBNF für Ausdrücke

```
factor     = unsigned_number
           | "(" expression ")"
           | "−" factor.

term       = factor { "∗" factor | "/" factor }.

expression = term { "+" term | "−" term }.
```

Parsen

- **Parsen:** Feststellen, ob ein Satz nach der (E)BNF gültig ist.

Parsen

- **Parsen:** Feststellen, ob ein Satz nach der (E)BNF gültig ist.
- **Parser:** Programm zum Parsen

Parsen

- **Parsen:** Feststellen, ob ein Satz nach der (E)BNF gültig ist.
- **Parser:** Programm zum Parsen
- **Praktisch:** Aus der (E)BNF kann (fast) 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 }.
```

Funktionen (Parser)

Ausdruck wird aus einem Eingabestrom gelesen.

```
// POST: returns true if and only if is = factor ...
//       and in this case extracts factor from is
bool factor (std::istream& is);

// POST: returns true if and only if is = term ...,
//       and in this case extracts all factors from is
bool term (std::istream& is);

// POST: returns true if and only if is = expression ...,
//       and in this case extracts all terms from is
bool expression (std::istream& is);
```

Funktionen (Parser mit Auswertung)

Ausdruck wird aus einem Eingabestrom gelesen.

```
// POST: extracts a factor from is
//       and returns its value
double factor (std::istream& is);

// POST: extracts a term from is
//       and returns its value
double term (std::istream& is);

// POST: extracts an expression from is
//       and returns its value
double expression (std::istream& is);
```

Vorausschau von einem Zeichen...

...um jeweils die richtige Alternative zu finden.

```
// POST: leading whitespace characters are extracted
//       from is, and the first non-whitespace character
//       is returned (0 if there is no such character)
char lookahead (std::istream& is)
{
  if (is.eof())
    return 0;
  is >> std::ws;          // skip whitespaces
  if (is.eof())
    return 0;             // end of stream
  return is.peek();     // next character in is
}
```

Rosinenpickerei

. . . um jeweils nur das gewünschte Zeichen zu extrahieren.

```
// POST: if ch matches the next lookahead then consume it
//       and return true; return false otherwise
bool consume (std::istream& is, char ch)
{
    if (lookahead(is) == ch){
        is >> ch;
        return true;
    }
    return false;
}
```

Faktoren auswerten

```
double factor (std::istream& is)
{
  double v;
  if (consume(is, '('){
    v = expression (is);
    consume(is, ')');
  } else if (consume(is, '-'))
    v = -factor (is);
  else
    is >> v;
  return v;
}
```

```
factor = "(" expression ")"
       | "-" factor
       | unsigned_number.
```

Terme auswerten

```
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;
  }
}
```

```
term = factor { "*" factor | "/" factor }
```

Ausdrücke auswerten

```
double expression (std::istream& is)
{
  double value = term(is);
  while(true){
    if (consume(is, '+'))
      value += term (is);
    else if (consume(is, '-'))
      value -= term(is)
    else
      return value;
  }
}
```

```
expression = term { "+" term | "-" term }
```

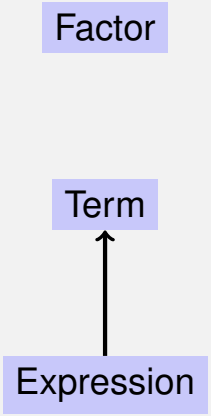
Rekursion!

Factor

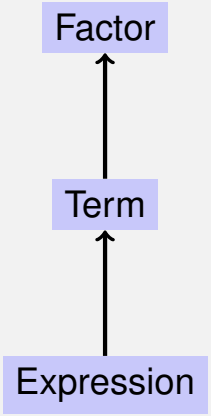
Term

Expression

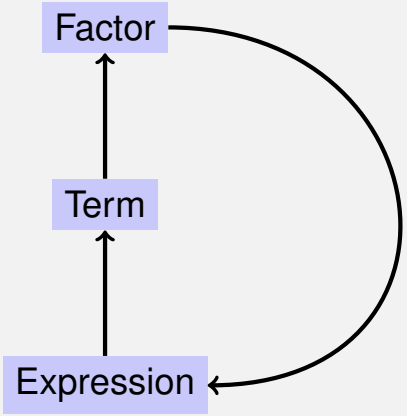
Rekursion!



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
```

BNF — Und es funktioniert nicht!

BNF (`calculator_r.cpp`, Auswertung von rechts nach links):

```
factor     = unsigned_number
           | "(" expression ")"
           | "-" factor.

term       = factor | factor "*" term | factor "/" term.

expression = term | term "+" expression | term "-" expression.
```

```
std::stringstream input ("1-2-3");
std::cout << expression (input) << "\n"; // 2
```

Analyse: Repetition vs. Rekursion

Vereinfachung: Summe / Differenz von Zahlen

Beispiele

$$3$$

Analyse: Repetition vs. Rekursion

Vereinfachung: Summe / Differenz von Zahlen

Beispiele

$$3, \quad 3 - 5$$

Analyse: Repetition vs. Rekursion

Vereinfachung: Summe / Differenz von Zahlen

Beispiele

$$3, \quad 3-5, \quad 3-7-1$$

Analyse: Repetition vs. Rekursion

Vereinfachung: Summe / Differenz von Zahlen

Beispiele

$$3, \quad 3-5, \quad 3-7-1$$

EBNF:

```
sum = value {"−" value | "+" value}.
```

Analyse: Repetition vs. Rekursion

Vereinfachung: Summe / Differenz von Zahlen

Beispiele

$$3, \quad 3-5, \quad 3-7-1$$

EBNF:

```
sum = value {"−" value | "+" value}.
```

BNF:

```
sum = value | value "−" sum | value "+" sum.
```

Analyse: Repetition vs. Rekursion

Vereinfachung: Summe / Differenz von Zahlen

Beispiele

$$3, \quad 3-5, \quad 3-7-1$$

EBNF:

```
sum = value {"−" value | "+" value}.
```

BNF:

```
sum = value | value "−" sum | value "+" sum.
```

Die beiden Gramatiken erlauben dieselben Ausdrücke.

value

```
double value (std::istream& is){
    double val;
    is >> val;
    return val;
}
```

EBNF Variante

```
// sum = value {"-" value | "+" value}.
double sum(std::istream& is) {
    double v = value(is);
    while(true){
        if (consume(is, '-'))
            v -= value(is);
        else if (consume(is, '+'))
            v += value(is);
        else
            return v;
    }
}
```

Wir testen: EBNF Variante

- Eingabe: `1-2`
  Ausgabe:

Wir testen: EBNF Variante

- Eingabe: `1-2`
  Ausgabe: `-1`

Wir testen: EBNF Variante

- Eingabe: `1-2`
  Ausgabe: `-1` ✓

Wir testen: EBNF Variante

- Eingabe: `1-2-3`
  Ausgabe:

Wir testen: EBNF Variante

- Eingabe: `1-2-3`
  Ausgabe: `-4`

Wir testen: EBNF Variante

- Eingabe: `1-2-3`
  Ausgabe: `-4` ✓

BNF Variante

```
// sum = value | value "-" sum | value "+" sum.
double sum(std::istream& is){
    double v = value(is);
    if (consume(is, '-'))
        return v - sum(is);
    else if(consume(is, '+'))
        return v + sum(is);
    return v;
}
```

Wir testen: BNF Variante

- Eingabe: `1-2`
  Ausgabe:

Wir testen: BNF Variante

- Eingabe: `1-2`
  Ausgabe: `-1`

Wir testen: BNF Variante

- Eingabe: `1-2`
  Ausgabe: `-1` ✓

Wir testen: BNF Variante

- Eingabe: `1-2-3`
  Ausgabe:

Wir testen: BNF Variante

- Eingabe: `1-2-3`
  Ausgabe: `2`

Wir testen: BNF Variante

- Eingabe: `1-2-3`

  Ausgabe: `2` 

Wir testen

 Ist die BNF falsch ?

Wir testen


Ist die BNF falsch ?

```
sum = value
      | value "—" sum
      | value "+" sum.
```

Wir testen

 Ist die BNF falsch ?

```
sum = value
      | value "−" sum
      | value "+" sum.
```

- Nein, denn sie spricht nur über die Gültigkeit von Ausdrücken, nicht über deren Werte!

Wir testen

 Ist die BNF falsch ?

```
sum = value
      | value "−" sum
      | value "+" sum.
```

- Nein, denn sie spricht nur über die Gültigkeit von Ausdrücken, nicht über deren Werte!
- Die Auswertung haben wir naiv “obendrauf” gesetzt.

Dem Problem auf den Grund gehen

```
double sum (std::istream& is){
   double v = value (is);
   if (consume (is,'-'))
      v -= sum (is);
   else if (consume (is,'+'))
    v += sum(is);
   return v;
}
...
std::stringstream input ("1-2-3");
std::cout << sum (input) << "\n"; // 4
```

"1 - 2 - 3"

Dem Problem auf den Grund gehen

```
double sum (std::istream& is){
   double v = value (is);
   if (consume (is,'-'))
      v -= sum (is);
   else if (consume (is,'+'))
    v += sum(is);
   return v;
}
...
std::stringstream input ("1-2-3");
std::cout << sum (input) << "\n"; // 4
```

1 - "2 - 3"

"1 - 2 - 3"

Dem Problem auf den Grund gehen

```
double sum (std::istream& is){
   double v = value (is);
   if (consume (is,'-'))
      v -= sum (is);
   else if (consume (is,'+'))
    v += sum(is);
   return v;
}
...
std::stringstream input ("1-2-3");
std::cout << sum (input) << "\n"; // 4
```

2 - "3"

1 - "2 - 3"

"1 - 2 - 3"

Dem Problem auf den Grund gehen

```
double sum (std::istream& is){
   double v = value (is);
   if (consume (is,'-'))
      v -= sum (is);
   else if (consume (is,'+'))
    v += sum(is);
   return v;
}
...
std::stringstream input ("1-2-3");
std::cout << sum (input) << "\n"; // 4
```

3

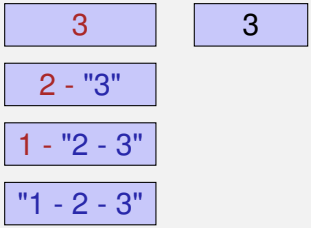
2 - "3"

1 - "2 - 3"

"1 - 2 - 3"

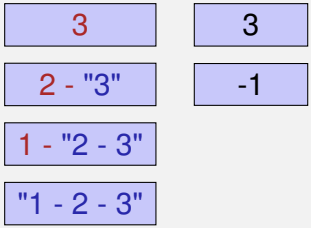
Dem Problem auf den Grund gehen

```
double sum (std::istream& is){
   double v = value (is);
   if (consume (is,'-'))
      v -= sum (is);
   else if (consume (is,'+'))
    v += sum(is);
   return v;
}
...
std::stringstream input ("1-2-3");
std::cout << sum (input) << "\n"; // 4
```



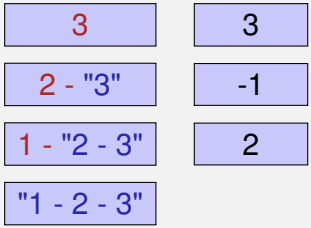
Dem Problem auf den Grund gehen

```
double sum (std::istream& is){
   double v = value (is);
   if (consume (is,'-'))
      v -= sum (is);
   else if (consume (is,'+'))
    v += sum(is);
   return v;
}
...
std::stringstream input ("1-2-3");
std::cout << sum (input) << "\n"; // 4
```



Dem Problem auf den Grund gehen

```
double sum (std::istream& is){
   double v = value (is);
   if (consume (is,'-'))
      v -= sum (is);
   else if (consume (is,'+'))
    v += sum(is);
   return v;
}
...
std::stringstream input ("1-2-3");
std::cout << sum (input) << "\n"; // 4
```



Dem Problem auf den Grund gehen

```
double sum (std::istream& is){
   double v = value (is);
   if (consume (is,'-'))
      v -= sum (is);
   else if (consume (is,'+'))
    v += sum(is);
   return v;
}
...
std::stringstream input ("1-2-3");
std::cout << sum (input) << "\n"; // 4
```

3	3
2 - "3"	-1
1 - "2 - 3"	2
"1 - 2 - 3"	2

Was ist denn falsch gelaufen?

Die BNF

- spricht offiziell zwar nicht über Werte,

Was ist denn falsch gelaufen?

Die BNF

- spricht offiziell zwar nicht über Werte,
- legt uns aber trotzdem die falsche Klammerung (von rechts nach links) nahe.

Was ist denn falsch gelaufen?

Die BNF

- spricht offiziell zwar nicht über Werte,
- legt uns aber trotzdem die falsche Klammerung (von rechts nach links) nahe.

```
sum = value | value "−" sum | value "+" sum.
```

führt sehr natürlich zu

$1 - 2 - 3 = 1 - (2 - 3)$

Eine Lösung: Linksrekursion

```
sum = value | sum "−" value | sum "+" value.
```

Implementationsmuster von vorher funktioniert nicht mehr.
Linksrekursion muss wieder zu Rechtsrekursion aufgelöst werden.

Das sähe dann so aus:

```
sum = value | value s.
s = "−" sum | "+" sum.
```

Eine Lösung: Linksrekursion

```
sum  = value | sum "−" value | sum "+" value.
```

Implementationsmuster von vorher funktioniert nicht mehr.
Linksrekursion muss wieder zu Rechtsrekursion aufgelöst werden.

Das sähe dann so aus:

```
sum  = value | value s.
s = "−" sum | "+" sum.
```

Eine Lösung: Linksrekursion

```
sum  = value | sum "−" value | sum "+" value.
```

Implementationsmuster von vorher funktioniert nicht mehr.
Linksrekursion muss wieder zu Rechtsrekursion aufgelöst werden.

Das sähe dann so aus:

```
sum  = value | value s.
s = "−" sum | "+" sum.
```

17. Structs und Klassen I

Rationale Zahlen, Struct-Definition, Funktions- und Operatorüberladung, Const-Referenzen, Datenkapselung

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

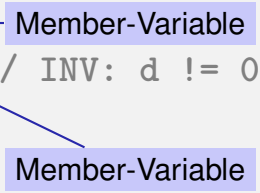
```
struct rational {
  int n;
  int d; // INV: d != 0
};
```

Member-Variable (**n**umerator)

Member-Variable (**d**enominator)

Ein erstes Struct

```
struct rational {
  int n;
  int d; // INV: d != 0
};
```



- `struct` definiert einen neuen *Typ*

Ein erstes Struct

```
struct rational {
  int n;
  int d; // INV: d != 0
};
```

Member-Variable

Member-Variable

- `struct` definiert einen neuen *Typ*
- Formaler Wertebereich: *kartesisches Produkt* der Wertebereiche existierender Typen

Ein erstes Struct

```
struct rational {
  int n;
  int d; // INV: d != 0
};
```

Member-Variable

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}; ⟵

rational u = t;

t = u;

rational v = add (u,t);
```

Memberweise Initialisierung:
`t.n = 1, t.d = 5`

Structs: Initialisierung und Zuweisung

```
rational s;

rational t = {1,5};

rational u = t;
t = u;

rational v = add (u,t);
```

⟵ Memberweise Kopie

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 *Operator-Überladung*.

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);
std::cout << sq (1.414);
std::cout << pow (2);
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);
std::cout << sq (1.414);
std::cout << pow (2);
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);
std::cout << sq (1.414);
std::cout << pow (2);
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);
std::cout << pow (2);
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);
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);
```

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;
}
```

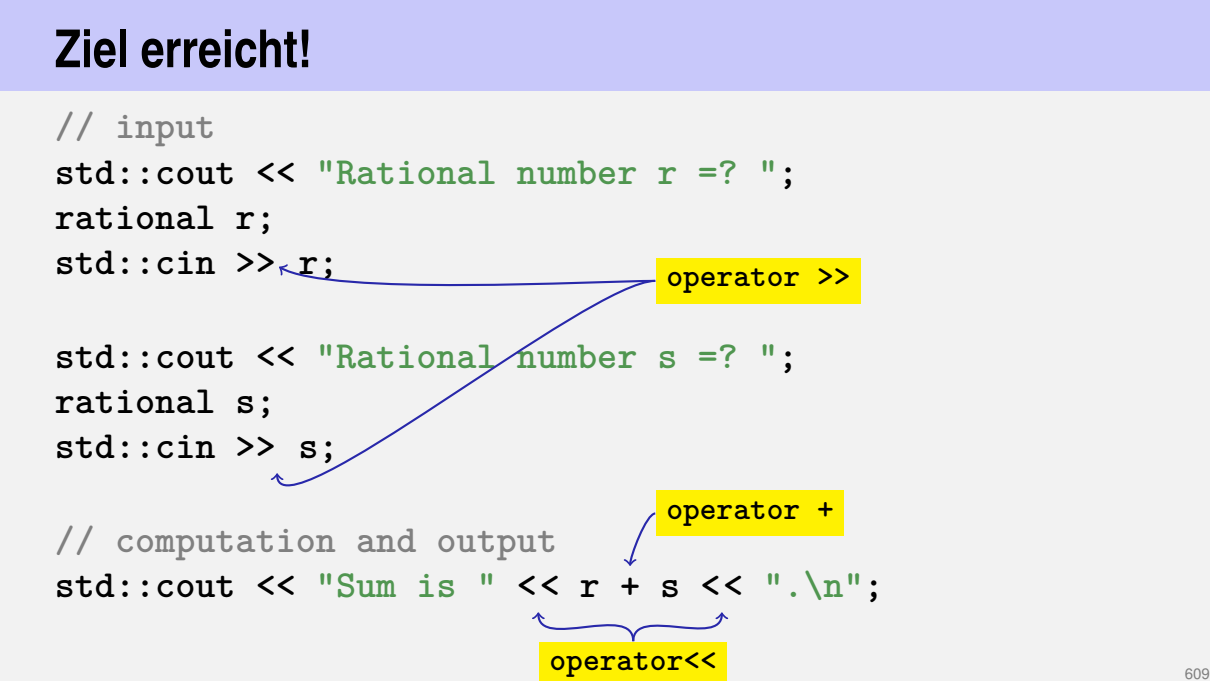
liest `r` aus dem Eingabestrom
und gibt diesen als L-Wert zurück.

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";
```



Zur Erinnerung: Grosse Objekte ...

```
struct SimulatedCPU {
    unsigned int pc;
    int stack[16];
    unsigned int stackPosition;
    unsigned int memory[65536];
};
```

Zur Erinnerung: Grosse Objekte ...

```
struct SimulatedCPU {
    unsigned int pc;
    int stack[16];
    unsigned int stackPosition;
    unsigned int memory[65536];
};

void outputState (SimulatedCPU p) {
    std::cout << "pc=" << p.pc;
    std::cout << ", stack: ";
    for (unsigned int i = p.stackPosition; i != 0; --i)
        std::cout << p.stack[i-1];
}
```

Zur Erinnerung: Grosse Objekte ...

```
struct SimulatedCPU {
    unsigned int pc;
    int stack[16];
    unsigned int stackPosition;
    unsigned int memory[65536];
};
```

Call by value: mehr als 256k werden kopiert!

```
void outputState (SimulatedCPU p) {
    std::cout << "pc=" << p.pc;
    std::cout << ", stack: ";
    for (unsigned int i = p.stackPosition; i != 0; --i)
        std::cout << p.stack[i-1];
}
```

... übergibt man als Const-Referenz

```
struct SimulatedCPU {
    unsigned int pc;
    int stack[16];
    unsigned int stackPosition;
    unsigned int memory[65536];
};
```

Call by reference: nur eine Adresse wird kopiert.

```
void outputState (const SimulatedCPU& p) {
    std::cout << "pc=" << p.pc;
    std::cout << ", stack: ";
    for (int i = p.stackPosition; i != 0; --i)
        std::cout << p.stack[i-1];
}
```

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 $(\frac{3}{5} \to 0.6)$

Der Kunde ist zufrieden

. . . und programmiert fleissig mit `rational`.

- Ausgabe als `double`-Wert $(\frac{3}{5} \to 0.6)$

```
// POST: double approximation of r
double to_double (rational r)
{
  double result = r.n;
  return result / r.d;
}
```

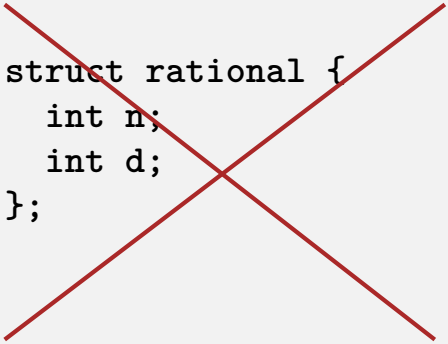
Der Kunde will mehr

“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;
};
```

$\Rightarrow$

```
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.

$\Rightarrow$ 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),...
```

Klassen

- 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

18. Klassen

Klassen, Memberfunktionen, Konstruktoren, Stapel, verkettete Liste, dynamischer Speicher, Copy-Konstruktor, Zuweisungsoperator, Destruktor, Konzept Dynamischer Datentyp

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 . . .

. . . und wir auch nicht
(kein `operator+`,. . .)

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
```

Memberfunktionen: Deklaration

```
class rational {
public:
   // POST: return value is the numerator of *this
   int numerator () const {
     return n;
   }
   // POST: return value is the denominator of *this
   int denominator () const {
     return d;
   }
private:
   int n;
   int d; // INV: d!= 0
};
```

Memberfunktionen: Deklaration

öffentlicher Bereich

```
class rational {
public:
  // POST: return value is the numerator of *this
  int numerator () const {
    return n;
  }
  // POST: return value is the denominator of *this
  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
  int numerator () const {
    return n;
  }
  // POST: return value is the denominator of *this
  int denominator () const {
    return d;
  }
private:
   int n;
   int d; // INV: d!= 0
};
```

öffentlicher Bereich

Memberfunktion

Memberfunktionen: Deklaration

```
class rational {
public:
  // POST: return value is the numerator of *this
  int numerator () const {
    return n;
  }
  // POST: return value is the denominator of *this
  int denominator () const {
    return d;
  }
private:
  int n;
  int d; // INV: d!= 0
};
```

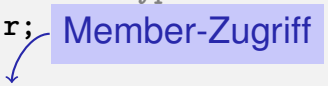
öffentlicher Bereich

Memberfunktion

Memberfunktionen haben Zugriff auf private Daten

Memberfunktionen: Aufruf

```
// Definition des Typs
class rational {
    ...
};
...
// Variable des Typs
rational r;

int n = r.numerator();   // Zaehler
int d = r.denominator(); // Nenner
```



Memberfunktionen: Definition

```
// POST: returns numerator of *this
int numerator () const
{
  return n;
}
```

Memberfunktionen: Definition ???

```
// POST: returns numerator of *this
int numerator ()  const
{
  return n;
}
```

Memberfunktionen: Definition

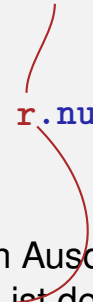
```
// POST: returns numerator of *this
int numerator () const
{
  return n;                          r.numerator()
}
```

- Eine Memberfunktion wird für einen Ausdruck der Klasse aufgerufen.

Memberfunktionen: Definition

```
// POST: returns numerator of *this
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.

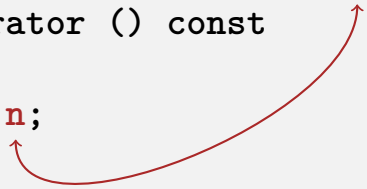
Memberfunktionen: Definition

```
// POST: returns numerator of *this
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 `*this`

Memberfunktionen: Definition

```
// POST: returns  numerator of *this
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 `*this`
- `n` ist Abkürzung für `(*this).n`

This rational vs. dieser Bruch

```
class rational {
    int n;
    ...

    int numerator () const
    {
        return .n;
    }
};

rational r;
...
std::cout << r.numerator();
```

This rational vs. dieser Bruch

```
class rational {
    int n;
    ...

    int numerator () const
    {
        return (*this).n;
    }
};

rational r;
...
std::cout << r.numerator();
```

This rational vs. dieser Bruch

So würde es aussehen...

```
class rational {
    int n;
    ...

    int numerator () const
    {
        return (*this).n;
    }
};

rational r;
...
std::cout << r.numerator();
```

This rational vs. dieser Bruch

So würde es aussehen...

```
class rational {
    int n;
    ...

    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;
    ...

    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;
    ...

    int numerator () const
    {
        return n;
    }
    ....
};
```

- Keine Trennung zwischen Deklaration und Definition (schlecht für Bibliotheken)

```
class rational {
    int n;
    ...

    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)
    {
        assert (den != 0);
    }
...
};
...
rational r (2,3); // r = 2/3
```

Initialisierung der Membervariablen

Funktionsrumpf.

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 ()
      : n (0), d (1)
    {}
...
};
...
rational r;     // r = 0
```

Leere Argumentliste

Der Default-Konstruktor

```
class rational
{
public:
     ...
     rational ()
        : n (0), d (1)
     {}
...
};
...
rational r;     // r = 0
```

Leere Argumentliste

$\Rightarrow$ Es gibt keine uninitialisierten Variablen vom Typ rational mehr!

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`):

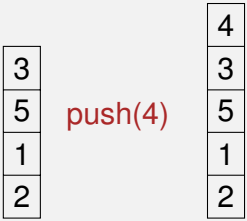
```
class rational {
public:
   // POST: returns numerator of *this
   int numerator () const;
   ...
private:
  // none of my business
};
```

- Wir legen uns auf Zähler-/Nennertyp `int` fest.
- Lösung: Nicht nur Daten, auch Typen kapseln (Handout).

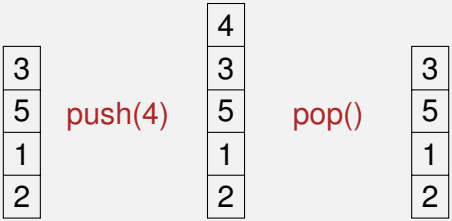
Motivation: Stapel

Motivation: Stapel

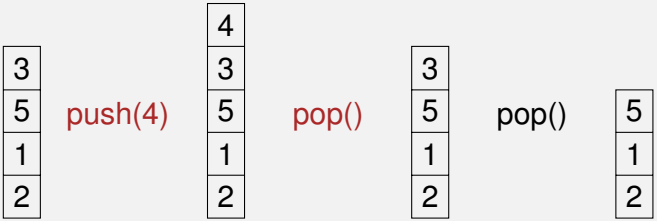
Motivation: Stapel



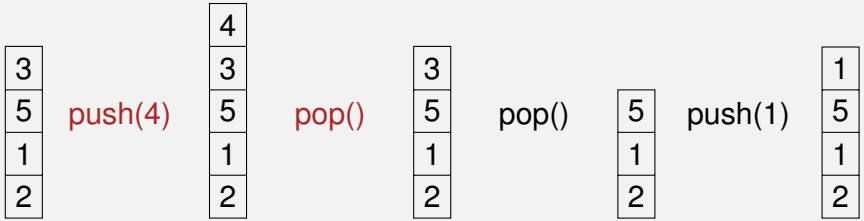
Motivation: Stapel



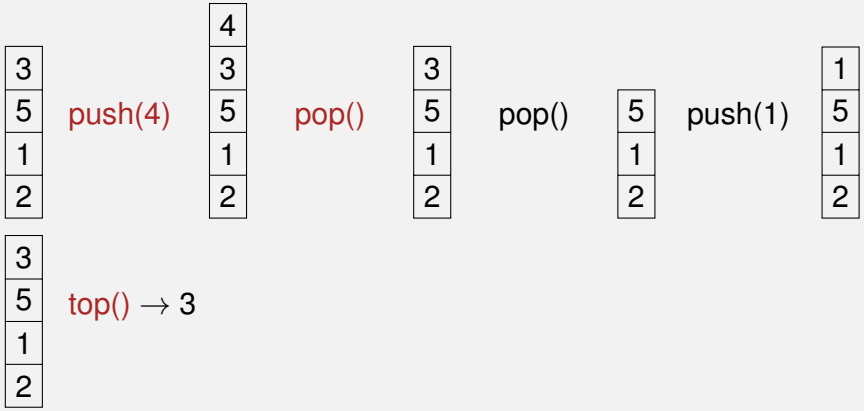
Motivation: Stapel



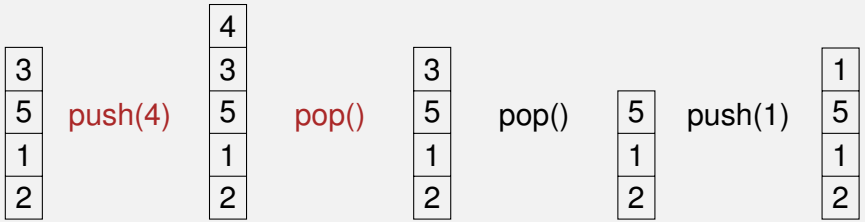
Motivation: Stapel



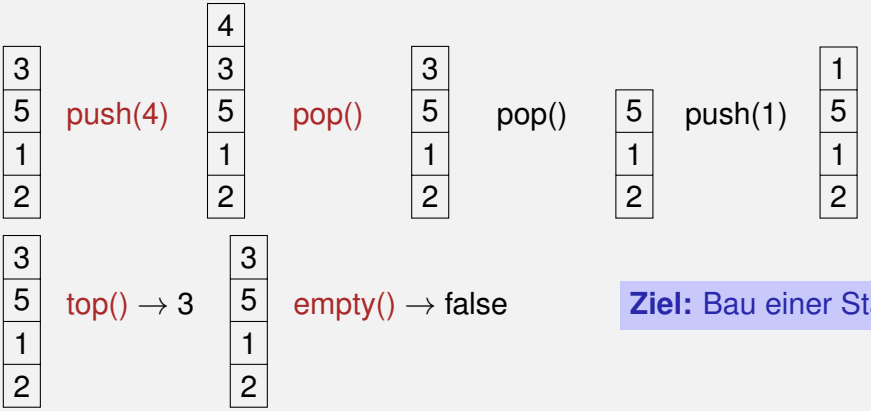
Motivation: Stapel



Motivation: Stapel (`push, pop, top, empty`)

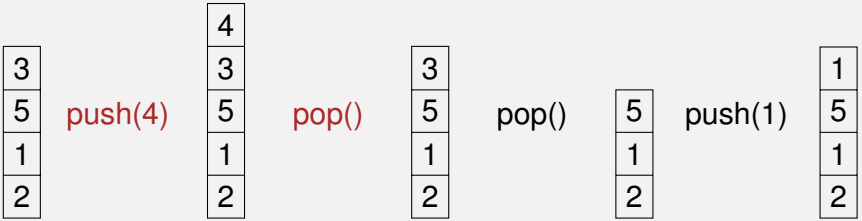


Motivation: Stapel (`push, pop, top, empty`)



Ziel: Bau einer Stapel-Klasse!

Motivation: Stapel (`push, pop, top, empty`)



Ziel: Bau einer Stapel-Klasse!

Frage: wie schaffen wir bei push neuen Platz auf dem Stapel?

Wir brauchen einen neuen Container!

Unser Haupt-Container bisher: Array (`T[]`)

Wir brauchen einen neuen Container!

Unser Haupt-Container bisher: Array (`T[]`)

- Zusammenhängender Speicherbereich, wahlfreier Zugriff (auf i-tes Element)

Wir brauchen einen neuen Container!

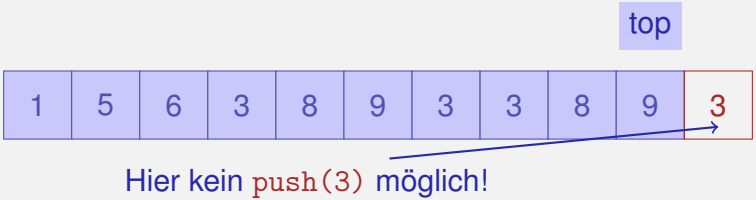
Unser Haupt-Container bisher: Array (`T[]`)

- Zusammenhängender Speicherbereich, wahlfreier Zugriff (auf i-tes Element)
- Simulation eines Stapels durch ein Array?

Wir brauchen einen neuen Container!

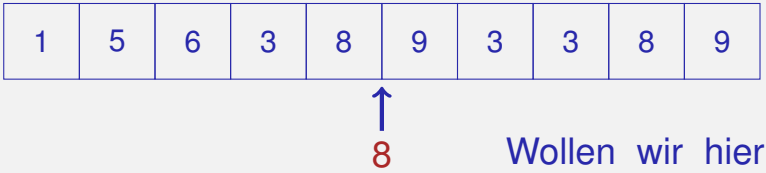
Unser Haupt-Container bisher: Array (`T[]`)

- Zusammenhängender Speicherbereich, wahlfreier Zugriff (auf i-tes Element)
- Simulation eines Stapels durch ein Array?
- Nein, irgendwann ist das Array “voll.”



Arrays können wirklich nicht alles…

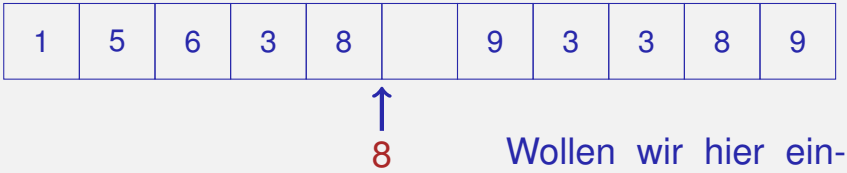
- Einfügen oder Löschen von Elementen „in der Mitte” ist aufwändig.



Wollen wir hier einfügen, müssen wir alles rechts davon verschieben (falls da überhaupt noch Platz ist!)

Arrays können wirklich nicht alles…

- Einfügen oder Löschen von Elementen „in der Mitte” ist aufwändig.



Wollen wir hier einfügen, müssen wir alles rechts davon verschieben (falls da überhaupt noch Platz ist!)

Arrays können wirklich nicht alles…

- Einfügen oder Löschen von Elementen „in der Mitte” ist aufwändig.

Wollen wir hier einfügen, müssen wir alles rechts davon verschieben (falls da überhaupt noch Platz ist!)

Arrays können wirklich nicht alles…

- Einfügen oder Löschen von Elementen „in der Mitte” ist aufwändig.

Arrays können wirklich nicht alles…

- Einfügen oder Löschen von Elementen „in der Mitte” ist aufwändig.

Arrays können wirklich nicht alles…

- Einfügen oder Löschen von Elementen „in der Mitte” ist aufwändig.

Wollen wir hier löschen, müssen wir alles rechts davon verschieben

Der neue Container: Verkettete Liste

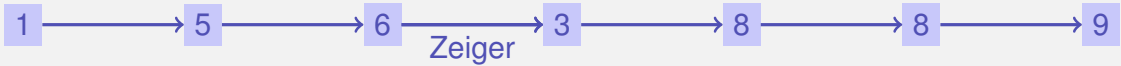
- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff

Der neue Container: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element “kennt” seinen Nachfolger

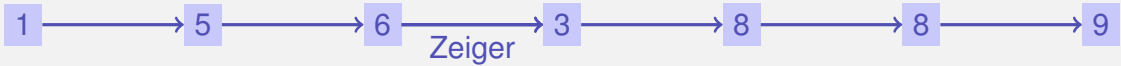
Der neue Container: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element “kennt” seinen Nachfolger



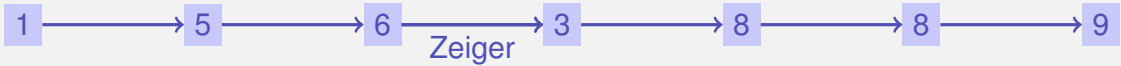
Der neue Container: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element “kennt” seinen Nachfolger
- Einfügen und Löschen beliebiger Elemente ist einfach, *auch am Anfang der Liste*

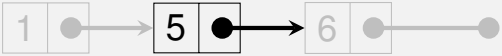


Der neue Container: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element “kennt” seinen Nachfolger
- Einfügen und Löschen beliebiger Elemente ist einfach, *auch am Anfang der Liste*
- $\Rightarrow$ Ein Stapel kann als verkettete Liste realisiert werden

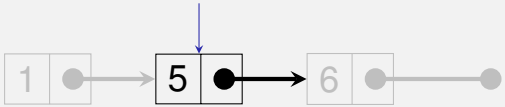


Verkettete Liste: Zoom

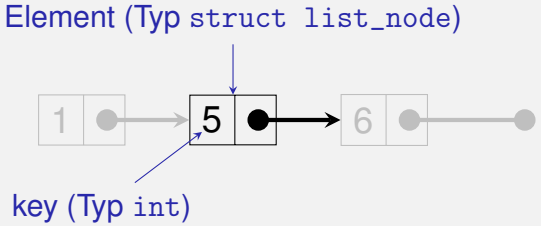


Verkettete Liste: Zoom

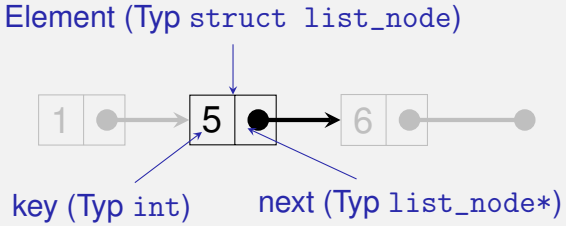
Element (Typ `struct list_node`)



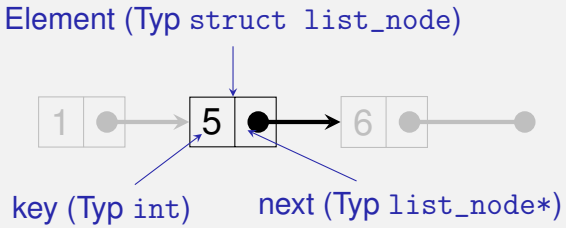
Verkettete Liste: Zoom



Verkettete Liste: Zoom

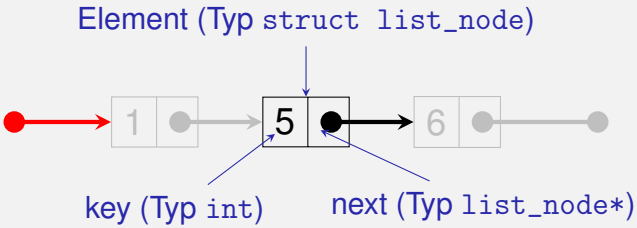


Verkettete Liste: Zoom




```
struct list_node {
  int           key;
  list_node*    next;
  // constructor
  list_node (int k, list_node* n)
    : key (k), next (n) {}
};
```

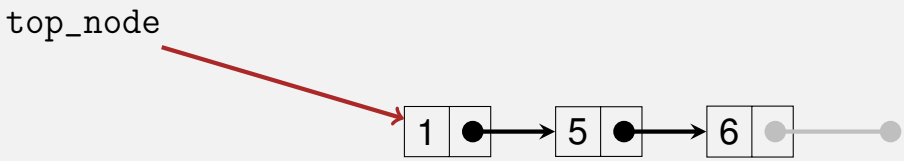
Stapel = Zeiger aufs oberste Element




```
class stack {
public:
   void push (int value) {...}
   ...
private:
   list_node* top_node;
};
```

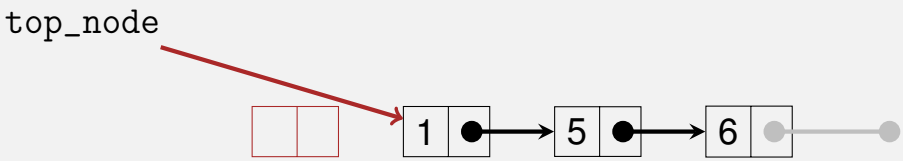
Sneak Preview: push(4)

```
void push (int value)
{
  top_node = new list_node (value, top_node);
}
```



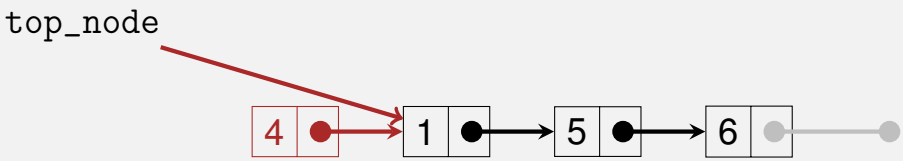
Sneak Preview: push(4)

```
void push (int value)
{
  top_node = new list_node (value, top_node);
}
```



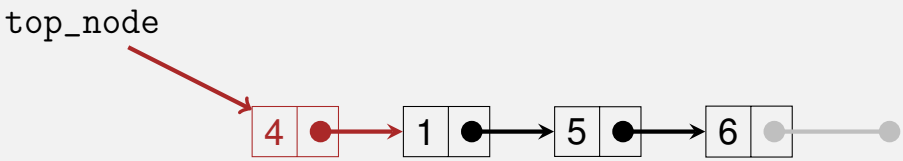
Sneak Preview: push(4)

```
void push (int value)
{
  top_node = new list_node (value, top_node);
}
```

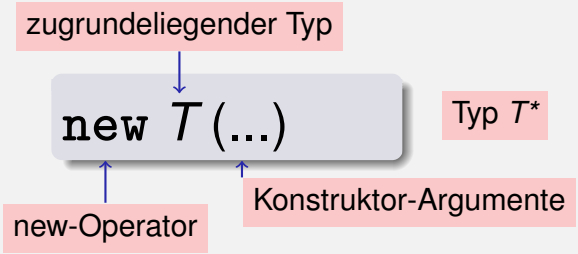


Sneak Preview: push(4)

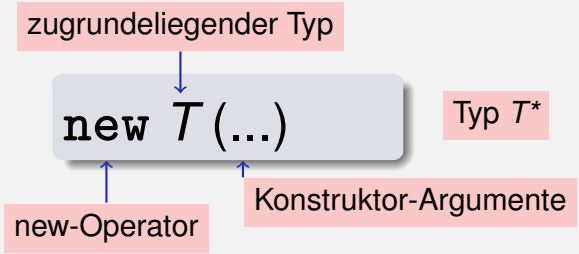
```
void push (int value)
{
  top_node = new list_node (value, top_node);
}
```



Der `new`-Ausdruck

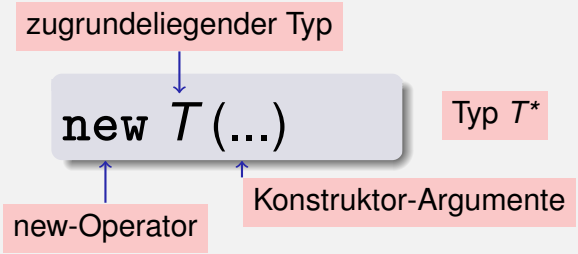


Der `new`-Ausdruck



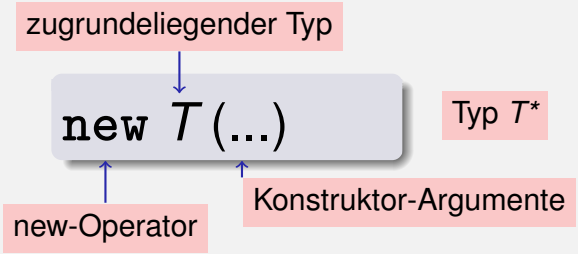
- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt ...

Der `new`-Ausdruck



- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.

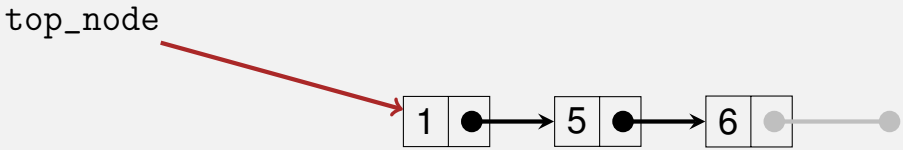
Der `new`-Ausdruck



- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.
- **Wert:** Adresse des neuen Objekts

Der `new`-Ausdruck: `push(4)`

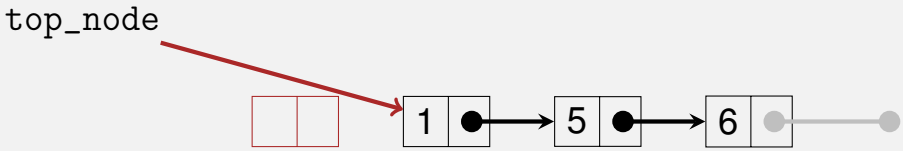
```
top_node = new list_node (value, top_node);
```



Der `new`-Ausdruck: `push(4)`

- **Effekt:** Neues Objekt vom Typ *T* wird im Speicher angelegt ...

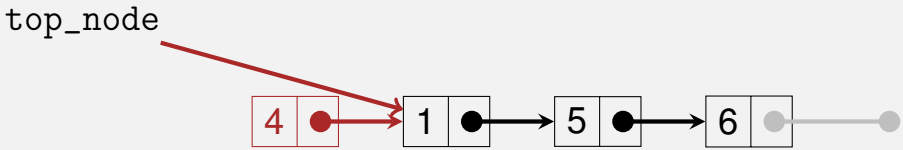
```
top_node = new list_node (value, top_node);
```



Der `new`-Ausdruck: `push(4)`

- **Effekt:** Neues Objekt vom Typ *T* wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.

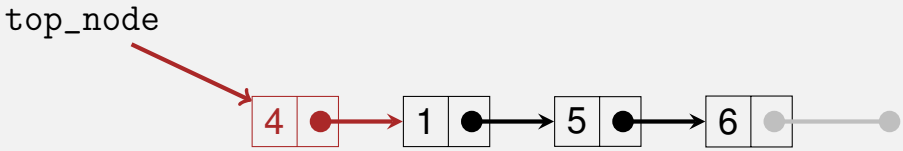
```
top_node = new list_node (value, top_node);
```



Der `new`-Ausdruck: `push(4)`

- **Effekt:** Neues Objekt vom Typ *T* wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.
- **Wert:** Adresse des neuen Objekts

```
top_node = new list_node (value, top_node);
```

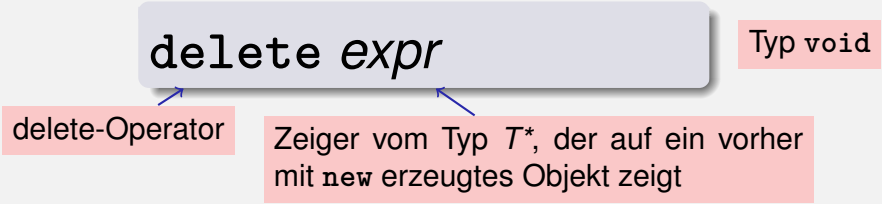


Der `delete`-Ausdruck

Objekte, die mit **`new`** erzeugt worden sind, haben *dynamische Speicherdauer:* sie “leben”, bis sie explizit *gelöscht* werden.

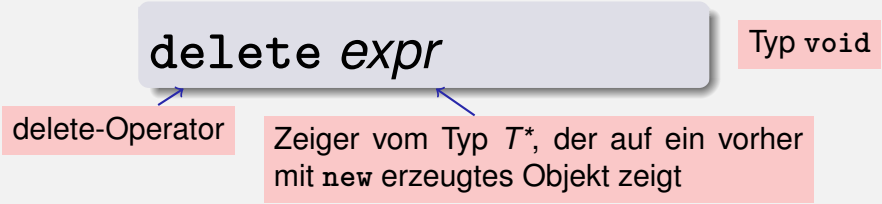
Der `delete`-Ausdruck

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 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`!

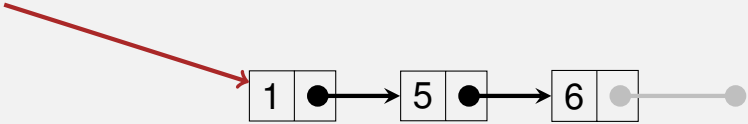
Nichtbeachtung führt zu *Speicherlecks*:

- “Alte” Objekte, die den Speicher blockieren...
- ...bis er irgendwann voll ist (heap overflow)

Weiter mit dem Stapel: pop()

```
void pop()
{
    assert (!empty());
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

top_node

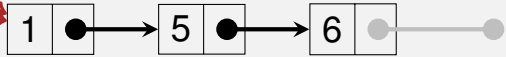


Weiter mit dem Stapel: pop()

```
void pop()
{
    assert (!empty());
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

top_node

p

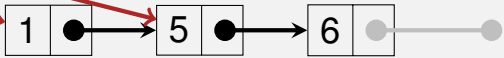


Weiter mit dem Stapel: pop()

```
void pop()
{
    assert (!empty());
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

top_node

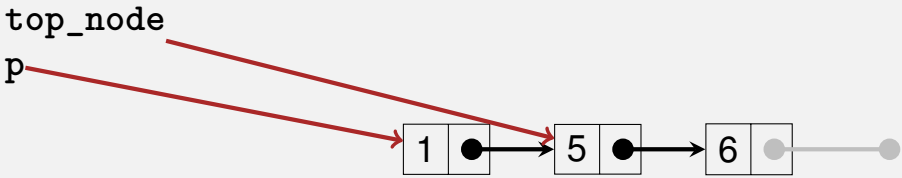
p



Weiter mit dem Stapel: pop()

```
void pop()
{
    assert (!empty());
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

Abkürzung für `(*top_node).next`

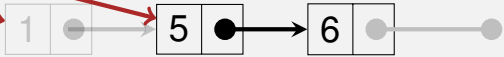


Weiter mit dem Stapel: pop()

```
void pop()
{
    assert (!empty());
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

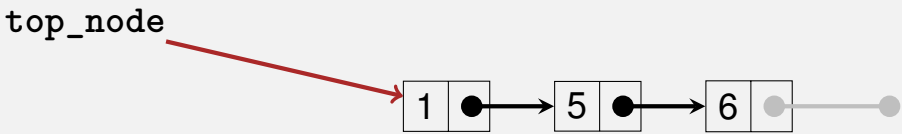
top_node

p



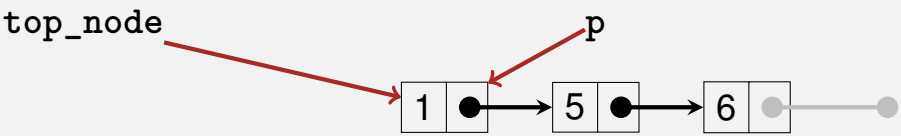
Stapel traversieren: print()

```
void print (std::ostream& o) const
{
  const list_node* p = top_node;
  while (p != 0) {
    o << p->key << " ";
    p = p->next;
  }
}
```



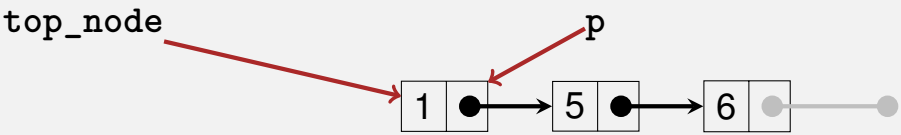
Stapel traversieren: print()

```
void print (std::ostream& o) const
{
  const list_node* p = top_node;
  while (p != 0) {
    o << p->key << " ";
    p = p->next;
  }
}
```



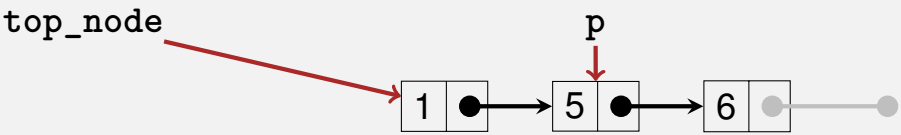
Stapel traversieren: print()

```
void print (std::ostream& o) const
{
  const list_node* p = top_node;
  while (p != 0) {
    o << p->key << " "; // 1
    p = p->next;
  }
}
```



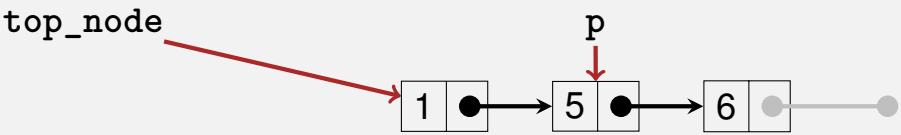
Stapel traversieren: print()

```
void print (std::ostream& o) const
{
  const list_node* p = top_node;
  while (p != 0) {
    o << p->key << " "; // 1
    p = p->next;
  }
}
```



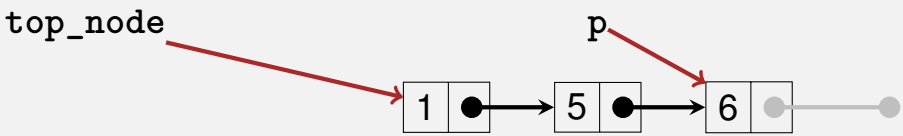
Stapel traversieren: print()

```
void print (std::ostream& o) const
{
  const list_node* p = top_node;
  while (p != 0) {
    o << p->key << " "; // 1 5
    p = p->next;
  }
}
```



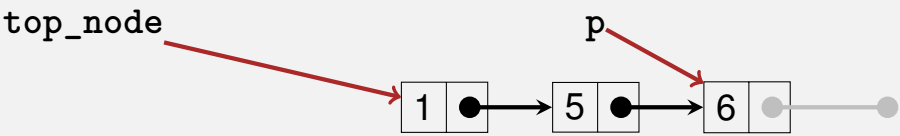
Stapel traversieren: print()

```
void print (std::ostream& o) const
{
  const list_node* p = top_node;
  while (p != 0) {
    o << p->key << " "; // 1 5
    p = p->next;
  }
}
```



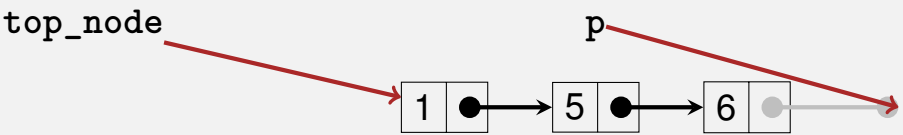
Stapel traversieren: print()

```
void print (std::ostream& o) const
{
  const list_node* p = top_node;
  while (p != 0) {
    o << p->key << " "; // 1 5 6
    p = p->next;
  }
}
```



Stapel traversieren: print()

```
void print (std::ostream& o) const
{
  const list_node* p = top_node;
  while (p != 0) {
    o << p->key << " "; // 1 5 6
    p = p->next;
  }
}
```



Stapel ausgeben: operator<<

```
class stack {
public:
   void push (int value) {...}
   ...
   void print (std::ostream& o) const {...}
private:
   list_node* top_node;
};
```

Stapel ausgeben: operator<<

```
class stack {
public:
   void push (int value) {...}
   ...
   void print (std::ostream& o) const {...}
private:
   list_node* top_node;
};

// POST: s is written to o
std::ostream& operator<< (std::ostream& o, const stack& s)
{
   s.print (o);
   return o;
}
```

Leerer Stapel

```
stack()      // default constructor
    : top_node (0)
{}
```

Leerer Stapel , `empty()`

```
stack()      // default constructor
    : top_node (0)
{}

bool empty () const
{
  return top_node == 0;
}
```

Leerer Stapel , `empty()`, `top()`

```
stack()      // default constructor
    : top_node (0)
{}

bool empty () const
{
  return top_node == 0;
}

int top () const
{
  assert (!empty());
  return top_node->key;
}
```

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?

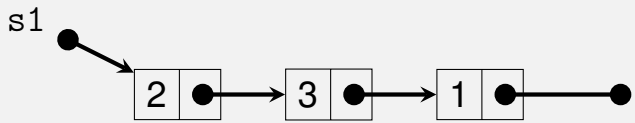
```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n"; // 2 3 1

stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop (); // Oops, Programmabsturz!
```

Stapel fertig?

Offenbar noch nicht...

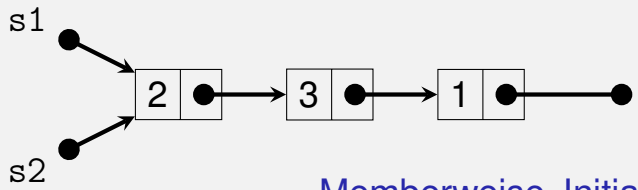
```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n"; // 2 3 1

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 ();
```

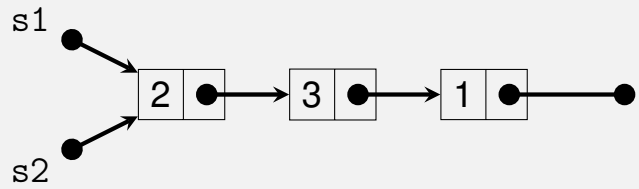
Was ist hier schiefgegangen?



Memberweise Initialisierung: kopiert nur den `top_node`-Zeiger

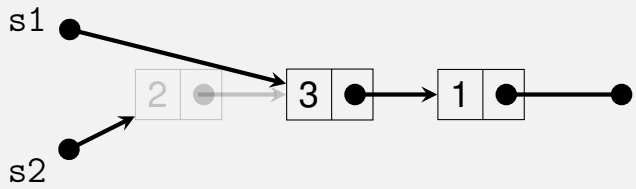
```
...
stack s2 = s1;
std::cout << s2 << "\n";

s1.pop ();
std::cout << s1 << "\n";

s2.pop ();
```

Was ist hier schiefgegangen?



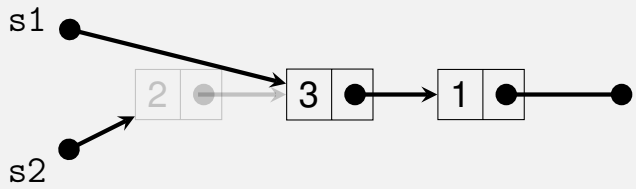

```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n";

s2.pop ();
```

Was ist hier schiefgegangen?



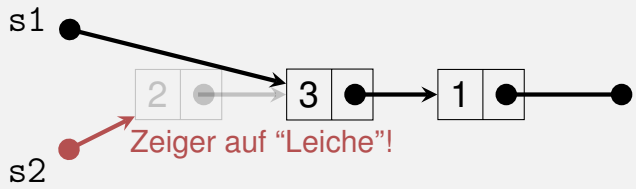

```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n";

s2.pop ();
```

Was ist hier schiefgegangen?



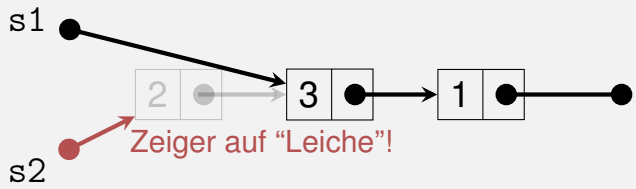

```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();
```

Was ist hier schiefgegangen?



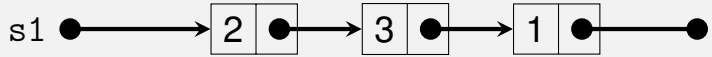

```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();
```

Was ist hier schiefgegangen?



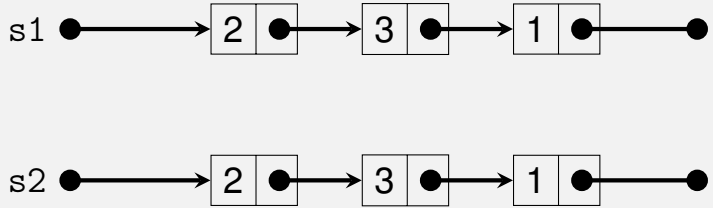

```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();  // Oops, Programmabsturz!
```

Wir brauchen eine echte Kopie!



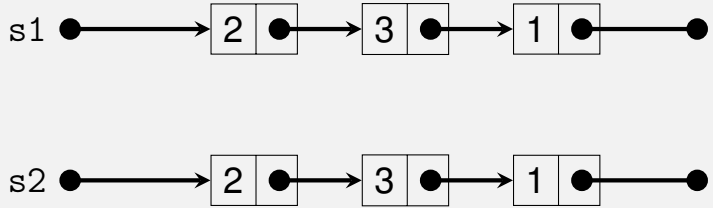

```
...
stack s2 = s1;
std::cout << s2 << "\n";

s1.pop ();
std::cout << s1 << "\n";

s2.pop ();
```

Wir brauchen eine echte Kopie!



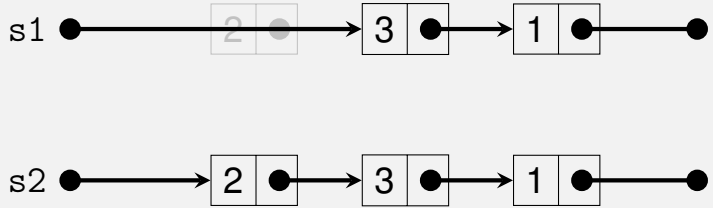

```
...
stack s2 = s1;
std::cout << s2 << "\n";

s1.pop ();
std::cout << s1 << "\n";

s2.pop ();
```

Wir brauchen eine echte Kopie!



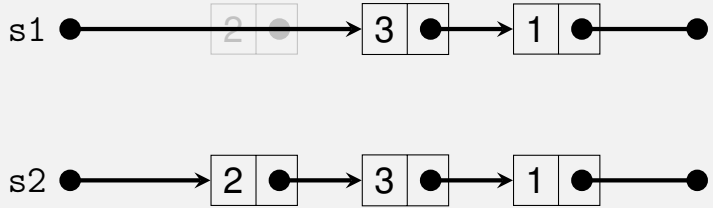

```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n";

s2.pop ();
```

Wir brauchen eine echte Kopie!



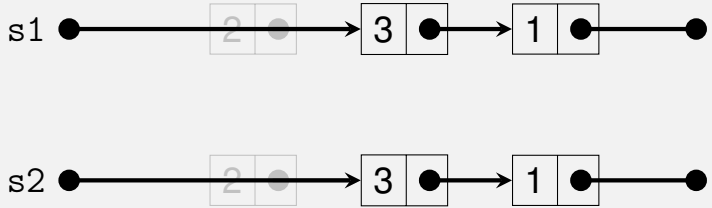

```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n";

s2.pop ();
```

Wir brauchen eine echte Kopie!



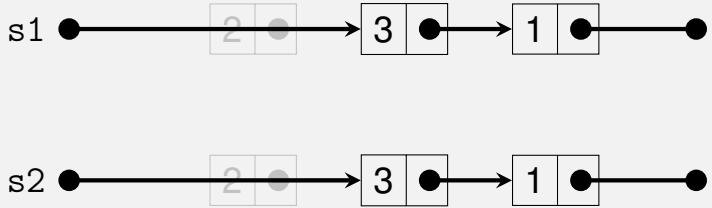

```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();
```

Wir brauchen eine echte Kopie!




```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();
```

Wir brauchen eine echte Kopie!

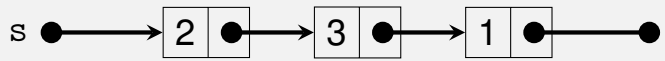



```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();  // ok
```

Mit dem Copy-Konstruktor klappt's!

Hier wird eine Kopierfunktion des `list_node` benutzt:

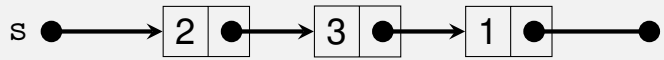
```
// POST: *this is initialized with a copy of s
stack (const stack& s)
  : top_node (0)
{
  if (s.top_node != 0)
    top_node = s.top_node->copy();
}
```



Mit dem Copy-Konstruktor klappt's!

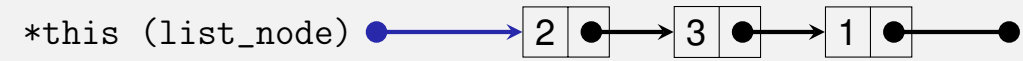
Hier wird eine Kopierfunktion des `list_node` benutzt:

```
// POST: *this is initialized with a copy of s
stack (const stack& s)
  : top_node (0)
{
  if (s.top_node != 0)
    top_node = s.top_node->copy();
}
```



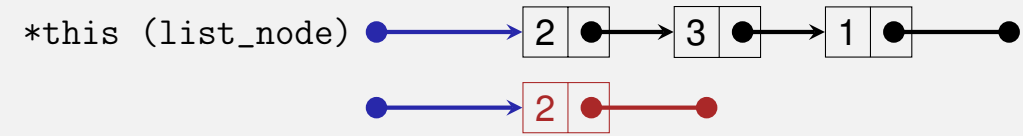
Die (rekursive) Kopierfunktion von `list_node`

```
// POST: pointer to a copy of the list starting
//       at *this is returned
list_node* copy () const
{
  if (next != 0)
    return new list_node (key, next->copy());
  else
    return new list_node (key, 0);
}
```



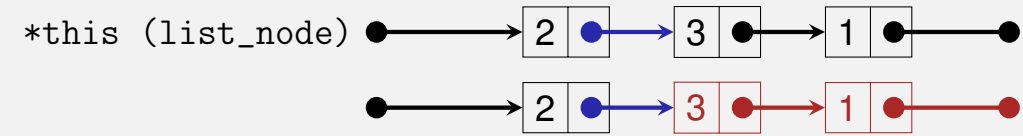
Die (rekursive) Kopierfunktion von `list_node`

```
// POST: pointer to a copy of the list starting
//       at *this is returned
list_node* copy () const
{
  if (next != 0)
    return new list_node (key, next->copy());
  else
    return new list_node (key, 0);
}
```



Die (rekursive) Kopierfunktion von `list_node`

```
// POST: pointer to a copy of the list starting
//       at *this is returned
list_node* copy () const
{
  if (next != 0)
    return new list_node (key, next->copy());
  else
    return new list_node (key, 0);
}
```



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!
```

Mit dem Zuweisungsoperator klappt's!

Hier wird eine Aufräumfunktion des `list_node` benutzt:

```
// POST: *this (left operand) is getting a copy of s (right operand)
stack& operator= (const stack& s)
{
  if (top_node != s.top_node) { // keine Selbtszuweisung!
    if (top_node != 0) {
      top_node->clear(); // loesche Knoten in *this
      top_node = 0;
  }
    if (s.top_node != 0)
       top_node = s.top_node->copy(); // kopiere s nach *this
    }
  return *this; // Rueckgabe als L-Wert (Konvention)
}
```

Mit dem Zuweisungsoperator klappt's!

Hier wird eine Aufräumfunktion des `list_node` benutzt:

```
// POST: *this (left operand) is getting a copy of s (right operand)
stack& operator= (const stack& s)
{
  if (top_node != s.top_node) { // keine Selbtszuweisung!
    if (top_node != 0) {
      top_node->clear(); // loesche Knoten in *this
      top_node = 0;
  }
    if (s.top_node != 0)
       top_node = s.top_node->copy(); // kopiere s nach *this
    }
  return *this; // Rueckgabe als L-Wert (Konvention)
}
```

Mit dem Zuweisungsoperator klappt's!

Hier wird eine Aufräumfunktion des `list_node` benutzt:

```
// POST: *this (left operand) is getting a copy of s (right operand)
stack& operator= (const stack& s)
{
  if (top_node != s.top_node) { // keine Selbtszuweisung!
    if (top_node != 0) {
      top_node->clear(); // loesche Knoten in *this
      top_node = 0;
  }
    if (s.top_node != 0)
       top_node = s.top_node->copy(); // kopiere s nach *this
    }
  return *this; // Rueckgabe als L-Wert (Konvention)
}
```

Mit dem Zuweisungsoperator klappt's!

Hier wird eine Aufräumfunktion des `list_node` benutzt:

```
// POST: *this (left operand) is getting a copy of s (right operand)
stack& operator= (const stack& s)
{
  if (top_node != s.top_node) { // keine Selbtszuweisung!
    if (top_node != 0) {
      top_node->clear(); // loesche Knoten in *this
      top_node = 0;
  }
    if (s.top_node != 0)
       top_node = s.top_node->copy(); // kopiere s nach *this
    }
  return *this; // Rueckgabe als L-Wert (Konvention)
}
```

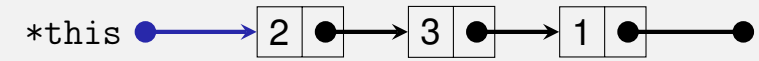
Mit dem Zuweisungsoperator klappt's!

Hier wird eine Aufräumfunktion des `list_node` benutzt:

```
// POST: *this (left operand) is getting a copy of s (right operand)
stack& operator= (const stack& s)
{
  if (top_node != s.top_node) { // keine Selbtszuweisung!
    if (top_node != 0) {
      top_node->clear(); // loesche Knoten in *this
      top_node = 0;
  }
    if (s.top_node != 0)
       top_node = s.top_node->copy(); // kopiere s nach *this
    }
  return *this; // Rueckgabe als L-Wert (Konvention)
}
```

Die (rekursive) Aufräumfunktion des `list_node`

```
// POST: the list starting at *this is deleted
void clear ()
{
  if (next != 0)
    next->clear();
  delete this;
}
```



Die (rekursive) Aufräumfunktion des `list_node`

```
// POST: the list starting at *this is deleted
void clear ()
{
  if (next != 0)
    next->clear();
  delete this;
}
```

Die (rekursive) Aufräumfunktion des `list_node`

```
// POST: the list starting at *this is deleted
void clear ()
{
  if (next != 0)
    next->clear();
  delete this;
}
```

Zombie-Elemente

```
{
  stack s1; // local variable
  s1.push (1);
  s1.push (3);
  s1.push (2);
  std::cout << s1 << "\n"; // 2 3 1
}
// s1 has died (become invalid)...
```

Zombie-Elemente

```
{
  stack s1; // local variable
  s1.push (1);
  s1.push (3);
  s1.push (2);
  std::cout << s1 << "\n"; // 2 3 1
}
// s1 has died (become invalid)...
```

- . . . aber die drei *Elemente* des Stapels `s1` leben weiter (Speicherleck)!

Zombie-Elemente

```
{
  stack s1; // local variable
  s1.push (1);
  s1.push (3);
  s1.push (2);
  std::cout << s1 << "\n"; // 2 3 1
}
// s1 has died (become invalid)...
```

- . . . aber die drei *Elemente* des Stapels `s1` leben weiter (Speicherleck)!
- Sie sollten zusammen mit `s1` aufgeräumt werden!

Mit dem Destruktor klappt's!

```
// POST: the dynamic memory of *this is deleted
~stack()
{
  if (top_node != 0)
    top_node->clear();
}
```

Mit dem Destruktor klappt's!

```
// POST: the dynamic memory of *this is deleted
~stack()
{
  if (top_node != 0)
    top_node->clear();
}
```

- löscht automatisch alle Stapelemente, wenn der Stapel ungültig wird

Mit dem Destruktor klappt's!

```
// POST: the dynamic memory of *this is deleted
~stack()
{
  if (top_node != 0)
    top_node->clear();
}
```

- löscht automatisch alle Stapelemente, wenn der Stapel ungültig wird
- Unsere Stapel-Klasse befolgt jetzt die Richtlinie "Dynamischer Speicher"!

Dynamischer Datentyp

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Stapel)

Dynamischer Datentyp

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Stapel)
- Andere Anwendungen:
  - Listen (mit Einfügen und Löschen “in der Mitte”)
  - Bäume (nächste Woche)
  - Warteschlangen
  - Graphen

Dynamischer Datentyp

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Stapel)
- Mindestfunktionalität:
  - Konstruktoren
  - Destruktor
  - Copy-Konstruktor
  - Zuweisungsoperator

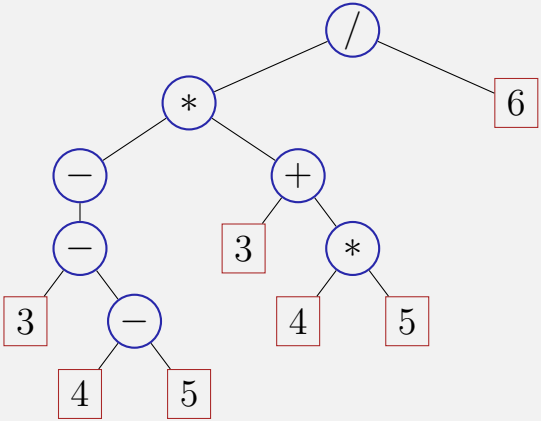
} Dreierregel: definiert eine Klasse eines davon, so sollte sie auch die anderen zwei definieren!

19. Vererbung und Polymorphie

Ausdrucksbäume, Vererbung, Code-Wiederverwendung, virtuelle Funktionen, Polymorphie, Konzepte des objektorientierten Programmierens

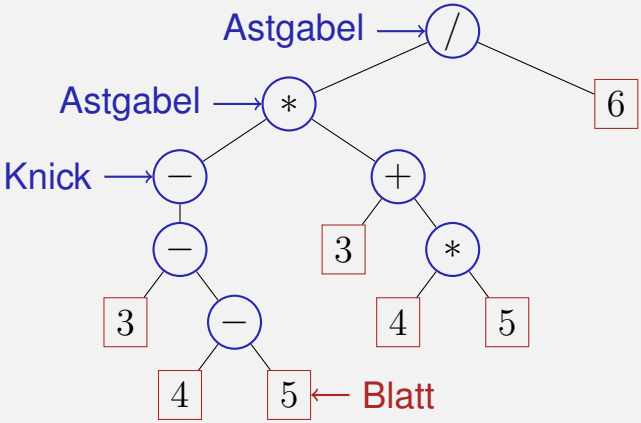
(Ausdrucks-)Bäume

```
-(3-(4-5))*(3+4*5)/6
```



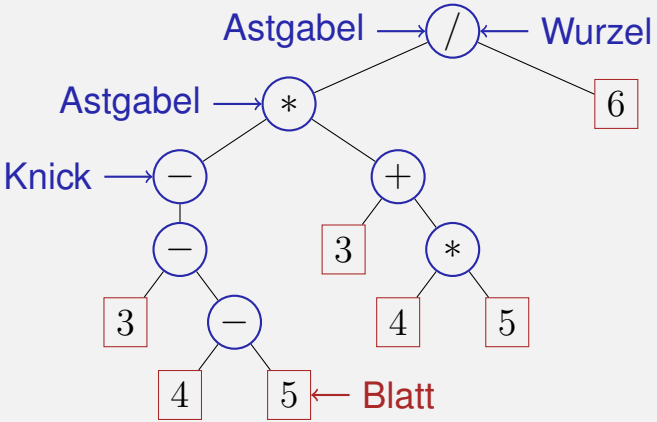
(Ausdrucks-)Bäume

```
-(3-(4-5))*(3+4*5)/6
```

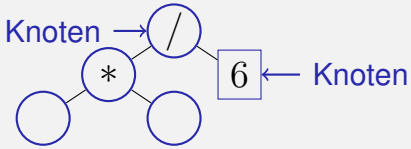


(Ausdrucks-)Bäume

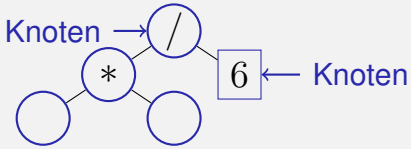
```
-(3-(4-5))*(3+4*5)/6
```



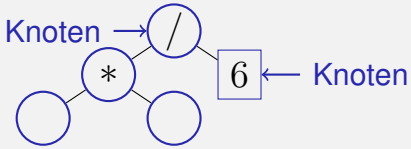
Astgabeln + Blätter + Knicke = Knoten

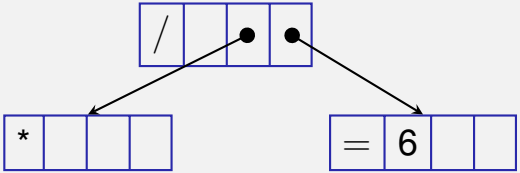


Astgabeln + Blätter + Knicke = Knoten

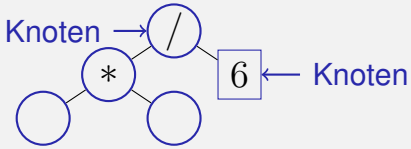


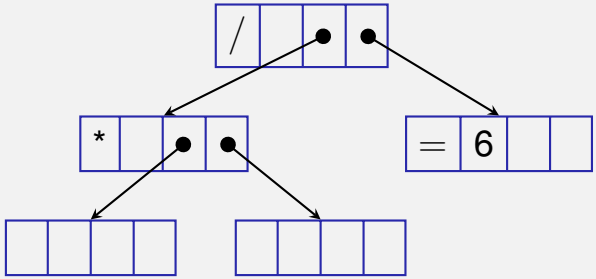
Astgabeln + Blätter + Knicke = Knoten



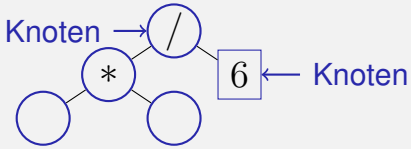


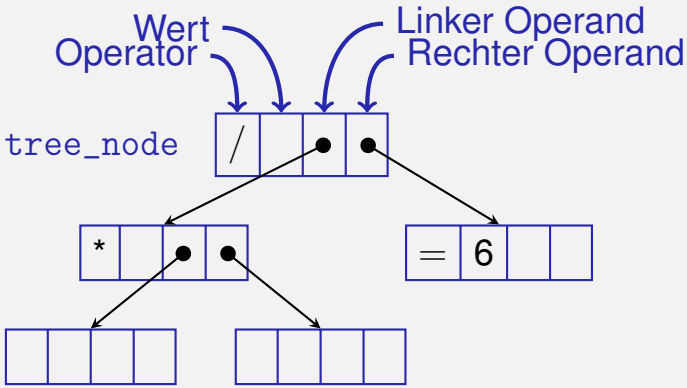
Astgabeln + Blätter + Knicke = Knoten



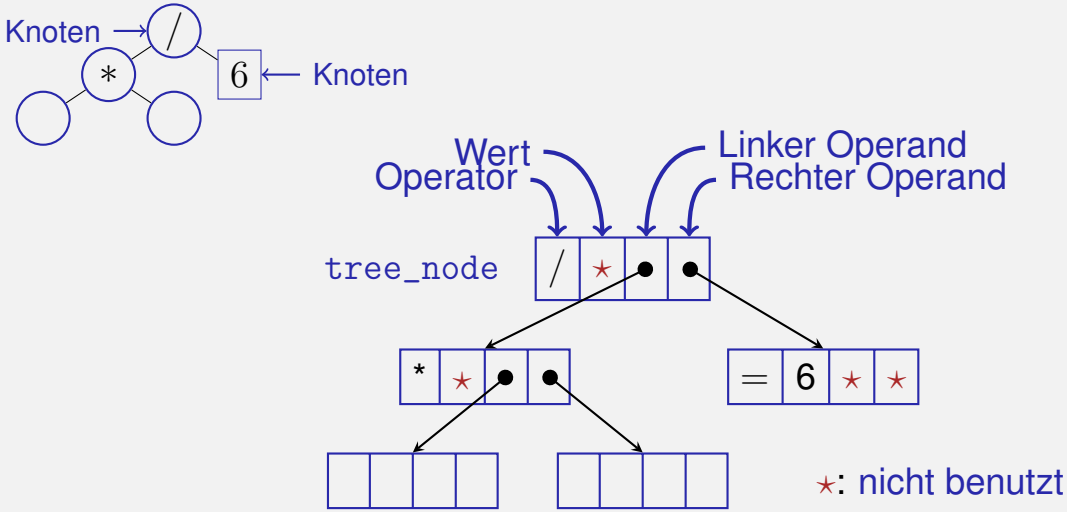


Astgabeln + Blätter + Knicke = Knoten

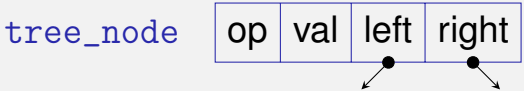




Astgabeln + Blätter + Knicke = Knoten

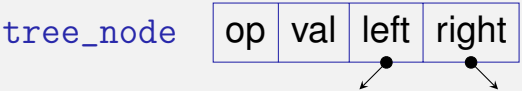


Knoten (`struct tree_node`)



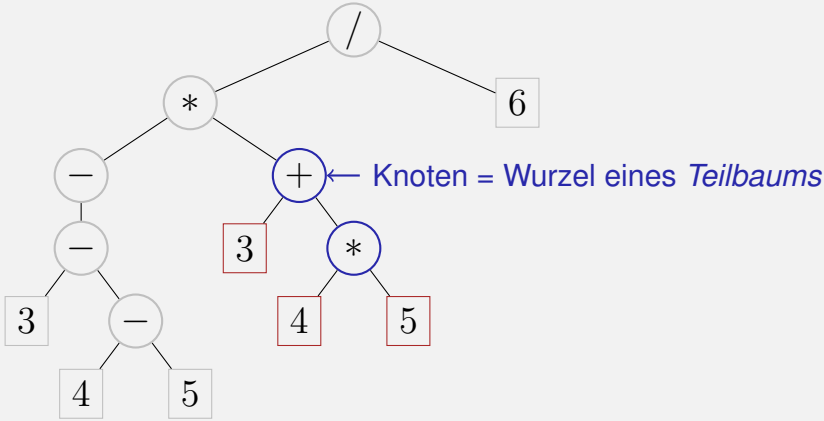

```
struct tree_node {
  char op;
  // leaf node (op: '=')
  double val;
  // internal node ( op: '+', '-', '*', '/')
  tree_node* left;
  tree_node* right;
  // constructor
  tree_node (char o, double v, tree_node* l, tree_node* r)
    : op (o), val (v), left (l), right (r)
  {}
};
```

Knoten (`struct tree_node`)

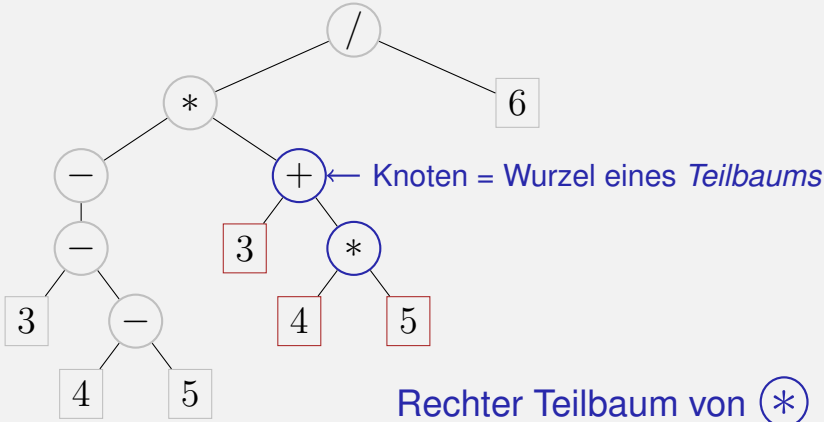



```
struct tree_node {
  char op;
  // leaf node (op: '=')
  double val;
  // internal node ( op: '+', '-', '*', '/')
  tree_node* left; // == 0 für unäres minus
  tree_node* right;
  // constructor
  tree_node (char o, double v, tree_node* l, tree_node* r)
    : op (o), val (v), left (l), right (r)
  {}
};
```

Knoten und Teilbäume



Knoten und Teilbäume



Knoten in Teilbäumen zählen


```
struct tree_node {
  ...
    // POST: returns the size (number of nodes) of
    //       the subtree with root *this
    int size () const
    {
        int s=1;
        if (left)
            s += left->size();
        if (right)
            s += right->size();
        return s;
    }
};
```

Knoten in Teilbäumen zählen


```
struct tree_node {
  ...
    // POST: returns the size (number of nodes) of
    //       the subtree with root *this
    int size () const
    {
        int s=1;
        if (left) // kurz für left != 0
            s += left->size();
        if (right)
            s += right->size();
        return s;
    }
};
```

Teilbäume auswerten


```
struct tree_node {
  ...
  // POST: evaluates the subtree with root *this
  double eval () const {
    if (op == '=') return val;
    double l = 0;
    if (left) l = left->eval();
    double r = right->eval();
    if (op == '+') return l + r;
    if (op == '-') return l - r;
    if (op == '*') return l * r;
    if (op == '/') return l / r;
    return 0;
  }
};
```

← Blatt...

...oder Astgabel:

← op unär, oder linker Ast

← rechter Ast

Teilbäume klonen


```
struct tree_node {
  ...
  // POST: a copy of the subtree with root *this is
  //       made, and a pointer to its root node is
  //       returned
  tree_node* copy () const {
    tree_node* to = new tree_node (op, val, 0, 0);
    if (left)
        to->left = left->copy();
    if (right)
        to->right = right->copy();
    return to;
  }
};
```

Teilbäume klonen - Kompaktere Schreibweise


```
struct tree_node {
  ...
  // POST: a copy of the subtree with root *this is
  //       made, and a pointer to its root node is
  //       returned
  tree_node* copy () const {
     return new tree_node (op, val,
            left ? left->copy() : 0,
            right ? right->copy() : 0);
  }
};
```

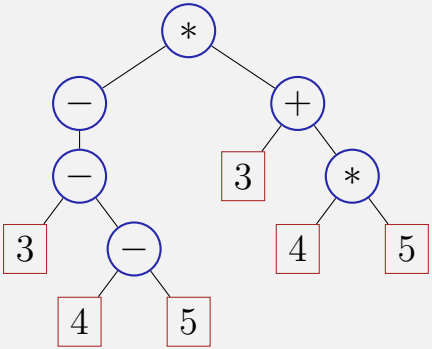
Teilbäume klonen - Kompaktere Schreibweise


```
struct tree_node {
  ...
  // POST: a copy of the subtree with root *this is
  //       made, and a pointer to its root node is
  //       returned
  tree_node* copy () const {
     return new tree_node (op, val,
           left ? left->copy() : 0,
           right ? right->copy() : 0);
  }
};
```

cond ? expr1 : expr 2 hat Wert *expr1*, falls *cond* gilt, *expr2* sonst

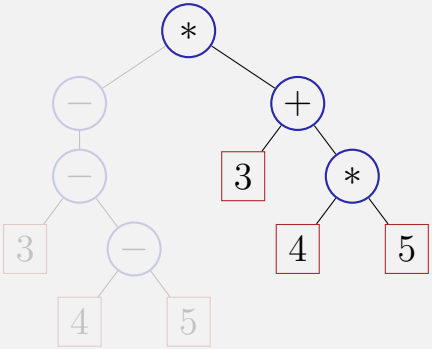
Teilbäume fällen


```
struct tree_node {
  ...
  // POST: all nodes in the subtree with root
  // *this are deleted
  void clear() {
    if (left) {
      left->clear();
    }
    if (right) {
      right->clear();
    }
    delete this;
  }
};
```



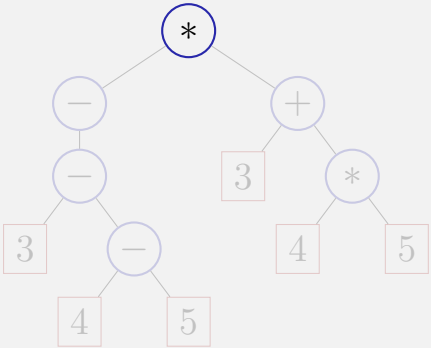
Teilbäume fällen

```
struct tree_node {
  ...
  // POST: all nodes in the subtree with root
  // *this are deleted
  void clear() {
    if (left) {
      left->clear();
    }
    if (right) {
      right->clear();
    }
    delete this;
  }
};
```



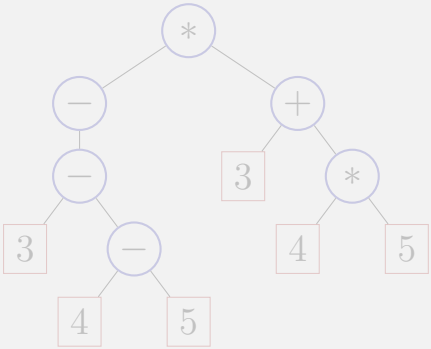
Teilbäume fällen

```
struct tree_node {
  ...
  // POST: all nodes in the subtree with root
  // *this are deleted
  void clear() {
    if (left) {
      left->clear();
    }
    if (right) {
      right->clear();
    }
    delete this;
  }
};
```



Teilbäume fällen

```
struct tree_node {
  ...
  // POST: all nodes in the subtree with root
  // *this are deleted
  void clear() {
    if (left) {
      left->clear();
    }
    if (right) {
      right->clear();
    }
    delete this;
  }
};
```



Bäumige Teilbäume

```
struct tree_node {
  ...
  // constructor
  tree_node (char o, tree_node* l,
             tree_node* r, double v)

  // functionality
  double eval () const;
  void print (std::ostream& o) const;
  int size () const;
  tree_node* copy () const;
  void clear ();
};
```

Bäume pflanzen

```
class texpression {
private:
    tree_node* root;
public:
    ...
    texpression (double d)
        : root (new tree_node ('=', d, 0, 0)) {}
    ...
};
```

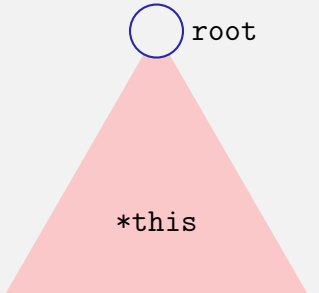
erzeugt Baum mit einem Blatt

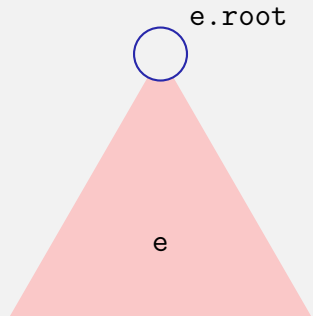
Bäume wachsen lassen

```
texpression& operator-= (const texpression& e)
{
  assert (e.root);
  root = new tree_node ('-', 0, root, e.root->copy());
  return *this;
}
```

Bäume wachsen lassen

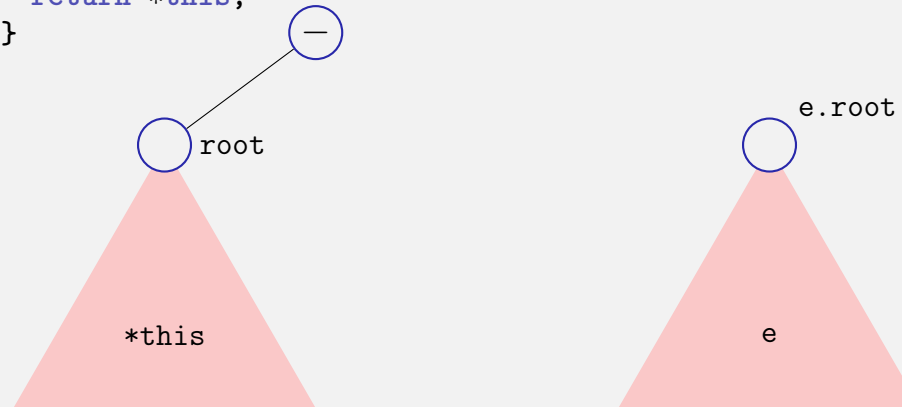
```
texpression& operator-= (const texpression& e)
{
  assert (e.root);
  root = new tree_node ('-', 0, root, e.root->copy());
  return *this;
}
```





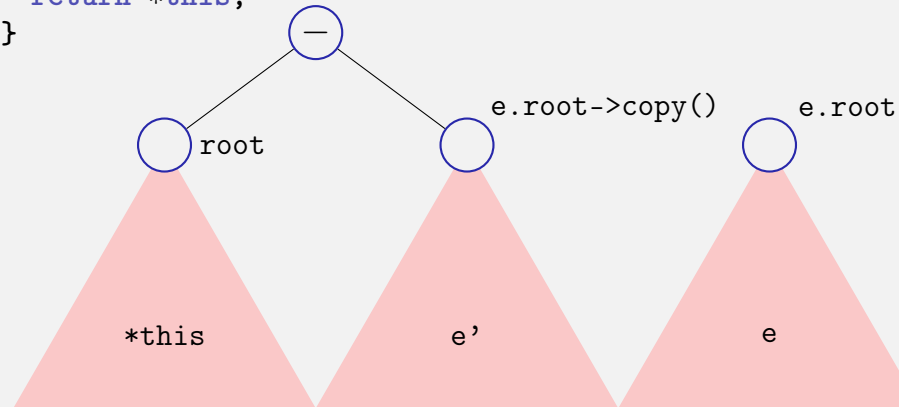
Bäume wachsen lassen

```
texpression& operator-= (const texpression& e)
{
  assert (e.root);
  root = new tree_node ('-', 0, root, e.root->copy());
  return *this;
}
```



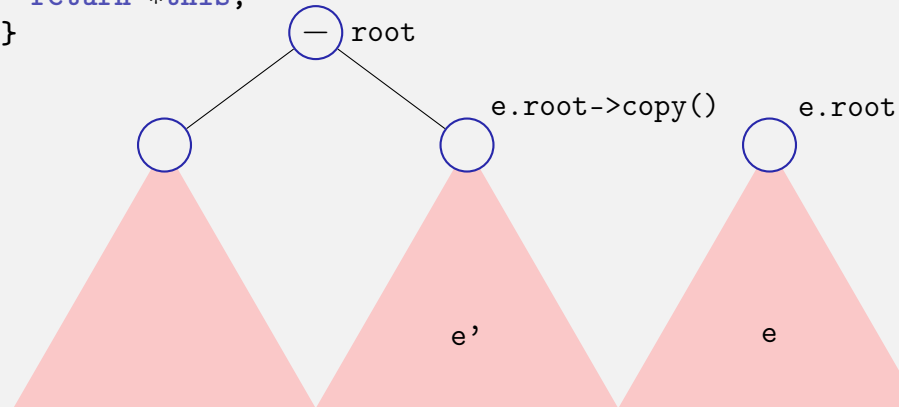
Bäume wachsen lassen

```
texpression& operator-= (const texpression& e)
{
  assert (e.root);
  root = new tree_node ('-', 0, root, e.root->copy());
  return *this;
}
```



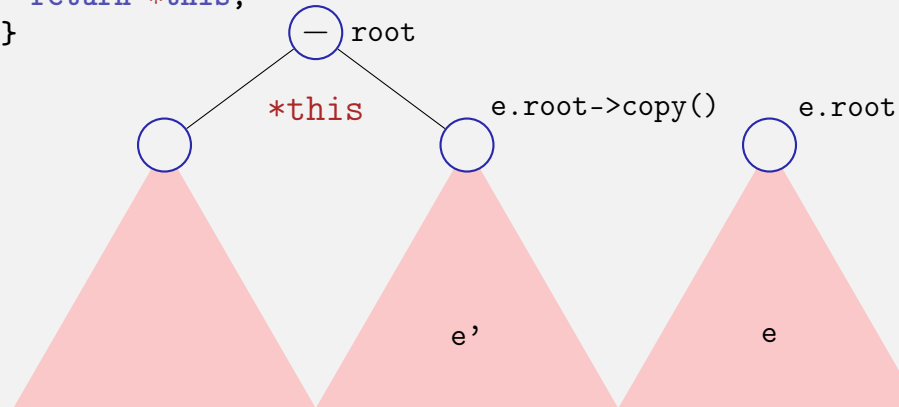
Bäume wachsen lassen

```
texpression& operator-= (const texpression& e)
{
  assert (e.root);
  root = new tree_node ('-', 0, root, e.root->copy());
  return *this;
}
```



Bäume wachsen lassen

```
texpression& operator-= (const texpression& e)
{
  assert (e.root);
  root = new tree_node ('-', 0, root, e.root->copy());
  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

```
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

Bäume züchten

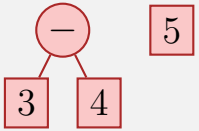
```
texpression operator- (const texpression& l,
                       const texpression& r)
{
    texpression result = l;
    return result -= r;
}

texpression a = 3;
texpression b = 4;
texpression c = 5;
texpression d = a-b-c;
```

Bäume züchten

```
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;
```

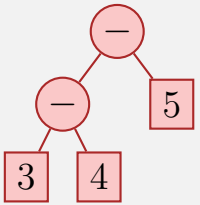
Bäume züchten

```
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

```
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;
```



Von Werten 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)

Von Werten zu Bäumen!

```
typedef double result_type;   // Typ-Alias

// term = factor { "*" factor | "/" factor }
result_type term (std::istream& is){
{
  result_type value = factor (is);
  while (true) {
    if (consume (is, '*'))
      value *= factor (is);
    else if (consume (is, '/'))
      value /= factor (is);
  else
      return value;
  }
}
```

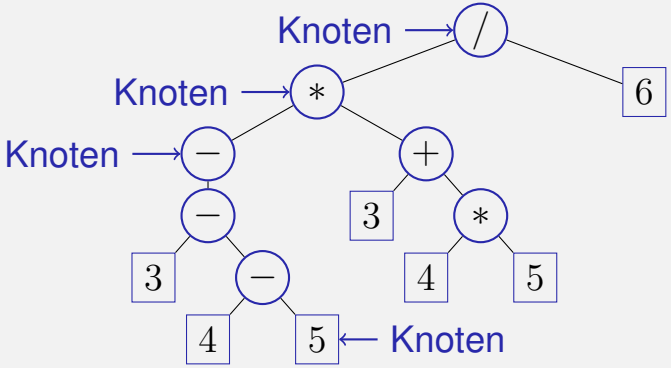
`double_calculator.cpp`
(Ausdruckswert)

Von Werten zu Bäumen!

```
typedef texpression result_type;   // Typ-Alias

// term = factor { "*" factor | "/" factor }
result_type term (std::istream& is){
{
  result_type value = factor (is);
  while (true) {
    if (consume (is, '*'))
      value *= factor (is);
    else if (consume (is, '/'))
      value /= factor (is);
  else
      return value;
  }
}
```

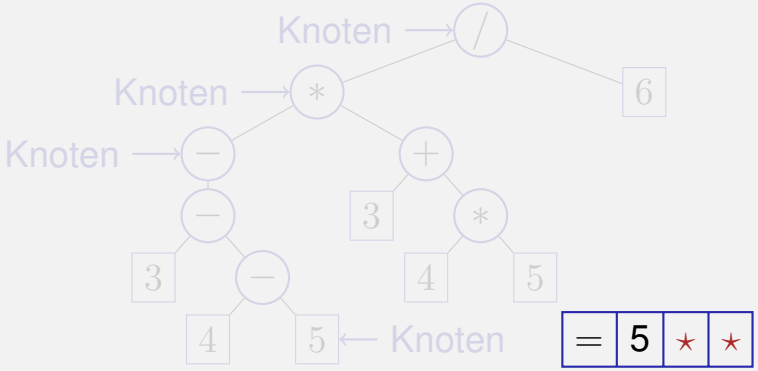
`double_calculator.cpp`
(Ausdruckswert)
$\rightarrow$
`texpression_calculator_1.cpp`
(Ausdrucksbaum)

Motivation Vererbung: Bisher



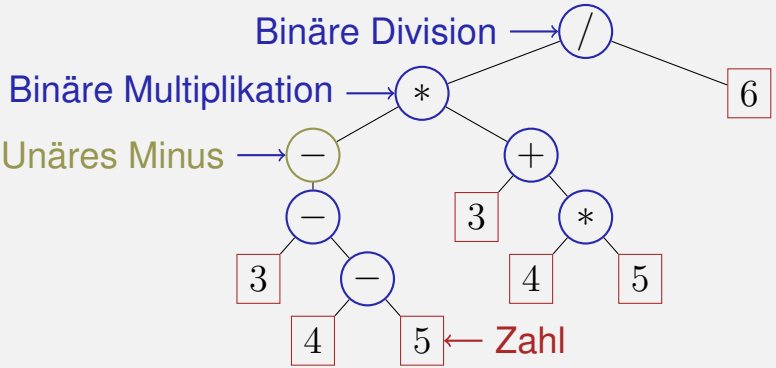
- Astgabeln + Blätter + Knicke = Knoten

Motivation Vererbung: Bisher



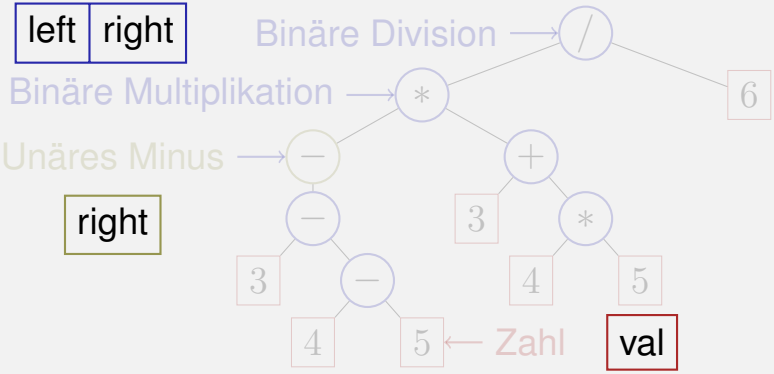
- Astgabeln + Blätter + Knicke = Knoten
- $\Rightarrow$ Unbenutzte Membervariablen $\star$

Motivation Vererbung: Die Wahrheit



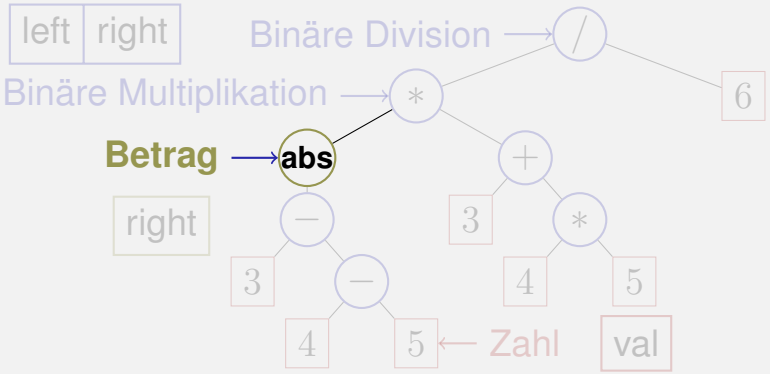
- “Zoo” von Astgabeln, Blättern und Knicken

Motivation Vererbung: Die Idee



- “Zoo” von Astgabeln, Blättern und Knicken
- Überall nur die benötigten Membervariablen!

Motivation Vererbung: Die Idee



- “Zoo” von Astgabeln, Blättern und Knicken
- Zoo-Erweiterung mit neuen Arten!

Vererbung – Der Hack, zum ersten...

Szenario: Erweiterung des Ausdrucksbaumes um mathematische Funktionen, z.B. `abs`, `sin`, `cos`:

Vererbung – Der Hack, zum ersten...

Szenario: Erweiterung des Ausdrucksbaumes um mathematische Funktionen, z.B. `abs`, `sin`, `cos`:

- Erweiterung der Klasse `tree_node` um noch mehr Membervariablen

```
struct tree_node{
    char op; // neu: op = 'f' -> Funktion
    ...
    std::string name; // function name;
}
```

Vererbung – Der Hack, zum zweiten…

Szenario: Erweiterung des Ausdrucksbaumes um mathematische Funktionen, z.B. `abs`, `sin`, `cos`:

Vererbung – Der Hack, zum zweiten...

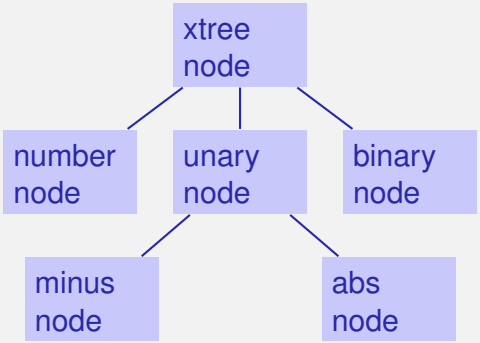
Szenario: Erweiterung des Ausdrucksbaumes um mathematische Funktionen, z.B. `abs`, `sin`, `cos`:

- Anpassung jeder einzelnen Memberfunktion member function

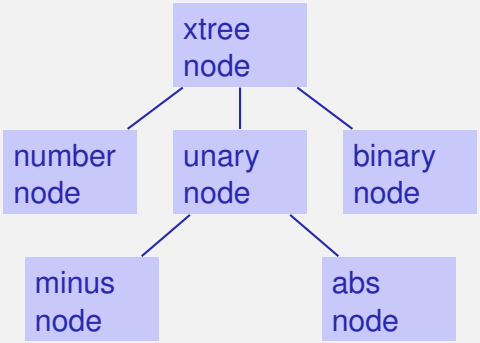
```
double eval () const
{   ...
    else if (op == 'f')
        if (name == "abs")
            return std::abs(right->eval());
    ...
}
```

Vererbung – die saubere Lösung

- „Aufspaltung” von `tree_node`



Vererbung – die saubere Lösung



- „Aufspaltung” von `tree_node`
- Gemeinsame Eigenschaften verbleiben in der *Basisklasse* `xtree_node` (Erklärung folgt)

Vererbung

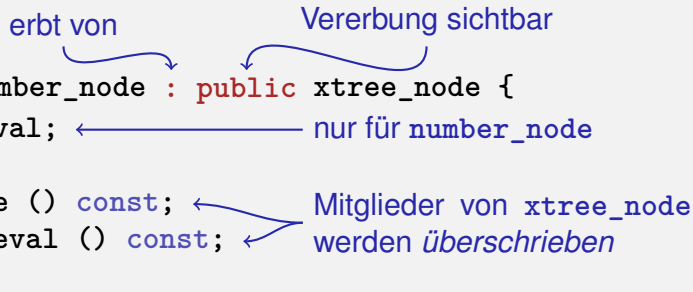
```
struct xtree_node{
  virtual int size() const;
  virtual double eval () const;
};
```

Vererbung

```
struct xtree_node{
  virtual int size() const;
  virtual double eval () const;
};


struct number_node : public xtree_node {
  double val;

  int size () const;
  double eval () const;
};
```

Vererbung

```
struct xtree_node{
  virtual int size() const;
  virtual double eval () const;
};
```



Aufgabenteilung: Der Zahlknoten

```
struct number_node: public xtree_node{
  double val;

  number_node (double v) : val (v) {}

  double eval () const {
    return val;
  }

  int size () const {
      return 1;
  }
};
```

Ein Zahlknoten ist ein Baumknoten…

- Ein (Zeiger auf ein) erbendes Objekt kann überall dort verwendet werden, wo ein (Zeiger auf ein) Basisobjekt gefordert ist

```
number_node* num = new number_node (5);
```

Ein Zahlknoten ist ein Baumknoten...

- Ein (Zeiger auf ein) erbendes Objekt kann überall dort verwendet werden, wo ein (Zeiger auf ein) Basisobjekt gefordert ist

```
number_node* num = new number_node (5);

xtree_node* tn = num; // ok, number_node is
                      // just a special xtree_node
```

Ein Zahlknoten ist ein Baumknoten...

- Ein (Zeiger auf ein) erbendes Objekt kann überall dort verwendet werden, wo ein (Zeiger auf ein) Basisobjekt gefordert ist

```
number_node* num = new number_node (5);

xtree_node* tn = num; // ok, number_node is
                      // just a special xtree_node

xtree_node* bn = new add_node (tn, num); // ok
```

Ein Zahlknoten ist ein Baumknoten...

- Ein (Zeiger auf ein) erbendes Objekt kann überall dort verwendet werden, wo ein (Zeiger auf ein) Basisobjekt gefordert ist, aber nicht umgekehrt.

```
number_node* num = new number_node (5);

xtree_node* tn = num; // ok, number_node is
                      // just a special xtree_node

xtree_node* bn = new add_node (tn, num); // ok

number_node* nn = tn; //error:invalid conversion
```

Anwendung

```
class xexpression {
private:
   xtree_node* root;
public:
    xexpression (double d)
      : root (new number_node (d)) {}

    xexpression& operator-= (const xexpression& t)
    {
      assert(t.root);
      root = new sub_node (root, t.root->copy());
      return *this;
    }
     ...
}
```

Anwendung

```
class xexpression {
private:
   xtree_node* root;
public:
    xexpression (double d)
      : root (new number_node (d)) {}

    xexpression& operator-= (const xexpression& t)
    {
      assert(t.root);
      root = new sub_node (root, t.root->copy());
      return *this;
    }
     ...
}
```

statischer Typ

dynamischer Typ

Polymorphie

- Virtuelle Mitgliedsfunktion: der *dynamische* Typ bestimmt bei Zeigern auf erbende Objekte die auszuführenden Memberfunktionen

```
struct xtree_node {
  virtual double eval();
...
};
```

- Ohne `virtual` wird der *statische Typ* zur Bestimmung der auszuführenden Funktion herangezogen.

Wir vertiefen das nicht weiter.

Polymorphie

- Virtuelle Mitgliedsfunktion: der *dynamische* Typ bestimmt bei Zeigern auf erbende Objekte die auszuführenden Memberfunktionen

```
struct xtree_node {
  virtual double eval();
...
};
```

- Ohne `virtual` wird der *statische Typ* zur Bestimmung der auszuführenden Funktion herangezogen.

Wir vertiefen das nicht weiter.

Polymorphie

- Virtuelle Mitgliedsfunktion: der *dynamische* Typ bestimmt bei Zeigern auf erbende Objekte die auszuführenden Memberfunktionen

```
struct xtree_node {
  virtual double eval();
...
};
```

- Ohne `virtual` wird der *statische Typ* zur Bestimmung der auszuführenden Funktion herangezogen.

Wir vertiefen das nicht weiter.

Polymorphie

- Virtuelle Mitgliedsfunktion: der *dynamische* Typ bestimmt bei Zeigern auf erbende Objekte die auszuführenden Memberfunktionen

```
struct xtree_node {
  virtual double eval();
...
};
```

- Ohne `virtual` wird der *statische Typ* zur Bestimmung der auszuführenden Funktion herangezogen.

Wir vertiefen das nicht weiter.

Aufgabenteilung: Binäre Knoten

```
struct binary_node : public xtree_node {
  xtree_node* left; // INV != 0
  xtree_node* right; // INV != 0

  binary_node (xtree_node* l, xtree_node* r) :
    left (l), right (r)
  {
    assert (left);
    assert (right);
  }

  int size () const {
    return 1 + left->size() + right->size();
  }
};
```

Aufgabenteilung: Binäre Knoten

```
struct binary_node : public xtree_node {
  xtree_node* left; // INV != 0
  xtree_node* right; // INV != 0

  binary_node (xtree_node* l, xtree_node* r) :
    left (l), right (r)
  {
    assert (left);
    assert (right);
  }

  int size () const {
    return 1 + left->size() + right->size();
  }
};
```

`size` funktioniert für *alle* binären Knoten. Abgeleitete Klassen (`add_node`, `sub_node`...) *erben* diese Funktion!

Aufgabenteilung: +, −, * ...

```
struct add_node : public binary_node {
  add_node (xtree_node* l, xtree_node* r)
    : binary_node (l, r) {}

  double eval () const {
    return left->eval() + right->eval();
  }
};
```

Aufgabenteilung: +, −, * ...

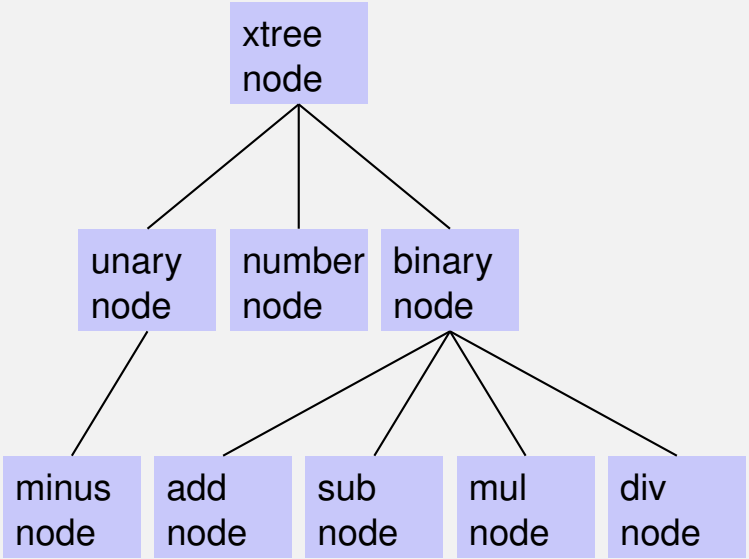
```
struct sub_node : public binary_node {
  sub_node (xtree_node* l, xtree_node* r)
    : binary_node (l, r) {}

  double eval () const {
    return left->eval() - right->eval();
  }
};
```

Aufgabenteilung: +, −, ∗ ...

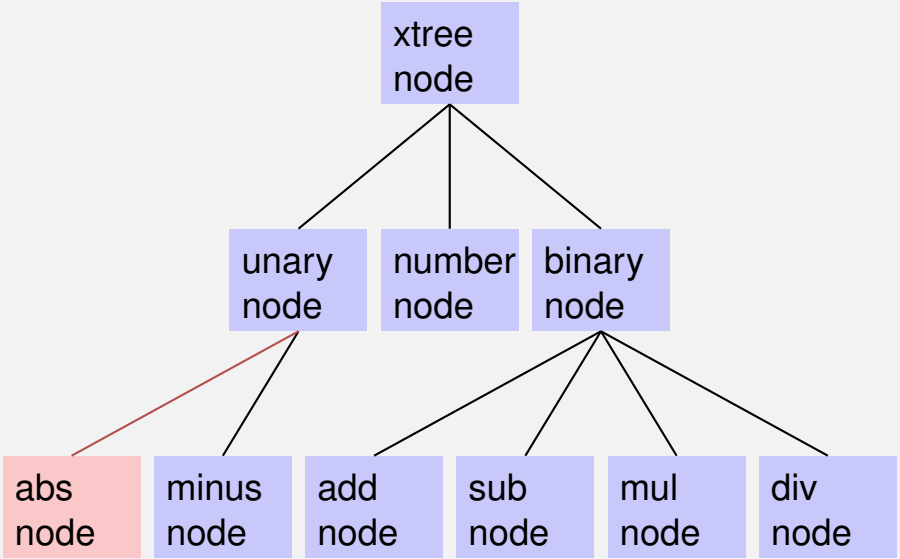
```
struct sub_node : public binary_node {
  sub_node (xtree_node* l, xtree_node* r)
    : binary_node (l, r) {}

  double eval () const {
    return left->eval() - right->eval();
  }
};
```

`eval` spezifisch
für `+`, `-`, `*`, `/`

Erweiterung um `abs` Funktion



Erweiterung um `abs` Funktion



Erweiterung um `abs` Funktion

```
struct unary_node: public xtree_node
{
  xtree_node* right; // INV != 0
  unary_node (xtree_node* r);
  int size () const;
};

struct abs_node: public unary_node
{
  abs_node (xtree_node* arg) : unary_node (arg) {}

  double eval () const {
    return std::abs (right->eval());
  }

};
```

Erweiterung um `abs` Funktion

```
struct unary_node: public xtree_node
{
  xtree_node* right; // INV != 0
  unary_node (xtree_node* r);
  int size () const;
};
```

```
struct abs_node: public unary_node
{
  abs_node (xtree_node* arg) : unary_node (arg) {}

  double eval () const {
    return std::abs (right->eval());
  }

};
```

Da ist noch was... Speicherbehandlung

```
struct xtree_node {
  ...
  // POST: a copy of the subtree with root
  //       *this is made, and a pointer to
  //       its root node is returned
  virtual xtree_node* copy () const;

  // POST: all nodes in the subtree with
  //        root *this are deleted
  virtual void clear () {};
};
```

Da ist noch was... Speicherbehandlung

```
struct unary_node: public xtree_node {
  ...
  virtual void clear () {
    right->clear();
    delete this;
  }
};

struct minus_node: public unary_node {
  ...
  xtree_node* copy () const
  {
    return new minus_node (right->copy());
  }
};
```

`xtree_node` ist kein dynamischer Datentyp ??

- Wir haben keine Variablen vom Typ `xtree_node` mit automatischer Speicherdauer

`xtree_node` ist kein dynamischer Datentyp ??

- Wir haben keine Variablen vom Typ `xtree_node` mit automatischer Speicherdauer
- Copy-Konstruktor, Zuweisungsoperator und Destruktor sind überflüssig

`xtree_node` ist kein dynamischer Datentyp ??

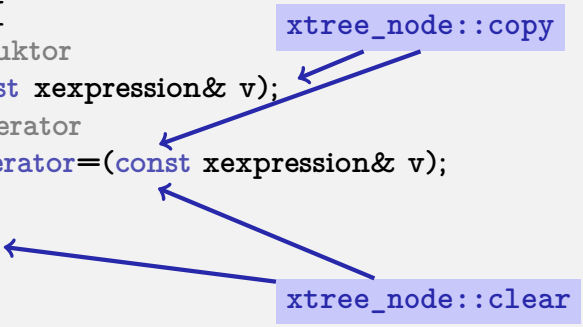
- Wir haben keine Variablen vom Typ `xtree_node` mit automatischer Speicherdauer
- Copy-Konstruktor, Zuweisungsoperator und Destruktor sind überflüssig
- Speicherverwaltung in der *Container-Klasse*

```
class xexpression {
  // Copy−Konstruktor
  xexpression (const xexpression& v);
  // Zuweisungsoperator
  xexpression& operator=(const xexpression& v);
  // Destruktor
  ~xexpression ();
};
```

`xtree_node` ist kein dynamischer Datentyp ??

- Wir haben keine Variablen vom Typ `xtree_node` mit automatischer Speicherdauer
- Copy-Konstruktor, Zuweisungsoperator und Destruktor sind überflüssig
- Speicherverwaltung in der *Container-Klasse*

```
class xexpression {
  // Copy−Konstruktor
  xexpression (const xexpression& v);
  // Zuweisungsoperator
  xexpression& operator=(const xexpression& v);
  // Destruktor
  ~xexpression ();
};
```



Mission: Monolithisch → modular ✓

```
struct tree_node {
  char op;
  double val;
  tree_node* left;
  tree_node* right;
  ...

  double eval () const
  {
    if (op == '=') return val;
    else {
    double l = 0;
    if (left != 0) l = left->eval();
    double r = right->eval();
    if (op == '+') return l + r;
    if (op == '-') return l - r;
    if (op == '*') return l * r;
    if (op == '/') return l / r;
    assert (false); // unknown operator
    return 0;
  }

  int size () const { ... }

  void clear() { ... }

  tree_node* copy () const { ... }

};
```

```
struct number_node: public xtree_node {
  double val;
  ...
  double eval () const {
  return val;
  }
};

struct minus_node: public unary_node {
  ...
  double eval () const {
    return -right->eval();
  }
};

struct minus_node : public binary_node {
  ...
  double eval () const {
    return left->eval() - right->eval();
  }
}
```

+

```
struct abs_node : public unary_node {
  ...
  double eval () const {
    return left->eval() - right->eval();
  }
}
```

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Kapselung

- Verbergen der Implementierungsdetails von Typen (privater Bereich)
- Definition einer Schnittstelle zum Zugriff auf Werte und Funktionalität (öffentlicher Bereich)
- Ermöglicht das Sicherstellen von Invarianten und den Austausch der Implementierung

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Vererbung

- Typen können Eigenschaften von Typen erben.
- Abgeleitete Typen können neue Eigenschaften besitzen oder vorhandene überschreiben.
- Macht Code- und Datenwiederverwendung möglich.

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Polymorphie

- Ein Zeiger kann abhängig von seiner Verwendung unterschiedliche zugrundeliegende Typen haben.
- Die unterschiedlichen Typen können bei gleichem Zugriff auf ihre gemeinsame Schnittstelle verschieden reagieren.
- Macht „nicht invasive” Erweiterung von Bibliotheken möglich.

20. 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) $\mathrm{C}{+}{+}$abhängen
- Ⓢ Sprachlich ($\mathrm{C}{+}{+}$): alles was mit der gewählten Sprache zusammenhängt
- Ⓑ Beispiele: genannte Beispiele der Vorlesung

1. Einführung

Ⓜ
- Euklidischer Algorithmus

Ⓚ
- Algorithmus, Turingmaschine, Programmiersprachen, Kompilation, Syntax und Semantik
- Werte und Effekte, (Fundamental)typen, Literale, Variablen, Bezeichner, Objekte, Ausdrücke, Operatoren, Anweisungen

Ⓢ
- 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;`, Rückgabeanweisung `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./5. Kontrollanweisungen

Ⓜ
- Linearer Kontrollfluss vs. interessante Programme, Spaghetti-Code

Ⓚ
- Auswahlanweisungen, Iterationsanweisungen
- (Vermeidung von) Endlosschleifen, Halteproblem
- Sichtbarkeits- und Gültigkeitsbereich, Automatische Speicherdauer
- Äquivalenz von Iterationsanweisungen

Ⓢ
- 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;`

Ⓑ
- Summenberechnung (Gauss), Primzahltest, Collatz-Folge, Fibonacci Zahlen, Taschenrechner

6./7. Fliesskommazahlen

Ⓜ
- Richtig Rechnen: Celsius / Fahrenheit

Ⓚ
- Fixkomma- vs. Fliesskommazahldarstellung
- (Löcher im) Wertebereich
- Rechnen mit Fliesskommazahlen, Umrechnung
- Fliesskommazahlensysteme, Normalisierung, IEEE Standard 754
- *Richtlinien für das Rechnen mit Fliesskommazahlen*

Ⓢ
- Typen `float`, `double`
- Fliesskommaliterale `1.23e-7f`

Ⓑ
- Celsius/Fahrenheit, Euler, Harmonische Zahlen

8./9. Funktionen

Ⓜ
- Potenzberechnung

Ⓚ
- Kapselung von Funktionalität
- Funktionen, formale Argumente, Aufrufargumente
- Gültigkeitsbereich, Vorwärts-Deklaration
- Prozedurales Programmieren, Modularierung, 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

10. Referenztypen

Ⓜ
- Funktion Swap

Ⓚ
- Werte-/ Referenzsemantik, Call by Value / Call 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

11./12. Felder (Arrays)

Ⓜ
- Iteration über Daten: Feld des Eratosthenes

Ⓚ
- Felder, Speicherlayout, Wahlfreier Zugriff
- (Fehlende) Grenzenprüfung
- Vektoren
- Zeichen: ASCII, UTF8, Texte, Strings

Ⓢ
- Feldtypen `int a[5] = {4,3,5,2,1};`
- Zeichen und Texte, der Typ char `char c = 'a';`, Konversion nach **`int`**
- Mehrdimensionale Felder, Vektoren von Vektoren

Ⓑ
- Sieb des Eratosthenes, Caesar-Code, Kürzeste Wege, Lindenmayer-Systeme

13./14. Zeiger, Iteratoren, Container

Ⓜ
- Arrays als Funktionsargumente

Ⓚ
- Zeiger, Möglichkeiten und Gefahren der Indirektion
- Wahlfreier Zugriff vs. Iteration, Zeiger-Arithmetik
- Container und Iteratoren

Ⓢ
- Zeiger `int* x;`, Konversion Feld → Zeiger, Nullzeiger
- Adress-, Dereferenzoperator `int *ip = &i; int j = *ip;`
- Zeiger und Const `const int *a;`
- Algorithmen und Iteratoren `std::fill (a, a+5, 1);`
- Typdefinitionen `typedef std::set<char>::const_iterator Sit;`

Ⓑ
- Füllen eines Feldes, Buchstabensalat

15./16. Rekursion

Ⓜ
- Rekursive math. Funktionen, Taschenrechner

Ⓚ
- Rekursion
- Aufrufstapel, Gedächtnis der Rekursion
- Korrektheit, Terminierung,
- Rekursion vs. Iteration
- EBNF, Formale Grammatiken, Ströme, Parsen
- Auswertung, Assoziativität

Ⓑ
- Fakultät, GGT, Fibonacci, Berge, Taschenrechner

17. Structs und Klassen I

Ⓜ
- Datentyp Rationale Zahlen selber bauen

Ⓚ
- Heterogene Datenstruktur
- Funktions- und Operator-Overloading
- Datenkapselung

Ⓢ
- 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

Ⓑ
- rationale Zahlen, komplexe Zahlen

18. Klassen, Dynamische Datentypen

Ⓜ
- Rationale Zahlen mit Kapselung, Stack

Ⓚ
- Verkettete Liste, Allokation, Deallokation, Dynamischer Datentyp

Ⓢ
- Klassen `class rational { ... };`
- Zugriffssteuerung `public:` / `private:`
- Mitgliedsfunktionen `int rational::denominator () const`
- Copy-Konstruktor, Destruktor, Dreierregel
- Konstruktoren `rational (int den, int nm): d(den), n(no) {}`
- **`new`** und **`delete`**
- Copy-Konstruktor, Zuweisungs-Operator, Destruktor

Ⓑ
- Verkettete Liste, Stack

19. Baumstrukturen, Vererbung und Polymorphie

Ⓜ
- Ausdrucksbäume,
- Erweiterung von Ausdrucksbäumen
- Vererbung

Ⓚ
- Baumstrukturen
- Vererbung
- Polymorphie

Ⓢ
- Vererbung `class tree_node: public number_node`
- Virtuelle Funktionen `virtual void size() const;`

Ⓑ
- Ausdrucksbaum, Parser auf Ausdrücken, Erweiterung um abs-Knoten

Ende

Ende der Vorlesung.