Informatik

Vorlesung für Rechnergestützte Wissenschaften am D-MATH der ETH Zürich

Felix Friedrich, Malte Schwerhoff

HS 2018

Willkommen

zur Vorlesung Informatik

für RW am MATH Department der ETH Zürich.

Ort und Zeit:

Montag 08:15 - 10:00, CHN C 14.
Pause 9:00 - 9:15, leichte Verschiebung möglich.

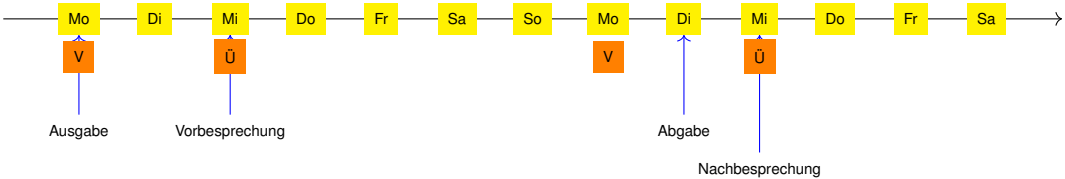
Vorlesungshomepage

`http://lec.inf.ethz.ch/math/informatik_cse`

Team

Chefassistent	Francois Serre
Back-Office	Lucca Hirschi
Assistenten	Manuel Kaufmann Robin Worreby Roger Barton Sebastian Balzer
Dozenten	Dr. Felix Friedrich / Dr. Malte Schwerhoff

Ablauf



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

Zu den Übungen

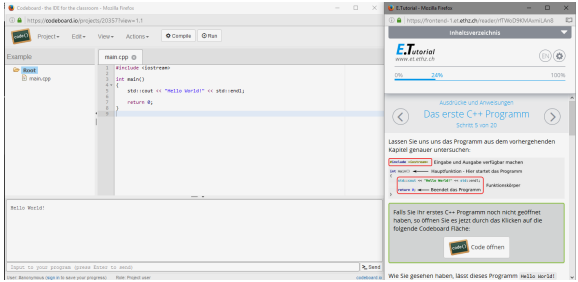
- Bearbeitung der wöchentlichen Uebungsserien ist also freiwillig, wird aber *dringend* empfohlen!

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.

Relevantes für die Prüfung

Prüfungsstoff für die Endprüfung (in der Prüfungssession 2019) schliesst ein

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

Prüfung ist schriftlich und findet (für die RW Studenten) höchstwahrscheinlich am Computer statt.

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

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 von speziell markierten Bonus-Aufgaben, wobei volle Punktzahl einem Bonus von 0.25 entspricht. Die Zulassung zu speziell markierten Bonusaufgaben hängt von der erfolgreichen Absolvierung anderer Übungsaufgaben ab. Der erreichte Notenbonus verfällt, sobald die Vorlesung neu gelesen wird.

Unser Angebot (Konkret)

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

Akademische Lauterkeit

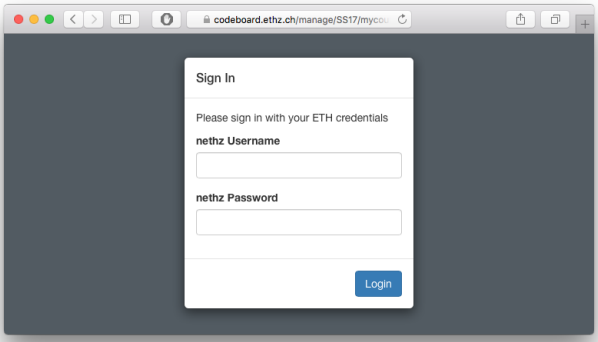
Regel: Sie geben nur eigene Lösungen ab, welche Sie selbst verfasst und verstanden haben.

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

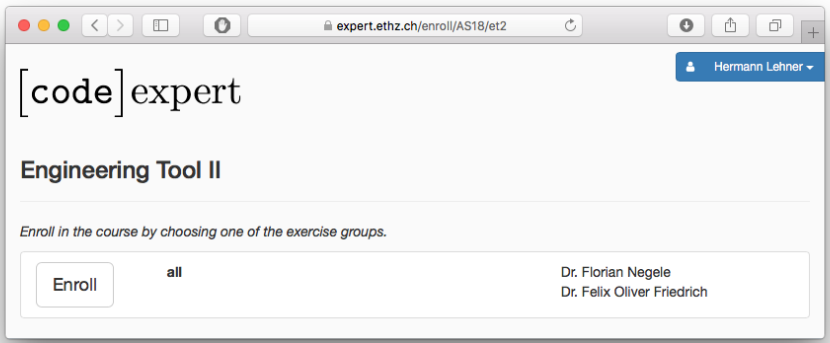
Einschreibung in Übungsgruppen - I

- Besuchen Sie `http://expert.ethz.ch/enroll/AS18/infcse`
- Loggen Sie sich mit Ihrem nethz Account ein.

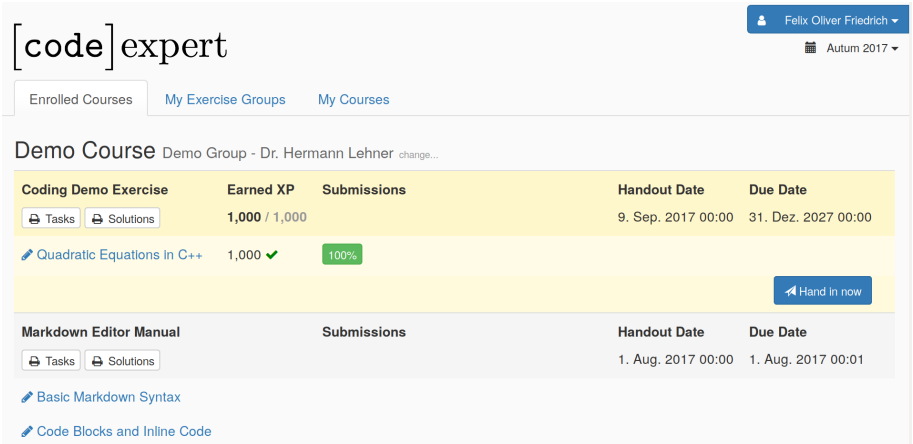


Einschreibung in Übungsgruppen - II

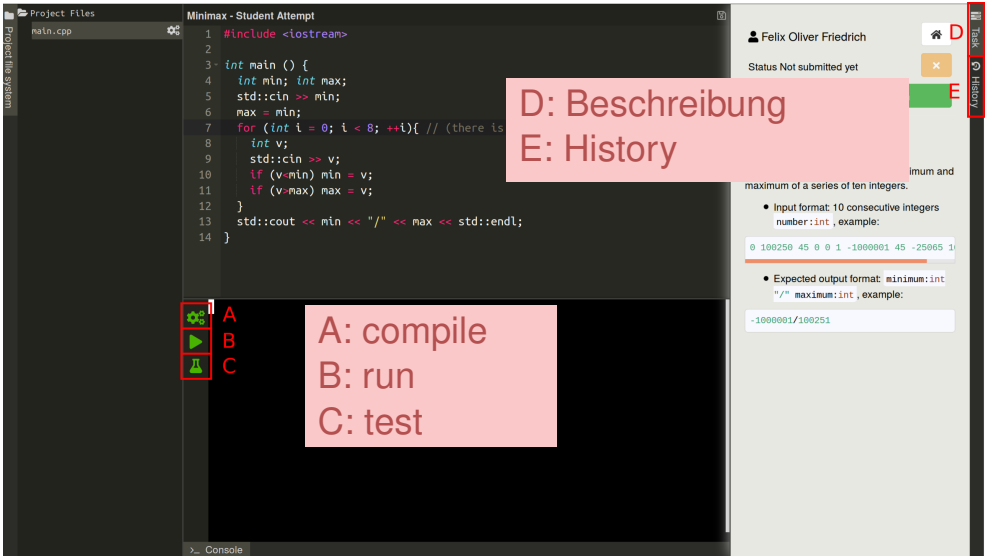
Schreiben Sie sich im folgenden Dialog in eine Übungsgruppe ein.



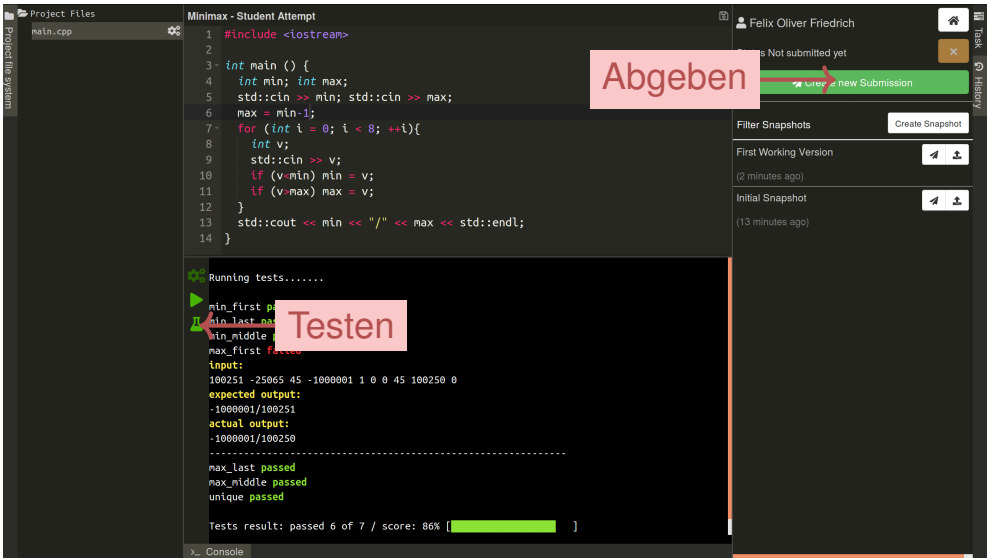
Übersicht


[code]expert
Felix Oliver Friedrich
Autum 2017
Enrolled Courses
My Exercise Groups
My Courses
Demo Course Demo Group - Dr. Hermann Lehner change...
Coding Demo Exercise
Earned XP
Submissions
Handout Date
Due Date
Tasks
Solutions
1,000 / 1,000
9. Sep. 2017 00:00
31. Dez. 2027 00:00
Quadratic Equations in C++
1,000
100%
Hand in now
Markdown Editor Manual
Submissions
Handout Date
Due Date
Tasks
Solutions
1. Aug. 2017 00:00
1. Aug. 2017 00:01
Basic Markdown Syntax
Code Blocks and Inline Code

Programmierübung



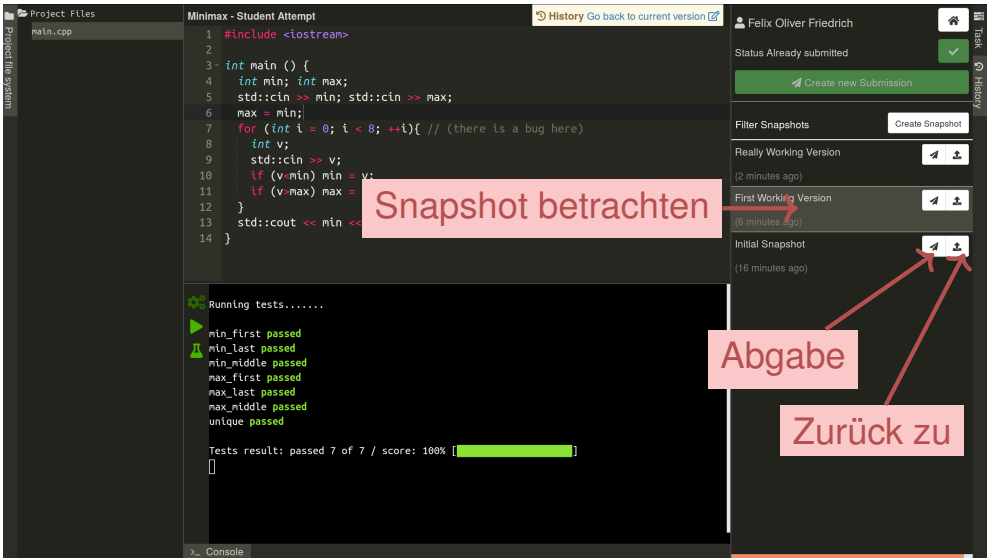
Testen und Abgeben



Wo ist der Save Knopf?

- Das Filesystem ist transaktionsbasiert und es wird laufend gespeichert („autosave“). Beim Öffnen eines Projektes findet man immer den zuletzt gesehenen Zustand wieder.
- Der derzeitige Stand kann als (benannter) *Snapshot* festgehalten werden. Zu gespeicherten Snapshots kann jederzeit zurückgekehrt werden.
- Der aktuelle Stand kann als Snapshot abgegeben (submitted) werden. Zudem kann jeder gespeicherts Snapshot abgegeben werden.

Snapshots


Project Files
main.cpp
Project file system
Minimax - Student Attempt
History Go back to current version
#include <iostream>
int main () {
int min; int max;
std::cin >> min; std::cin >> max;
max = min;
for (int i = 0; i < 8; ++i){ // (there is a bug here)
int v;
std::cin >> v;
if (v<min) min = v;
if (v>max) max =
}
std::cout << min <<
}
Running tests.....
min_first passed
min_last passed
min_middle passed
max_first passed
max_last passed
max_middle passed
unique passed
Tests result: passed 7 of 7 / score: 100% [
Console
Felix Oliver Friedrich
Status Already submitted
Create new Submission
Filter Snapshots
Create Snapshot
Really Working Version
(2 minutes ago)
First Working Version
(6 minutes ago)
Initial Snapshot
(16 minutes ago)
Task
History
Snapshot betrachten
Abgabe
Zurück zu

Literatur

- Der Kurs ist ausgelegt darauf, selbsterklärend zu sein.
- Vorlesungsskript gemeinsam mit Informatik für Mathematiker und Physiker. Insbesondere erster Teil.
- Empfehlenswerte Literatur
  - B. Stroustrup. *Einführung in die Programmierung mit C++*, Pearson Studium, 2010.
  - B. Stroustrup, *The C++ Programming Language* (4th Edition) Addison-Wesley, 2013.
  - A. Koenig, B.E. Moo, *Accelerated C++*, Adddison Wesley, 2000.
  - B. Stroustrup, *The design and evolution of C++*, Addison-Wesley, 1994.

Credits

- Vorlesung:
  - Ursprüngliche Fassung von Prof. B. Gärtner und Dr. F. Friedrich
  - Mit Änderungen von Dr. F. Friedrich, Dr. H. Lehner, Dr. M. Schwerhoff
- Skript: Prof. B. Gärtner
- Code Expert: Dr. H. Lehner, David Avanthay und anderen

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?

- Die Wissenschaft der **systematischen Verarbeitung von Informationen**,. . .
- . . . insbesondere der automatischen Verarbeitung mit Hilfe von Digitalrechnern.

(Wikipedia, nach dem „Duden Informatik“)

Informatik vs. Computer

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

Mike Fellows, US-Informatiker (1991)

http://larc.unt.edu/ian/research/cseducation/fellows1991.pdf

Informatik vs. Computer

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

Informatik $\neq$ EDV-Kenntnisse

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

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

Informatik: *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
- 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 $b \neq 0$

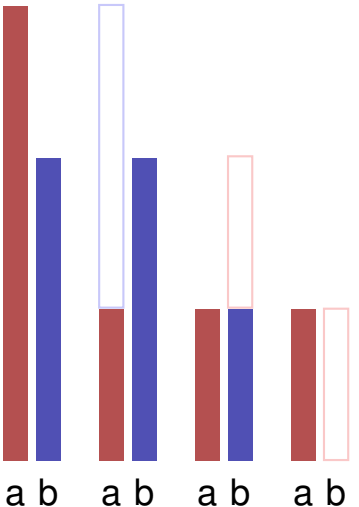
Wenn $a > b$ dann

$a \leftarrow a - b$

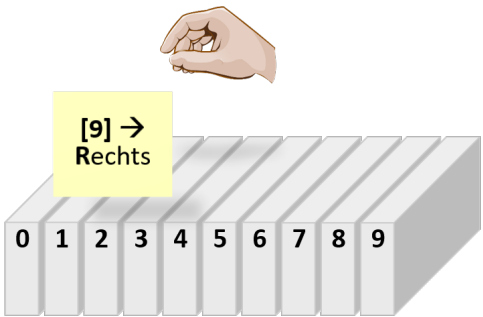
Sonst:

$b \leftarrow b - a$

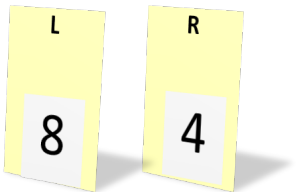
Ergebnis: a.



Live Demo: Turing Maschine



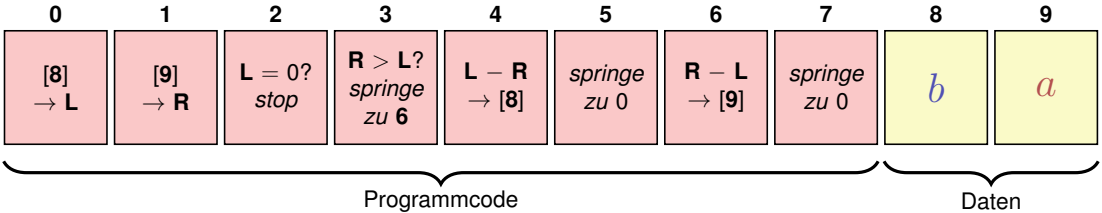
Speicher

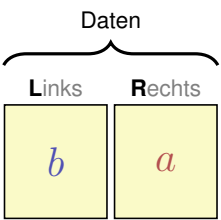


Register

Euklid in der Box

Speicher





Register

Solange $b \neq 0$

Wenn $a > b$ dann

$a \leftarrow a - b$

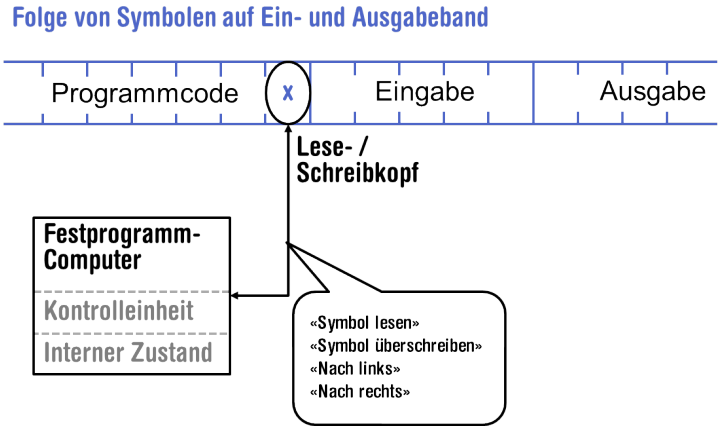
Sonst:

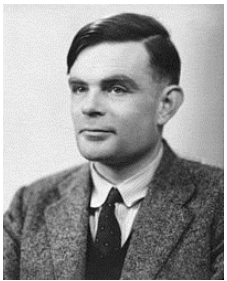
$b \leftarrow b - a$

Ergebnis: a.

Computer – Konzept

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





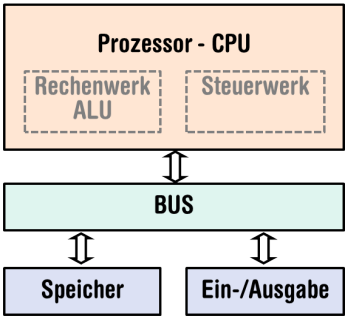
Alan Turing

http://en.wikipedia.org/wiki/Alan_Turing

Computer – Umsetzung

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

Von Neumann Architektur



Konrad Zuse

John von Neumann

http://www.hs.uni-hamburg.de/DE/GNT/hh/biogr/zuse.htm
http://commons.wikimedia.org/wiki/File:John_von_Neumann.jpg

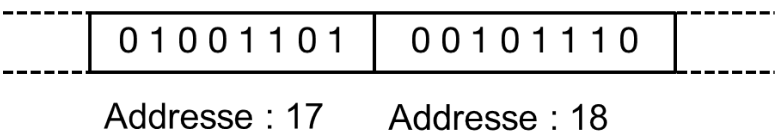
Computer

Zutaten der *Von Neumann Architektur:*

- Hauptspeicher (RAM) für Programme *und* Daten
- Prozessor (CPU) zur Verarbeitung der Programme und Daten
- I/O Komponenten zur Kommunikation mit der Aussenwelt

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



Prozessor

Der Prozessor (CPU)

- führt Befehle in Maschinensprache aus
- hat eigenen "schnellen" Speicher (Register)
- kann vom Hauptspeicher lesen und in ihn schreiben
- beherrscht eine Menge einfachster Operationen (z.B. Addieren zweier Registerinhalte)

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.[1]

[1] 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

http://en.wikipedia.org/wiki/Harvard_Computers

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

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

Darum Programmieren!

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

Programmiersprachen

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

Höhere Programmiersprachen

darstellbar als Programmtext, der

- von Menschen *verstanden* werden kann
- vom Computermodell *unabhängig* ist
  $\rightarrow$ Abstraktion!

Programmiersprachen – Einordnung

Unterscheidung in

- *Kompilierte* vs. interpretierte Sprachen
  - $C++$, C#, Pascal, Modula, Oberon, Java, Go
    vs.
    Python, Tcl, Javascript, Matlab
- *Höhere* Programmiersprachen vs. Assembler.
- *Mehrzweck*sprachen vs. zweckgebundene Sprachen.
- *Prozedurale, Objekt-Orientierte,* Funktionsorientierte und logische Sprachen.

Warum C++?

Andere populäre höhere Programmiersprachen: Java, C#, Objective-C, Oberon, Javascript, Go, 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

B. Stroustrup, Design and Evolution of C++, Kap. 4.5

Warum C++?

- C++versieht C mit der Mächtigkeit der Abstraktion einer höheren Programmiersprache
- In diesem Kurs: C++ als Hochsprache eingeführt (nicht als besseres C)
- Vorgehen: Traditionell prozedural $\rightarrow$ objekt-orientiert

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

C++: Fehlerarten illustriert an deutscher Sprache

Das Auto fuhr zu schnell.	Syntaktisch und semantisch korrekt.
DasAuto fuh r zu sxhnell.	Syntaxfehler: Wortbildung.
Rot das Auto ist.	Syntaxfehler: Satzstellung.
Man empfiehlt dem Dozenten nicht zu widersprechen	Syntaxfehler: Satzzeichen fehlen .
Sie ist nicht gross und rothaarig.	Syntaktisch korrekt aber mehrdeutig. [kein Analogon]
Die Auto ist rot.	Syntaktisch korrekt, doch semantisch fehlerhaft: Falscher Artikel. [Typfehler]
Das Fahrrad galoppiert schnell.	Syntaktisch und grammatikalisch korrekt! Semantisch fehlerhaft. [Laufzeitfehler]
Manche Tiere riechen gut.	Syntaktisch und semantisch korrekt. Semantisch mehrdeutig. [kein Analogon]

Syntax und Semantik von C++

Syntax

- Wann ist ein Text ein C++Programm?
- D.h. ist es *grammatikalisch* korrekt?

Semantik

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

Syntax und Semantik von C++

Der ISO/IEC Standard 14822 (1998, 2011, 2014, ...)

- ist das „Gesetz“ von C++
- legt Grammatik und Bedeutung von C++-Programmen fest
- wird seit 2011 regelmässig durch Neuerungen für *fortgeschrittenes* Programmieren erweitert

Was braucht es zum Programmieren?

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

Sprachbestandteile am Beispiel

- Kommentare/Layout
- Include-Direktiven
- Die main-Funktion
- Werte, Effekte
- Typen, Funktionalität
- Literale
- Variablen
- Konstanten
- Bezeichner, Namen
- Objekte
- **Ausdrücke**
- L- und R-Werte
- Operatoren
- Anweisungen

Das erste C++ Programm Wichtigste Bestandteile...

```
// 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; ←——— Anweisungen: Mache etwas (lies a ein)!
    // computation
    int b = a * a; // b = a^2 ←——— Ausdrücke: 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;
}
```

Verhalten eines Programmes

Zur Compilationszeit:

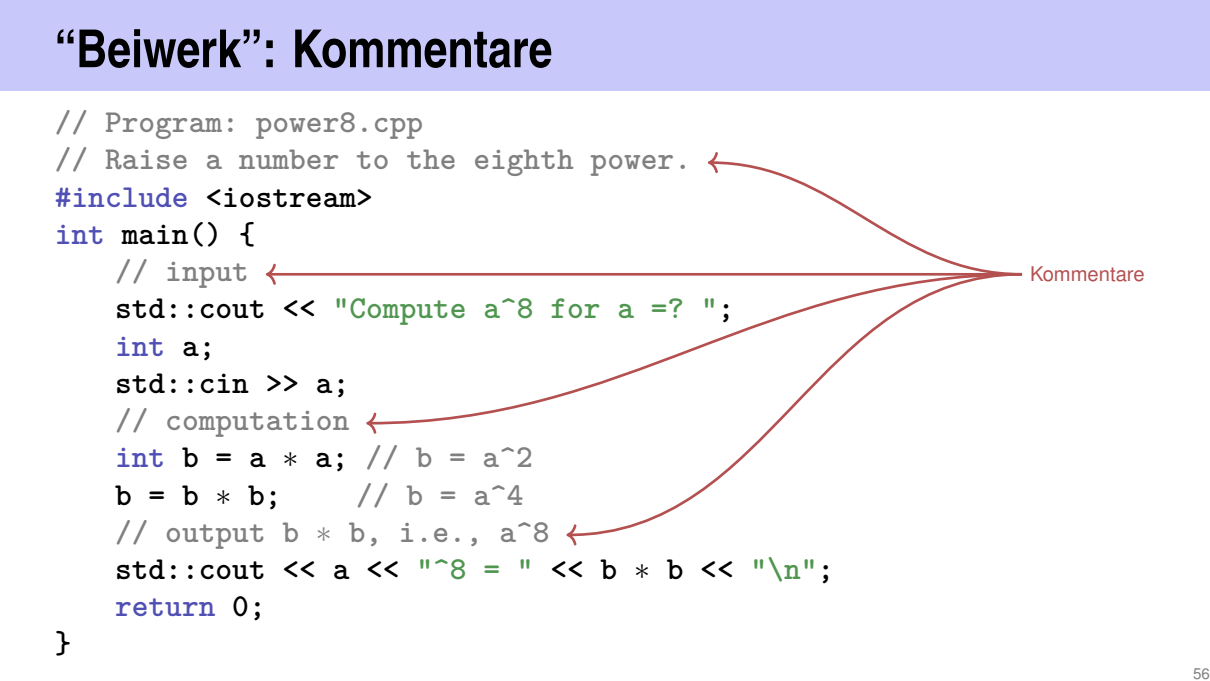
- vom Compiler akzeptiertes Programm (syntaktisch korrektes C++)
- Compiler-Fehler

Zur Laufzeit:

- korrektes Resultat
- inkorrektes Resultat
- Programmabsturz
- Programm *terminiert* nicht (Endlosschleife)

“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

Kommentare

- hat jedes gute Programm,
- dokumentieren, *was* das Programm *wie* macht und wie man es verwendet und
- werden vom Compiler ignoriert.
- Syntax: „Doppelslash“ `// bis Zeilenende.`

Ignoriert werden vom Compiler ausserdem

- Leerzeilen, Leerzeichen,
- Einrückungen, die die Programmlogik widerspiegeln (sollten)

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;
}
```

Include-Direktive

Funktionsdeklaration der main-Funktion

Include-Direktiven

C++ besteht aus

- Kernsprache
- Standardbibliothek
  - Ein/Ausgabe (Header iostream)
  - Mathematische Funktionen (cmath)
  - ...

```
#include <iostream>
```

- macht Ein/Ausgabe verfügbar

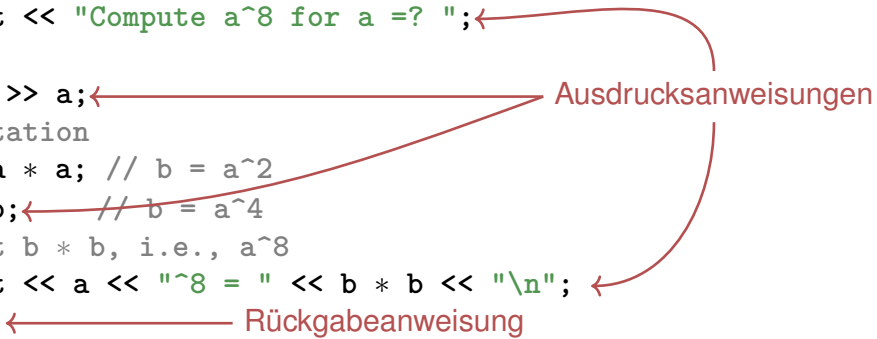
Die Hauptfunktion

Die `main`-Funktion

- existiert in jedem C++ Programm
- wird vom Betriebssystem aufgerufen
- wie eine mathematische Funktion ...
  - Argumente (bei `power8.cpp`: keine)
  - Rückgabe**wert** (bei `power8.cpp`: 0)
- ... aber mit zusätzlichem *Effekt*.
  - Lies eine Zahl ein und gib die 8-te Potenz aus.

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

- Bausteine eines C++ Programms
- werden (sequenziell) *ausgeführt*
- enden mit einem Semikolon
- Jede Anweisung hat (potenziell) einen *Effekt*.

Ausdrucksanweisungen

- haben die Form

  expr;

  wobei *expr* ein Ausdruck ist
- Effekt ist der Effekt von *expr*, der Wert von *expr* wird ignoriert.

Beispiel: `b = b*b;`

Rückgabeanweisungen

- treten nur in Funktionen auf und sind von der Form

  `return` expr;

  wobei *expr* ein Ausdruck ist
- spezifizieren Rückgabewert der Funktion

Beispiel: `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

Werte und Effekte

- bestimmen, was das Programm macht,
- sind rein semantische Konzepte:
  - Zeichen `0` bedeutet Wert $0 \in \mathbb{Z}$
  - `std::cin >> a;` bedeutet Effekt "Einlesen einer Zahl"
- hängen vom Programmzustand (Speicherinhalte / Eingaben) ab

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

Deklarationsanweisungen

- führen neue Namen im Programm ein,
- bestehen aus Deklaration + Semikolon

  Beispiel: `int a;`
- können Variablen auch initialisieren

  Beispiel: `int b = a * a;`

Typen und Funktionalität

`int`:

- $\mathrm{C}{+}{+}$ Typ für ganze Zahlen,
- entspricht $(\mathbb{Z}, +, \times)$ in der Mathematik

In $\mathrm{C}{+}{+}$ hat jeder Typ einen Namen sowie

- Wertebereich (z.B. ganze Zahlen)
- Funktionalität (z.B. Addition/Multiplikation)

Fundamentaltypen

C++ enthält fundamentale Typen für

- Ganze Zahlen (`int`)
- Natürliche Zahlen (`unsigned int`)
- Reelle Zahlen (`float`, `double`)
- Wahrheitswerte (`bool`)
- ...

Literale

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

Beispiele:

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

Variablen

- repräsentieren (wechselnde) Werte
- haben
  - *Name*
  - *Typ*
  - *Wert*
  - *Adresse*
- sind im Programmtext „sichtbar“

Beispiel

`int a;` definiert Variable mit

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

Objekte

- repräsentieren Werte im Hauptspeicher
- haben *Typ*, *Adresse* und *Wert* (Speicherinhalt an der Adresse),
- können benannt werden (Variable) ...
- ... aber auch anonym sein.

Anmerkung

Ein Programm hat eine *feste* Anzahl von Variablen. Um eine variable Anzahl von Werten behandeln zu können, braucht es „anonyme“ Adressen, die über temporäre Namen angesprochen werden können ($\rightarrow$ Informatik 1).

Bezeichner und Namen

(Variablen-)Namen sind Bezeichner:

- erlaubt: A,...,Z; a,...,z; 0,...,9;_
- erstes Zeichen ist Buchstabe.

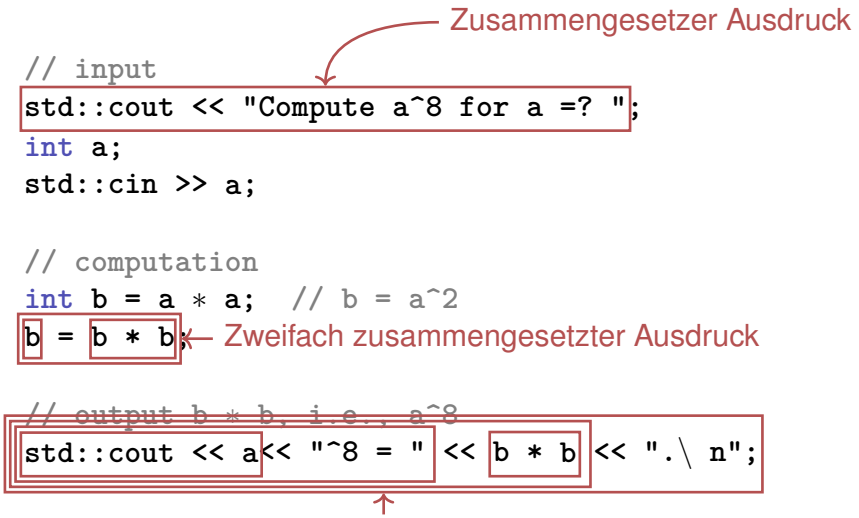
Es gibt noch andere Namen:

- `std::cin` (qualifizierter Name)

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


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

Ausdrücke (Expressions)

- repräsentieren *Berechnungen*,
- sind *primär* oder *zusammengesetzt* (aus anderen Ausdrücken und Operationen)

`a * a`
zusammengesetzt aus
Variablenname, Operatorsymbol, Variablenname
Variablenname: primärer Ausdruck

- können geklammert werden

`a * a` ist äquivalent zu `(a * a)`

Ausdrücke (Expressions)

haben *Typ*, *Wert* und *Effekt* (potenziell).

Beispiel

`a * a`

- Typ: `int` (Typ der Operanden)
- Wert: Produkt von `a` und `a`
- Effekt: keiner.

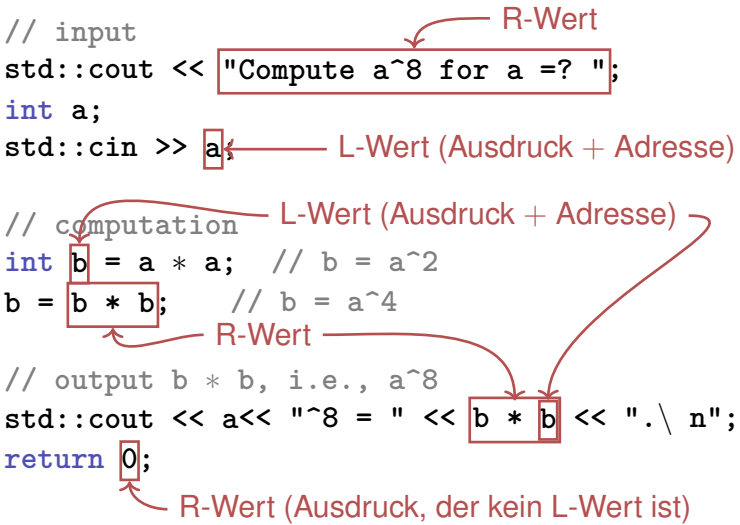
Beispiel

`b = b * b`

- Typ: `int` (Typ der Operanden)
- Wert: Produkt von `b` und `b`
- Effekt: Weise `b` diesen Wert zu.

Typ eines Ausdrucks ist fest, aber Wert und Effekt werden erst durch die *Auswertung* des Ausdrucks bestimmt.

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-Wert kann seinen Wert ändern (z.B. per Zuweisung).

Beispiel: Variablenname

L-Werte und R-Werte

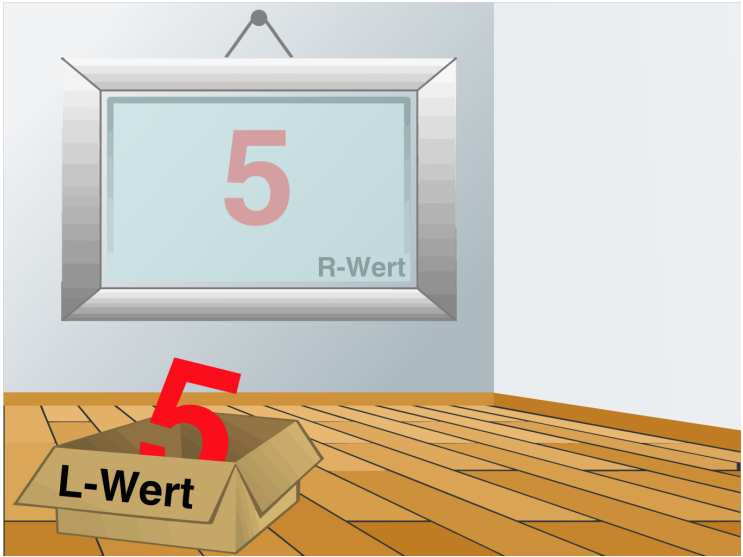
R-Wert (“**R**echts vom Zuweisungsoperator”)

- Ausdruck der kein L-Wert ist

  Beispiel: Literal `0`

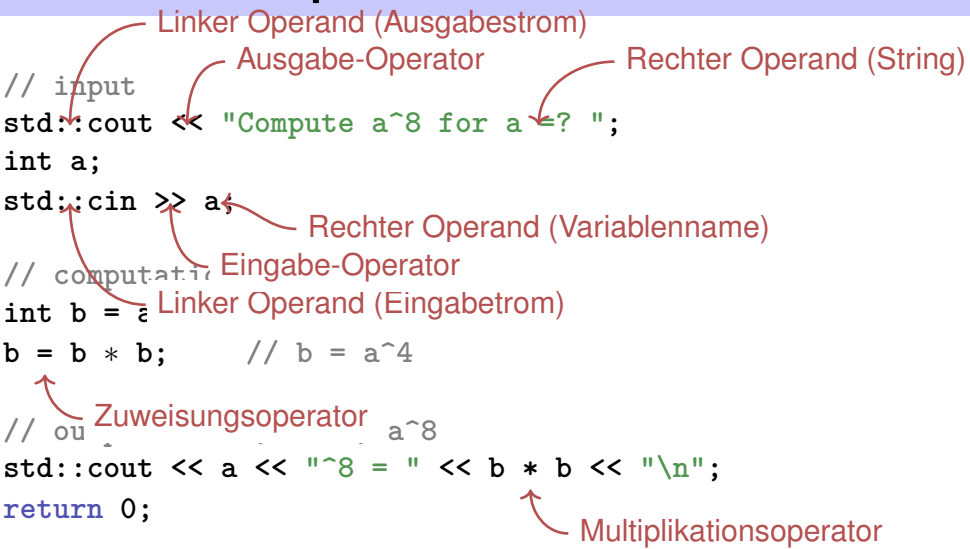
- Jeder L-Wert kann als R-Wert benutzt werden (aber nicht umgekehrt).
- Ein R-Wert kann seinen Wert *nicht ändern*.

L-Werte und R-Werte



Operatoren und Operanden

Baukasten



Operatoren

Operatoren

- machen aus Ausdrücken (*Operanden*) neue zusammengesetzte Ausdrücke
- spezifizieren für die Operanden und das Ergebnis die Typen, und ob sie L- oder R-Werte sein müssen
- haben eine Stelligkeit

Multiplikationsoperator `*`

- erwartet zwei R-Werte vom gleichen Typ als Operanden (Stelligkeit 2)
- "gibt Produkt als R-Wert des gleichen Typs zurück", das heisst formal:
  - Der zusammengesetzte Ausdruck ist ein R-Wert; sein Wert ist das Produkt der Werte der beiden Operanden

Beispiele: `a * a` und `b * b`

Zuweisungsoperator =

- **L**inker Operand ist **L**-Wert,
- **R**echter Operand ist **R**-Wert des gleichen Typs.
- Weist linkem Operanden den Wert des rechten Operanden zu und gibt den linken Operanden als L-Wert zurück

Beispiele: `b = b * b` und `a = b`

Vorsicht, Falle!

Der Operator = entspricht dem Zuweisungsoperator in der Mathematik ($:=$), nicht dem Vergleichsoperator ($=$).

Eingabeoperator >>

- linker Operand ist L-Wert (*Eingabestrom*)
- rechter Operand ist L-Wert
- weist dem rechten Operanden den nächsten Wert aus der Eingabe zu, *entfernt ihn aus der Eingabe* und gibt den Eingabestrom als L-Wert zurück

Beispiel: `std::cin >> a` (meist Tastatureingabe)

- Eingabestrom wird verändert und muss deshalb ein L-Wert sein!

Ausgabeoperator <<

- linker Operand ist L-Wert (*Ausgabestrom*)
- rechter Operand ist R-Wert
- gibt den Wert des rechten Operanden aus, fügt ihn dem Ausgabestrom hinzu und gibt den Ausgabestrom als L-Wert zurück

  Beispiel: `std::cout << a` (meist Bildschirmausgabe)
- Ausgabestrom wird verändert und muss deshalb ein L-Wert sein!

Ausgabeoperator <<

Warum Rückgabe des Ausgabestroms?

- erlaubt Bündelung von Ausgaben

```
std::cout << a << "^8 = " << b * b << "\n"
```

ist wie folgt logisch geklammert

```
((((std::cout << a) << "^8 = ") << b * b) << "\n")
```

- `std::cout << a` dient als linker Operand des nächsten << und ist somit ein L-Wert, der kein Variablenname ist.

`power8.cpp`

- Problem mit `power8.cpp`: grosse Eingaben werden nicht korrekt behandelt
- Grund: Wertebereich des Typs int ist beschränkt
- Mehr dazu erfahren Sie nächste Woche

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;
}
```

`9 * celsius / 5 + 32`

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

Wie ist der Ausdruck geklammert?

Präzedenz

Punkt vor Strichrechnung

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

bedeutet

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

Regel 1: Präzedenz

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

Assoziativität

Von links nach rechts

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

bedeutet

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

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

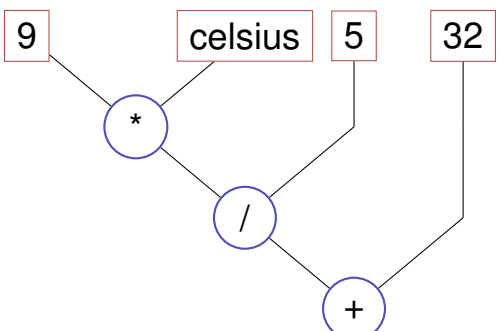
- Assoziativitäten
- Präzedenzen
- Stelligkeiten (Anzahl Operanden)

der beteiligten Operatoren eindeutig geklammert werden (Details im Skript).

Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

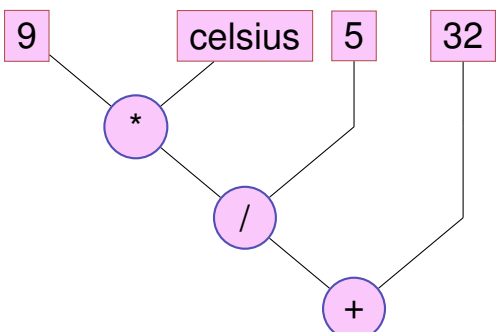
```
(((9 * celsius) / 5) + 32)
```



Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

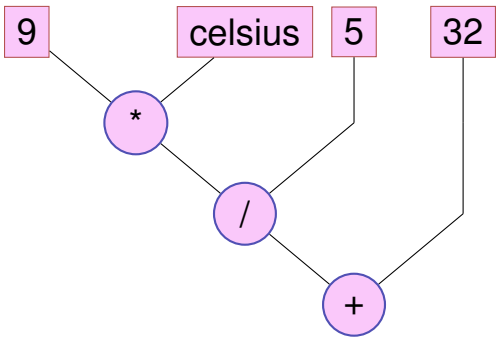
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

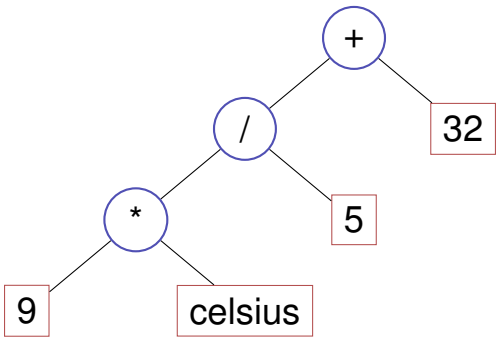
```
9 * celsius / 5 + 32
```



Ausdrucksbäume – Notation

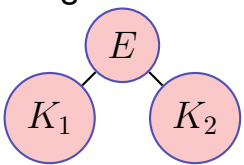
Üblichere Notation: Wurzel oben

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



Auswertungsreihenfolge – formaler

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



In C++ ist die anzuwendende gültige Reihenfolge nicht spezifiziert.

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

Auswertungsreihenfolge

Richtlinie

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

Arithmetische Operatoren

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

Alle Operatoren: [R-Wert $\times$] R-Wert $\rightarrow$ R-Wert

Einschub: 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` $\Longleftrightarrow$ `a = (b = c)`

Beispiel Mehrfachzuweisung:
`a = b = 0` $\Longrightarrow$ `b=0; a=0`

Division

- Operator `/` realisiert ganzzahlige Division

  `5 / 2` hat Wert 2

- In `fahrenheit.cpp`

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

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

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

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

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

Präzisionsverlust

Richtlinie

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

Division und Modulo

- Modulo-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.
```

Nachteile

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

In-/Dekrement Operatoren

Post-Inkrement

```
expr++
```

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

Prä-Inkrement

```
++expr
```

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

Post-Dekrement

```
expr--
```

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

Prä-Dekrement

```
--expr
```

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

In-/Dekrement Operatoren

	Gebrauch	Stelligkeit	Präz	Assoz.	L/R-Werte
Post-Inkrement	`expr++`	1	17	links	L-Wert $\rightarrow$ R-Wert
Prä-Inkrement	`++expr`	1	16	rechts	L-Wert $\rightarrow$ L-Wert
Post-Dekrement	`expr--`	1	17	links	L-Wert $\rightarrow$ R-Wert
Prä-Dekrement	`--expr`	1	16	rechts	L-Wert $\rightarrow$ L-Wert

In-/Dekrement Operatoren

Beispiel

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

In-/Dekrement Operatoren

Ist die Anweisung

`++expr;` ← wir bevorzugen dies

äquivalent zu

`expr++;`?

Ja, aber

- Prä-Inkrement ist effizienter (alter Wert muss nicht gespeichert werden)
- Post-In/Dekrement sind die einzigen linksassoziativen unären Operatoren (nicht sehr intuitiv)

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
```

Analog für `-`, `*`, `/` und `%`

Arithmetische Zuweisungen

	Gebrauch	Bedeutung
`+=`	`expr1 += expr2`	`expr1 = expr1 + expr2`
`-=`	`expr1 -= expr2`	`expr1 = expr1 - expr2`
`*=`	`expr1 *= expr2`	`expr1 = expr1 * expr2`
`/=`	`expr1 /= expr2`	`expr1 = expr1 / expr2`
`%=`	`expr1 %= expr2`	`expr1 = expr1 % expr2`

Arithmetische Zuweisungen werten expr1 nur einmal aus.

Zuweisungen haben Präzedenz 4 und sind rechtsassoziativ

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, 26. August 1959 Blatt 5
Mittagausgabe Nr. 2736 (59)

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 dieses 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 *Instituts 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 beiderseits genau dieselbe 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 Schalttischreihen und einem Speicher für 64 Zahlen, welche 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 erhöht 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 das bedeutet ja $1 \cdot 2^1 + 0 \cdot 2^0 = 2$. 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 *LOLAL* 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 *OLL* wie folgt übersetzt:

$$1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1 + 1/4 + 1/8 = 1{,}375$$

Der große Vorteil, der das Dualsystem für Rechenmaschinen so geeignet macht, besteht die Reduktion der Anzahl der verwendeten Symbole auf nur zwei, wird aber durch einen 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 erinnert, 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 werden, 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.

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

Binäre Zahlen: Zahlen der Computer?

Klischee: Computer reden 0/1-Kauderwelsch.

01001110 01011010 01011010

Freitag, 8. Juni 2012 · Nr. 131 · 233. Jhg. — *01001010 01010110 01001101* — www.nzz.ch · Fr. 4.00 · €3.50

01000010 01100101 01110010 01101001

01100011 01101000 01110100 01100101

00100000 11111100 01100010 01100101 01110010 00100000 01101110 01100101 01110101 011-00101 01110011 00100000 010-01101 01100001 01110011

01110011 01100001

01100110 01100101

00100000 01110110

01000110 01101100 11111100

Rechentricks

- Abschätzung der Grössenordnung von Zweierpotenzen[2]:

$$2^{10} = 1024 = 1\text{Ki} \approx 10^3.$$
$$2^{20} = 1\text{Mi} \approx 10^6,$$
$$2^{30} = 1\text{Gi} \approx 10^9,$$
$$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi}.$$
$$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}.$$

[2] 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

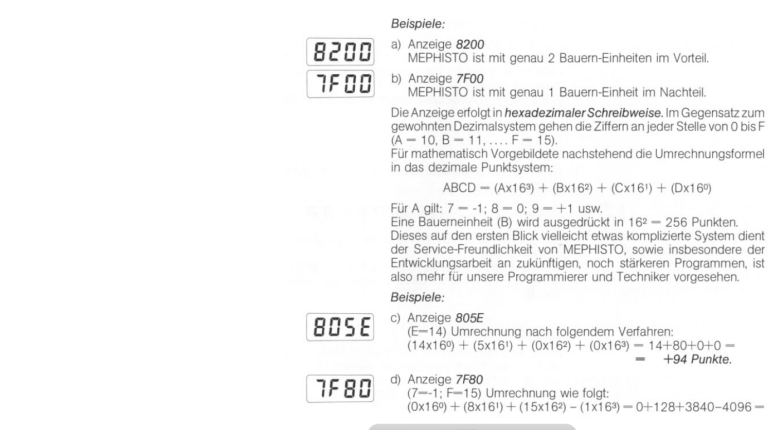
Wozu Hexadezimalzahlen?

- Ein Hex-Nibble entspricht *genau* 4 Bits. Die Zahlen 1, 2, 4 und 8 repräsentieren Bits 0, 1, 2 und 3.
- „Kompakte Darstellung von Binärzahlen“.

32-bit Zahlen bestehen aus acht Hex-Nibbles: `0x00000000 -- 0xffffffff` .
`0x400` $= 1Ki = 1'024$.
`0x100000` $= 1Mi = 1'048'576$.
`0x40000000` $= 1Gi = 1'073.741,824$.
`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?

„Für Programmierer und Techniker“ (Auszug aus der Bedienungsanleitung des Schachcomputers *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:

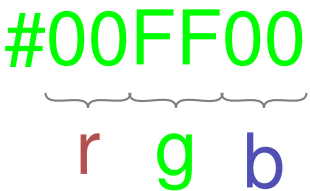
805E

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:
$(0x16^0) + (8x16^1) + (15x16^2) - (1x16^3) = 0+128+3840-4096 =$

Beispiel: Hex-Farben



Wozu Hexadezimalzahlen?

Die NZZ hätte viel Platz sparen können...

Wertebereich des Typs `int`

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

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

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

Woher kommen diese Zahlen?

Wertebereich des Typs `int`

- Repräsentation mit B Bits. Wertebereich umfasst die 2^B ganzen Zahlen:

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

Woher kommt gerade diese Aufteilung?

- Auf den meisten Plattformen $B = 32$
- Für den Typ `int` garantiert C++ $B \geq 16$
- Hintergrund: Abschnitt 2.2.8 (Binary Representation) im Skript

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 “andersherum”

Die Deklaration

```
int a = 3u;
```

konvertiert `3u` nach `int`.

Der Wert bleibt erhalten, weil er im Wertebereich von `int` liegt; andernfalls ist das Ergebnis implementierungsabhängig.

Vorzeichenbehaftete Zahlendarstellung

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

$$[b_{31}b_{30}\ldots b_0]_u \quad \widehat{=} \quad b_{31}\cdot 2^{31} + b_{30}\cdot 2^{30} + \cdots + b_0$$

- Nun offensichtlich notwendig: Verwende ein Bit für das Vorzeichen.
- Suche möglichst konsistente Lösung

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

Rechnen mit Binärzahlen (4 Stellen)

Einfache Addition

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

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

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

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

1	0001
$+(-1)$	1111
0	$(1)0000$

Nutzen das aus:

3	0011
$+?$	$+????$
-1	1111

Rechnen mit Binärzahlen (4 Stellen)

Invertieren!

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

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

Rechnen mit Binärzahlen (4 Stellen)

- Negation: Inversion und Addition von 1

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

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

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

Warum das funktioniert

Modulo-Arithmetik: Rechnen im Kreis[3]

$11 \equiv 23 \equiv -1 \equiv \ldots \mod 12$

$4 \equiv 16 \equiv \ldots \mod 12$

$3 \equiv 15 \equiv \ldots \mod 12$

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

Negative Zahlen (3 Stellen)

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

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

Zweierkomplement

- Negation durch bitweise Negation und Addition von 1.

$$-2 = -[0010] = [1101] + [0001] = [1110]$$

  - Arithmetik der Addition und Subtraktion *identisch* zur vorzeichenlosen Arithmetik.

$$3 - 2 = 3 + (-2) = [0011] + [1110] = [0001]$$

  - Intuitive „Wrap-Around“ Konversion negativer Zahlen.

$$-n \to 2^B - n$$

  - Wertebereich: $-2^{B-1} \ldots 2^{B-1} - 1$

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";
```

Verhalten hängt ab vom Wert eines Booleschen Ausdrucks

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*
- Literale `false` und `true`
- Wertebereich $\{$*false*, *true*$\}$

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

Relationale Operatoren

`a < b`	(kleiner als)
`a >= b`	(grösser gleich)
`a == b`	(gleich)
`a != b`	(ungleich)

Zahlentyp $\times$ Zahlentyp $\rightarrow$ `bool`

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

Relationale Operatoren: Tabelle

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Kleiner	<	2	11	links
Grösser	>	2	11	links
Kleiner gleich	<=	2	11	links
Grösser gleich	>=	2	11	links
Gleich	==	2	10	links
Ungleich	!=	2	10	links

Zahlentyp $\times$ Zahlentyp $\rightarrow$ `bool`

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

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“.
- 1 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)

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

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

```
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“.
- 1 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)

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

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

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

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

- „Logisches Nicht“

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

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

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

Logischer Operator !

`!b` (logisches Nicht)

`bool` $\rightarrow$ `bool`

R-Wert $\rightarrow$ R-Wert

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

Präzedenzen

```
!b && a
```

```
(!b) && a
```

```
a && b || c && d
```

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

```
a || b      &&      c || d
```

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

Logische Operatoren: Tabelle

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Logisches Und (AND)	`&&`	2	6	links
Logisches Oder (OR)	`||`	2	5	links
Logisches Nicht (NOT)	`!`	1	16	rechts

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

Vollständigkeit

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

$x \oplus y$

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

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

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

Vollständigkeit Beweis

- Identifiziere binäre Boolesche Funktionen mit ihrem charakteristischen Vektor.

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

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

- Schritt 3: erzeuge f_{0000}

$$f_{0000} = 0.$$

`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)`
- `!(a || b) == (!a && !b)`

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

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.

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

$\Rightarrow$ Keine Division durch 0

4. Defensives Programmieren

Konstanten und Assertions

Fehlerquellen

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

Der Compiler als Freund: Konstanten

Konstanten

- sind Variablen mit unveränderbarem Wert

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

- Verwendung: `const` vor der Definition

Der Compiler als Freund: Konstanten

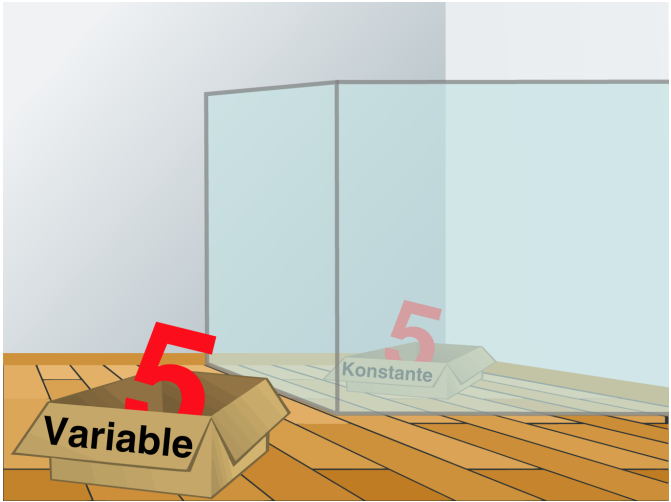
- Compiler kontrolliert Einhaltung des `const`-Versprechens

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

Compilerfehler!

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

Konstanten: Variablen hinter Glas



Die `const`-Richtlinie

`const`-Richtlinie

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

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

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens
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
- benötigt `#include <cassert>`
- kann abgeschaltet werden (potentieller Geschwindigkeitsgewinn)

Assertions für den $ggT(x, y)$

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

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

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

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

Eingabe der Argumente für die Berechnung

Vorbedingung für die weitere Berechnung

Assertions für den $ggT(x, y)$

... und hinterfrage das Offensichtliche! ...

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

Vorbedingung für die weitere Berechnung

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

assert (a >= 1);
assert (x % a == 0 && y % a == 0);
for (int i = a+1; i <= x && i <= y; ++i)
  assert(!(x % i == 0 && y % i == 0));
```

Verschiedene Eigenschaften des ggT überprüfen

Assertions abschalten

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

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

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

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

Fail-Fast mit Assertions

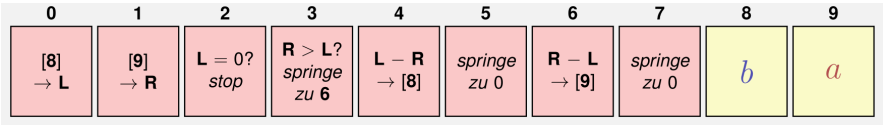
- Reale Software: viele C++-Dateien, komplexer Kontrollfluss
- Fehler machen sich erst spät(er) bemerkbar $\rightarrow$ Fehlersuche erschwert
- Assertions: Fehler frühzeitig bemerken

5. Kontrollanweisungen I

Auswahlanweisungen, Iterationsanweisungen, Terminierung, Blöcke

Kontrollfluss

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



Auswahlanweisungen

realisieren Verzweigungen

- `if` Anweisung
- `if-else` Anweisung

`if`-Anweisung

`if` (*condition*)
statement

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

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

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

`if-else`-Anweisung

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

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

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

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

Layout!

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

Iterationsanweisungen

realisieren „Schleifen“:

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

Berechne $1 + 2 + \ldots + n$

```
// Program: sum_n.cpp
// Compute the sum of the first n natural numbers.

#include <iostream>

int main()
{
  // 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";
  return 0;
}
```

`for`-Anweisung am Beispiel

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

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

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

`for`-Anweisung: Syntax

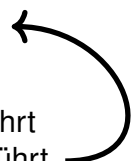
`for` (*init statement*`;` *condition*`;` *expression*)
    body statement

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

`for`-Anweisung: Semantik

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

- *init-statement* wird ausgeführt
- *condition* wird ausgewertet
  - `true`: Iteration beginnt
    statement wird ausgeführt
    expression wird ausgeführt
  - falsch: `for`-Anweisung wird beendet.



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

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

- Das ist die Hälfte von

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

- 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.
- 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.

- ... aber nicht automatisch zu erkennen.

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

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.

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

[4] 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.

Eine Schleife, die das testet:

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

Primzahltest: Terminierung

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

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

Primzahltest: Korrektheit

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

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

Blöcke

- Blöcke gruppieren mehrere Anweisungen zu einer neuen Anweisung

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

- Beispiel: Rumpf der main Funktion

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

- Beispiel: Schleifenrumpf

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

6. Kontrollanweisungen II

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

Sichtbarkeit

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

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

main block

block

„Blickrichtung"

Kontrollanweisung definiert Block

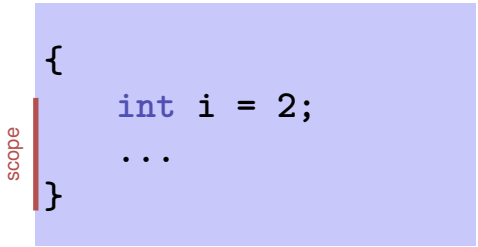
Kontrollanweisungen verhalten sich in diesem Zusammenhang wie Blöcke.

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

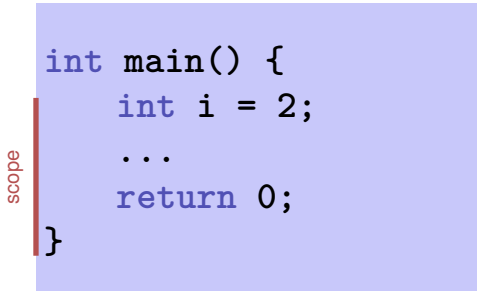
Gültigkeitsbereich einer Deklaration

Potenzieller Gültigkeitsbereich: Ab Deklaration bis Ende des Programmteils, der die Deklaration enthält.

Im Block



Im Funktionsrumpf

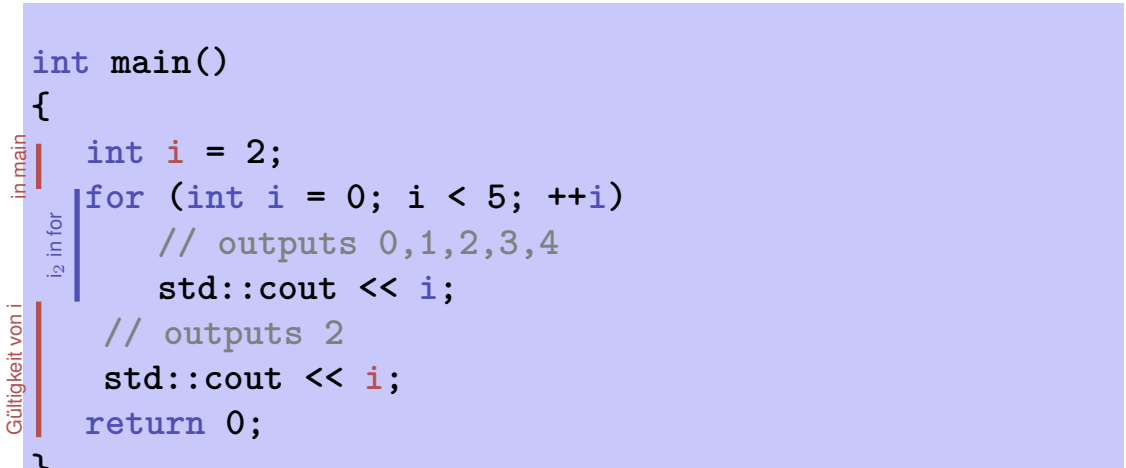


In Kontrollanweisung

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

Gültigkeitsbereich einer Deklaration

Wirklicher Gültigkeitsbereich = Potenzieller Gültigkeitsbereich minus darin enthaltene potenzielle Gültigkeitsbereiche von Deklarationen des gleichen Namens




```
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;
}
```

Automatische Speicherdauer

Lokale Variablen (Deklaration in Block)

- werden bei jedem Erreichen ihrer Deklaration neu „angelegt“, d.h.
  - Speicher / Adresse wird zugewiesen
  - evtl. Initialisierung wird ausgeführt
- werden am Ende ihrer deklarativen Region „abgebaut“ (Speicher wird freigegeben, Adresse wird ungültig)

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

`while` Anweisung

```
while ( condition )
  statement
```

- *statement*: beliebige Anweisung, Rumpf der `while` Anweisung.
- *condition*: konvertierbar nach `bool`.

`while` Anweisung

```
while ( condition )
  statement
```

ist äquivalent zu

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

`while`-Anweisung: Semantik

```
while ( condition )
  statement
```

- *condition* wird ausgewertet
  - `true`: Iteration beginnt
    statement wird ausgeführt
  - `false`: `while`-Anweisung wird beendet.

`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 $(n \in \mathbb{N})$

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

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

Die Collatz-Folge in C++

```
// Program: collatz.cpp
// Compute the Collatz sequence of a number n.

#include <iostream>

int main()
{
  // Input
  std::cout << "Compute the Collatz sequence for n =? ";
  unsigned int n;
  std::cin >> n;

  // Iteration
  while (n > 1) {
    if (n % 2 == 0)
      n = n / 2;
    else
      n = 3 * n + 1;
    std::cout << n << " ";
  }
  std::cout << "\n";
  return 0;
}
```

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
```

Die Collatz-Folge

Erscheint die 1 für jedes n?

- Man vermutet es, aber niemand kann es beweisen!
- Falls nicht, so ist die `while`-Anweisung zur Berechnung der Collatz-Folge für einige n theoretisch eine Endlosschleife.

`do` Anweisung

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

- *statement*: beliebige Anweisung, Rumpf der `do` Anweisung.
- *expression*: konvertierbar nach `bool`.

`do` Anweisung

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

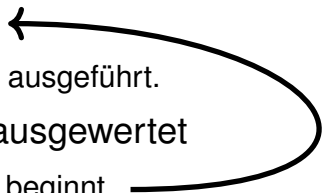
ist äquivalent zu

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

`do`-Anweisung: Semantik

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

- Iteration beginnt
  - *statement* wird ausgeführt.
- *expression* wird ausgewertet
  - `true`: Iteration beginnt
  - `false`: `do`-Anweisung wird beendet.



do-Anweisung: 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);
```

Zusammenfassung

- Auswahl (bedingte *Verzweigungen*)
  - `if` und `if-else`-Anweisung
- Iteration (bedingte *Sprünge*)
  - `for`-Anweisung
  - `while`-Anweisung
  - `do`-Anweisung
- Blöcke und Gültigkeit von Deklarationen

Sprunganweisungen

- `break;`
- `continue;`

`break`-Anweisung

```
break;
```

- umschliessende Iterationsanweisung wird sofort beendet
- nützlich, um Schleife „in der Mitte“ abbrechen zu können [5]

[5] und unverzichtbar bei switch-Anweisungen

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 `break`

Version ohne break wertet `a` zweimal aus und benötigt zusätzlichen Block.

```
int a = 1;
int s = 0;
for (;a != 0;) {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a != 0) {
        s += a;
        std::cout << "sum = " << s << "\n";
    }
```

`continue`-Anweisung

```
continue;
```

- Kontrolle überspringt den Rest des Rumpfes der umschliessenden Iterationsanweisung
- Iterationsanweisung wird aber *nicht* abgebrochen

`break` und `continue` in der Praxis

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

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";
}
```

Äquivalenz von Iterationsanweisungen

Wir haben gesehen:

- `while` und `do` können mit Hilfe von `for` simuliert werden

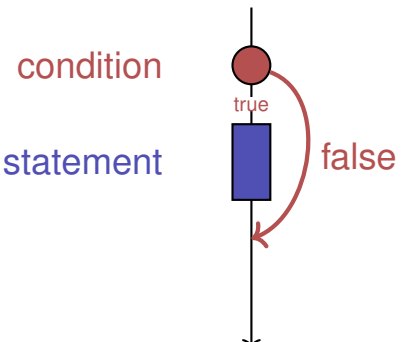
Es gilt aber sogar: Nicht ganz so einfach falls ein continue im Spiel ist!

- Alle drei Iterationsanweisungen haben die gleiche „Ausdruckskraft“ (Skript).

Kontrollfluss

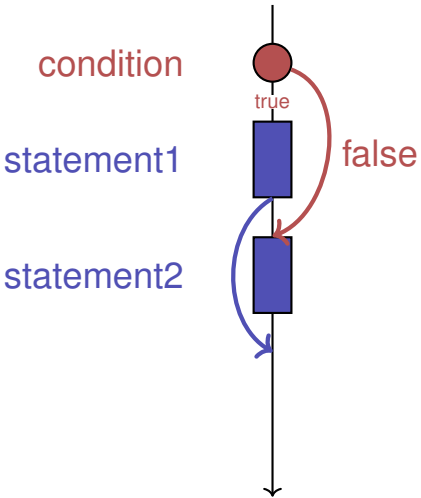
Reihenfolge der (wiederholten) Ausführung von Anweisungen

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



`if` (*condition*)
statement

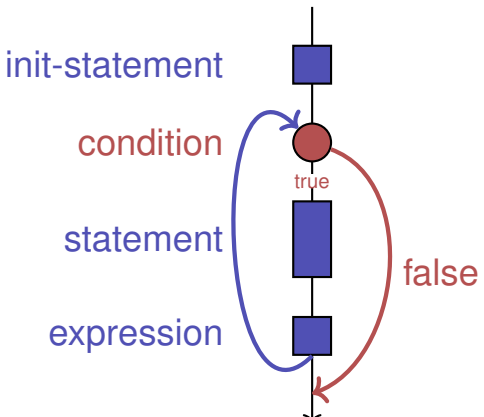
Kontrollfluss `if else`



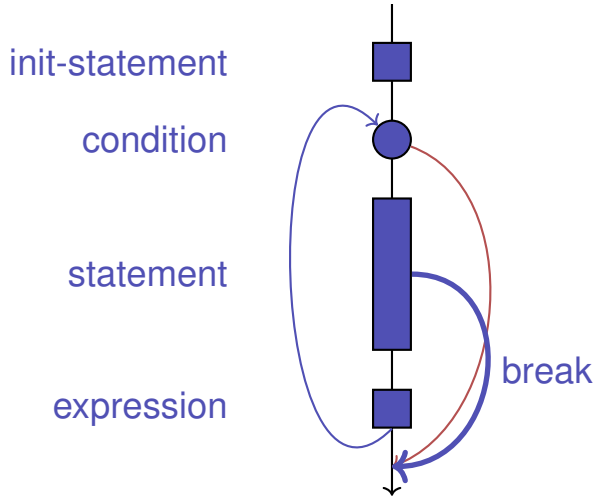

```
if ( condition )
    statement1
else
    statement2
```

Kontrollfluss `for`

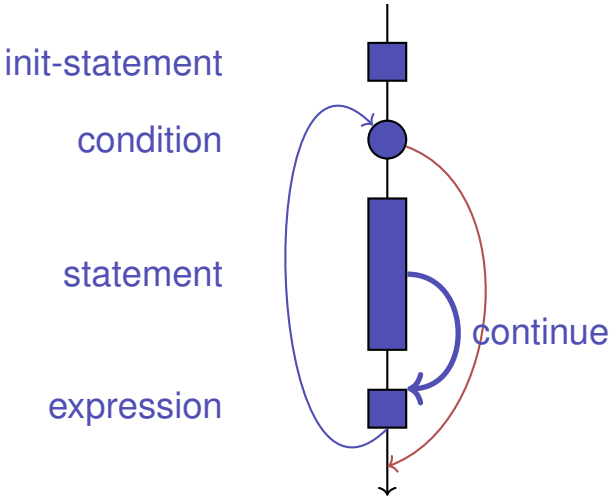
`for` (*init statement* *condition* `;` *expression*)
statement



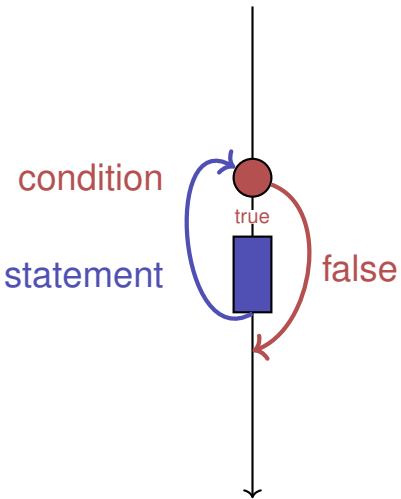
Kontrollfluss `break` in for



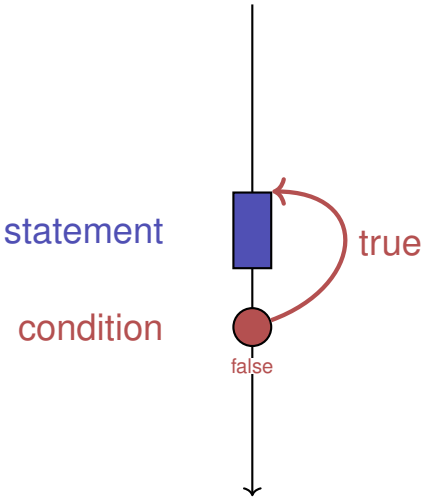
Kontrollfluss `continue` in for



Kontrollfluss `while`



Kontrollfluss `do while`



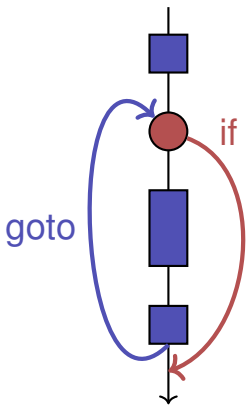
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

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

Sprachen, die darauf basieren:

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



BASIC und die Home-Computer...

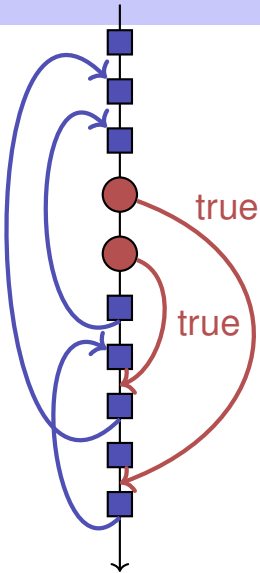
...ermöglichten einer ganzen Generation von Jugendlichen das Programmieren.

Home-Computer Commodore C64 (1982)

Spaghetti-Code mit `goto`

Ausgabe von ???????????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

- 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";
```

Das ist hier die „richtige“ Iterationsanweisung

Sprunganweisungen

- realisieren unbedingte Sprünge.
- sind wie `while` und `do` praktisch, aber nicht unverzichtbar
- sollten vorsichtig eingesetzt werden: nur dort wo sie den Kontrollfluss *vereinfachen*, statt ihn *komplizierter* zu machen

Notenausgabe

1. Funktionale Anforderung:

$$
\begin{aligned}
6 &\rightarrow \texttt{"Excellent ... You passed!"} \\
5,4 &\rightarrow \texttt{"You passed!"} \\
3 &\rightarrow \texttt{"Close, but ... You failed!"} \\
2,1 &\rightarrow \texttt{"You failed!"} \\
\textit{sonst} &\rightarrow \texttt{"Error!"}
\end{aligned}
$$

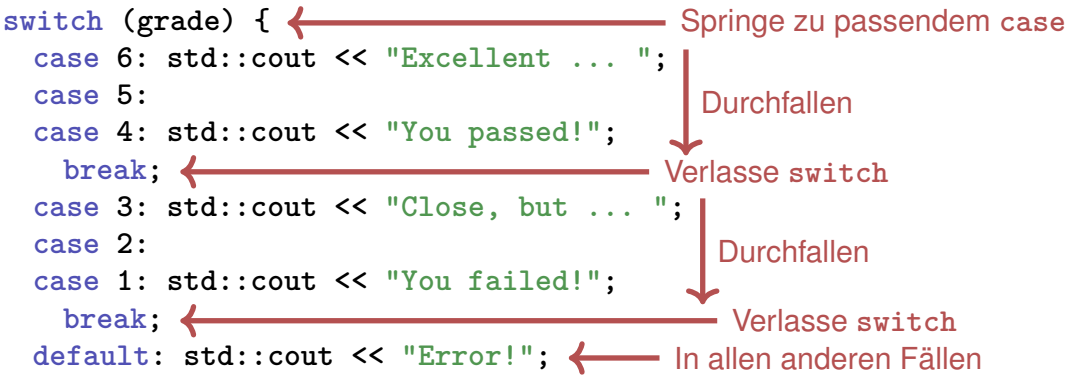
2. Ausserdem: Text- und Codeduplikation vermeiden

Notenausgabe mit `if`-Anweisungen

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

Nachteil: Kontrollfluss – und somit Programmverhalten – nicht gerade offensichtlich

Notenausgabe mit `switch`-Anweisung



Vorteil: Kontrollfluss klar erkennbar

Die `switch`-Anweisung

`switch` (*condition*)
statement

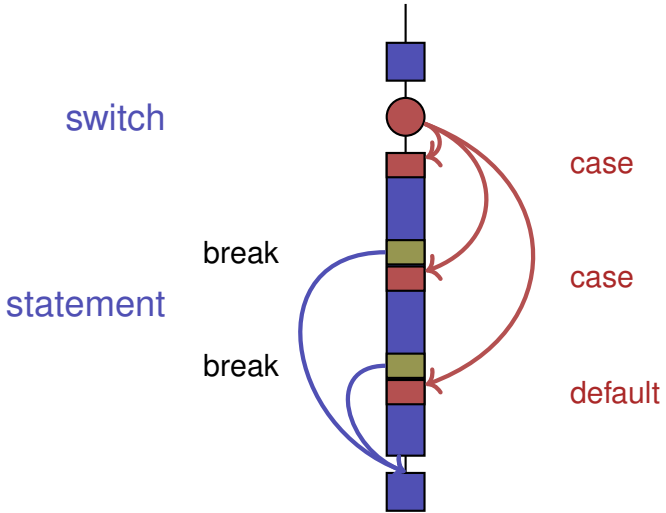
- *condition*: Ausdruck, konvertierbar in einen integralen Typ
- *statement* : beliebige Anweisung, in welcher **`case`** und **`default`**-Marken erlaubt sind, **`break`** hat eine spezielle Bedeutung.
- Benutzung des Durchfallens in der Praxis umstritten, Einsatz gut abwägen (entsprechende Compilerwarnung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

Semantik der `switch`-Anweisung

`switch` (*condition*)
statement

- **`condition`** wird ausgewertet.
- Beinhaltet **`statement`** eine **`case`**-Marke mit (konstantem) Wert von **`condition`**, wird dorthin gesprungen.
- Sonst wird, sofern vorhanden, an die **`default`**-Marke gesprungen. Wenn nicht vorhanden, wird **`statement`** übersprungen.
- Die **`break`**-Anweisung beendet die **`switch`**-Anweisung.

Kontrollfluss `switch`



7. Fliesskommazahlen I

Typen `float` und `double`; Gemischte Ausdrücke und Konversionen; Löcher im Wertebereich;

„Richtig“ Rechnen

```
// Program: fahrenheit_float.cpp
// Convert temperatures from Celsius to Fahrenheit.

#include <iostream>

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

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

Fixkommazahlen

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

```
0.0824 = 0000000.082
```
← dritte Stelle abgeschnitten

Nachteile

- Wertebereich wird *noch* kleiner als bei ganzen Zahlen.
- Repräsentierbarkeit hängt von der Stelle des Kommas ab.

Fliesskommazahlen

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

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

  Anzahl *signifikanter Stellen* bleibt konstant

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

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:
  - `float`: ca. 7 Stellen, Exponent bis ± 38
  - `double`: ca. 15 Stellen, Exponent bis ± 308
- sind auf den meisten Rechnern sehr schnell (Hardwareunterstützung)

Arithmetische Operatoren

Wie bei `int`, aber ...

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

Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma

`1.0` : Typ `double`, Wert 1

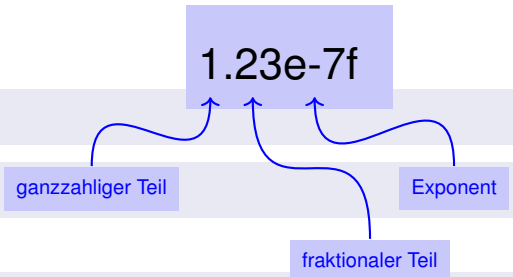
`1.27f` : Typ `float`, Wert 1.27

- und / oder Exponent.

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

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

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



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

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

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

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

Rechnen mit `float`: Eulersche Zahl

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

Gemischte Ausdrücke, Konversion

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

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

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?

Wertebereich

Ganzzahlige Typen:

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

Fliesskommatypen:

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

8. Fliesskommazahlen II

Fliesskommazahlensysteme; IEEE Standard; Grenzen der Fliesskommaarithmetik; Fliesskomma-Richtlinien; Harmonische Zahlen

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

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

Bezeichnung:

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

Fliesskommazahlensysteme

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

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

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

In Basis-β-Darstellung:

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

Fliesskommazahlensysteme

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

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

Unterschiedliche Darstellungsmöglichkeiten durch Wahl des Exponenten

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.

Bemerkung 2

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

Menge der normalisierten Zahlen

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

Normalisierte Darstellung

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

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

Binäre und dezimale Systeme

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

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

Binärdarstellung:

$$
\begin{aligned}
x &= \sum_{i=-\infty}^{0} b_i 2^i = b_0 \bullet b_{-1} b_{-2} b_{-3} \ldots \\
&= b_0 + \sum_{i=-\infty}^{-1} b_i 2^i = b_0 + \sum_{i=-\infty}^{0} b_{i-1} 2^{i-1} \\
&= b_0 + \underbrace{\left(\sum_{i=-\infty}^{0} b_{i-1} 2^i \right)}_{x' = b_{-1} \bullet b_{-2} b_{-3} b_{-4}} /2
\end{aligned}
$$

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

- Also: $x' = b_{-1}\bullet b_{-2}b_{-3}b_{-4}\ldots = 2 \cdot (x - b_0)$
- Schritt 1 (für x): Berechnen von b_0:

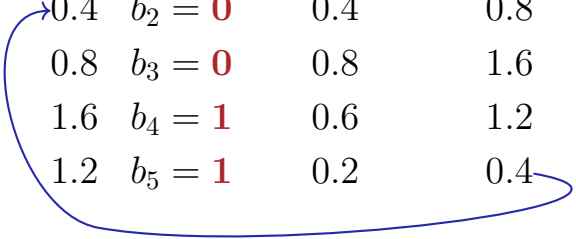
$$b_0 = \begin{cases} 1, & \text{falls } x \geq 1 \\ 0, & \text{sonst} \end{cases}$$

- Schritt 2 (für x): Berechnen von $b_{-1}, b_{-2}, \ldots$:

  Gehe zu Schritt 1 (für $x' = 2 \cdot (x - b_0)$)

Binärdarstellung von 1.1_{10}

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



$\Rightarrow 1.0\overline{0011}$, periodisch, *nicht* endlich

Binärdarstellungen von 1.1 und 0.1

- sind nicht endlich, also gibt es Fehler bei der Konversion in ein (endliches) binäres Fliesskommazahlensystem.
- `1.1f` und `0.1f` sind nicht 1.1 und 0.1, sondern geringfügig fehlerhafte Approximationen dieser Zahlen.
- In `diff.cpp`: $1.1 - 1.0 \neq 0.1$

Binärdarstellungen von 1.1 und 0.1

auf meinem Computer:

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

Rechnen mit Fliesskommazahlen

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

$$\begin{aligned} & 1.111 \cdot 2^{-2} \\ + \quad & 1.011 \cdot 2^{-1} \\ \hline = \quad & 1.001 \cdot 2^{0} \end{aligned}$$

1. Exponenten anpassen durch Denormalisieren einer Zahl 2. Binäre Addition der Signifikanden 3. Renormalisierung 4. Runden auf p signifikante Stellen, falls nötig

Der IEEE Standard 754

- legt Fliesskommazahlensysteme und deren Rundungsverhalten fest
- wird fast überall benutzt
- Single precision (`float`) Zahlen:

  $F^*(2, 24, -126, 127)$ (32 bit) plus $0, \infty, \ldots$
- Double precision (`double`) Zahlen:

  $F^*(2, 53, -1022, 1023)$ (64 bit) 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 (führendes Bit ist 1 und wird nicht gespeichert)
- 8 Bit für den Exponenten (256 mögliche Werte)(254 mögliche Exponenten, 2 Spezialwerte: 0, ∞,...)

$\Rightarrow$ insgesamt 32 Bit.

Der IEEE Standard 754

Warum

$$F^*(2, 53, -1022, 1023)?$$

- 1 Bit für das Vorzeichen
- 52 Bit für den Signifikanden (führendes Bit ist 1 und wird nicht gespeichert)
- 11 Bit für den Exponenten (2046 mögliche Exponenten, 2 Spezialwerte: 0, ∞,...)

$\Rightarrow$ insgesamt 64 Bit.

Beispiel: 32-bit Darstellung einer Fliesskommazahl

$\pm$ Exponent Mantisse

$$\pm \begin{matrix} 2^{-126}, \ldots, 2^{127} \\ 0, \infty, \ldots \end{matrix} \qquad \begin{matrix} 1.00000000000000000000000 \\ \vdots \\ 1.11111111111111111111111 \end{matrix}$$

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!

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

Addition von 1 hat keinen Effekt!

- Die n-te Harmonische Zahl ist

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

- Diese Summe kann vorwärts oder rückwärts berechnet werden, was mathematisch gesehen natürlich äquivalent ist.

Harmonische Zahlen

Regel 2

```
// Program: harmonic.cpp
// Compute the n-th harmonic number in two ways.

#include <iostream>

int main()
{
  // Input
  std::cout << "Compute H_n for n =? ";
  unsigned int n;
  std::cin >> n;

  // Forward sum
  float fs = 0;
  for (unsigned int i = 1; i <= n; ++i)
    fs += 1.0f / i;

  // Backward sum
  float bs = 0;
  for (unsigned int i = n; i >= 1; --i)
    bs += 1.0f / i;

  // Output
  std::cout << "Forward sum  = " << fs << "\n"
            << "Backward sum = " << bs << "\n";
  return 0;
}
```

Harmonische Zahlen

Ergebnisse:

- ```
  Compute H_n for n =?  10000000
  Forward sum = 15.4037
  Backward sum = 16.686
  ```
- ```
  Compute H_n for n =?  100000000
  Forward sum = 15.4037
  Backward sum = 18.8079
  ```

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

Regel 3

Subtrahiere keine zwei Zahlen sehr ähnlicher Grösse!

Auslöschungsproblematik, siehe Skript.

Literatur

David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic (1991)

Randy Glasbergen, 1996

9. Funktionen I

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

Funktionen

- kapseln häufig gebrauchte Funktionalität (z.B. Potenzberechnung) und machen sie einfach verfügbar
- strukturieren das Programm: Unterteilung in kleine Teilaufgaben, jede davon durch eine Funktion realisiert

$\Rightarrow$ Prozedurales Programmieren; Prozedur: anderes Wort für Funktion.

Beispiel: 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~~pow(a,n) << ".\n";
```

"Funktion pow"

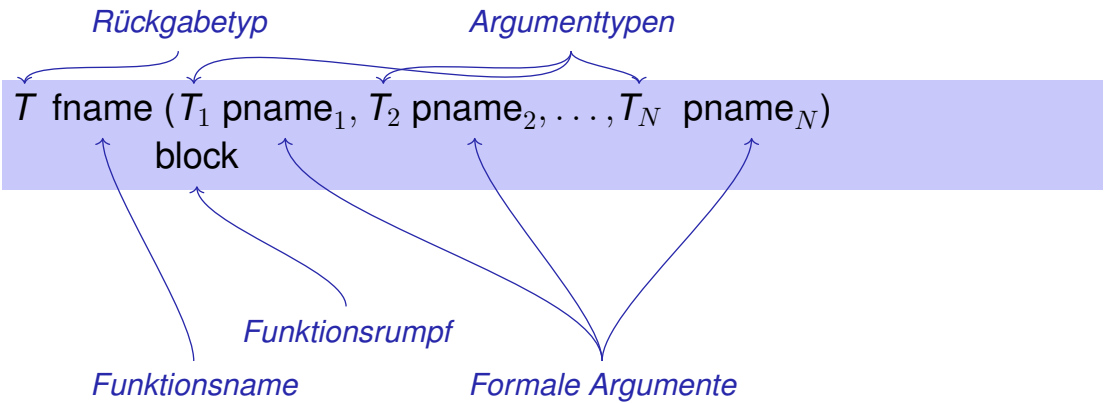
Funktion zur Potenzberechnung

```
// 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

```
// 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

- dürfen nicht *lokal* auftreten, also nicht in Blocks, nicht in anderen Funktionen und nicht in Kontrollanweisungen
- können im Programm ohne Trennsymbole aufeinander folgen

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

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

Beispiel: Xor

```
// post: returns l XOR r
bool Xor(bool l, bool r)
{
    return l != r;
}
```

Beispiel: 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;
}
```

Beispiel: 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. Wert und Effekt wie in der Nachbedingung der Funktion *fname* angegeben.

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

Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

- Aufrufargumente sind R-Werte
  $\hookrightarrow$ *call-by-value* (auch *pass-by-value*), dazu gleich mehr
- Funktionsaufruf selbst ist R-Wert.

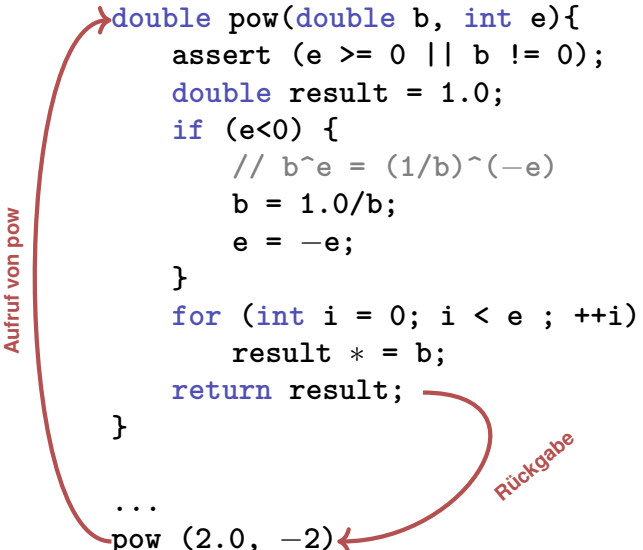
fname: R-Wert $\times$ R-Wert $\times \cdots \times$ R-Wert $\longrightarrow$ R-Wert

Auswertung eines Funktionsaufrufes

- Auswertung der Aufrufargumente
- Initialisierung der formalen Argumente mit den resultierenden Werten
- Ausführung des Funktionsrumpfes: formale Argumente verhalten sich dabei wie lokale Variablen
- Ausführung endet mit
  `return` *expression*;

Rückgabewert ergibt den Wert des Funktionsaufrufes.

Beispiel: Auswertung Funktionsaufruf



Formale Funktionsargumente[6]

- Deklarative Region: Funktionsdefinition
- sind ausserhalb der Funktionsdefinition *nicht* sichtbar
- werden bei jedem Aufruf der Funktion neu angelegt (automatische Speicherdauer)
- Änderungen ihrer Werte haben keinen Einfluss auf die Werte der Aufrufargumente (Aufrufargumente sind R-Werte)

[6] manchmal *formale Parameter*

Gültigkeit formaler Argumente

```
double pow(double b, int e){
    double r = 1.0;
    if (e<0) {
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e ; ++i)
        r *= b;
    return r;
}
```

```
int main(){
    double b = 2.0;
    int e = -2;
    double z = pow(b, e);

    std::cout << z; // 0.25
    std::cout << b; // 2
    std::cout << e; // -2
    return 0;
}
```

Nicht die formalen Argumente `b` und `e` von pow, sondern die hier definierten Variablen lokal zum Rumpf von `main`

Der Typ `void`

```
// POST: "(i, j)" has been written to standard output
void print_pair(int i, int j) {
    std::cout << "(" << i << ", " << j << ")\n";
}

int main() {
    print_pair(3,4); // outputs (3, 4)
    return 0;
}
```

Der Typ `void`

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

`void`-Funktionen

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

Ausdruck vom Typ `void` (z.B. Aufruf einer Funktion mit Rückgabetyp `void`

10. Funktionen II

Vor- und Nachbedingungen Stepwise Refinement,
Gültigkeitsbereich, Bibliotheken, Standardfunktionen

Vor- und Nachbedingungen

- beschreiben (möglichst vollständig) was die Funktion „macht“
- dokumentieren die Funktion für Benutzer / Programmierer (wir selbst oder andere)
- machen Programme lesbarer: wir müssen nicht verstehen, *wie* die Funktion es macht
- werden vom Compiler ignoriert
- Vor- und Nachbedingungen machen – unter der Annahme ihrer Korrektheit – Aussagen über die Korrektheit eines Programmes möglich.

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.

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

- Gilt Vorbedingung beim Funktionsaufruf nicht, so machen wir keine Aussage.
- C++-Standard-Jargon: „Undefined behavior“.

Funktion `pow`: Division durch 0

Vor- und Nachbedingungen

- Vorbedingung sollte so *schwach* wie möglich sein (möglichst grosser Definitionsbereich)
- Nachbedingung sollte so *stark* wie möglich sein (möglichst detaillierte Aussage)

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
```

Die exakten Vor- und Nachbedingungen sind plattformabhängig und meist sehr kompliziert. Wir abstrahieren und geben die mathematischen Bedingungen an. $\Rightarrow$ Kompromiss zwischen formaler Korrektheit und lascher Praxis.

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.
- 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;
}
```

Ausnahmen (Exception Handling)

- Assertions sind ein grober Hammer; falls eine Assertion fehlschlägt, wird das Programm hart abgebrochen.
- C++bietet elegantere Mittel (Exceptions), um auf solche Fehlschläge situationsabhängig (und oft auch ohne Programmabbruch) zu reagieren.
- „Narrensichere“ Programmen sollten nur im Notfall abbrechen und deshalb mit Exceptions arbeiten; für diese Vorlesung führt das aber zu weit.

Stepwise Refinement

- Einfache *Programmiertechnik* zum Lösen komplexer Probleme

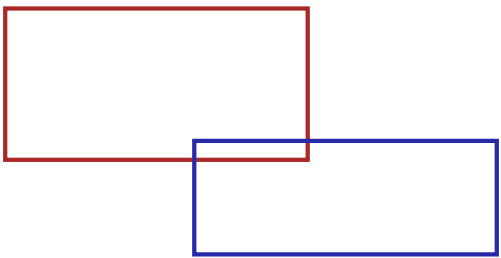
Niklaus Wirth. Program development by stepwise refinement. Commun. ACM 14, 4, 1971

Stepwise Refinement

- Problem wird schrittweise gelöst. Man beginnt mit einer groben Lösung auf sehr hohem Abstraktionsniveau (nur Kommentare und fiktive Funktionen).
- In jedem Schritt werden Kommentare durch Programmtext ersetzt und Funktionen implementiert unterteilt (demselben Prinzip folgend).
- Die Verfeinerung bezieht sich auch auf die Entwicklung der Datenrepräsentation (mehr dazu später).
- Wird die Verfeinerung so weit wie möglich durch Funktionen realisiert, entstehen Teillösungen, die auch bei anderen Problemen eingesetzt werden können.
- Stepwise Refinement fördert (aber ersetzt nicht) das strukturelle Verständnis des Problems.

Beispielproblem

Finde heraus, ob sich zwei Rechtecke schneiden!

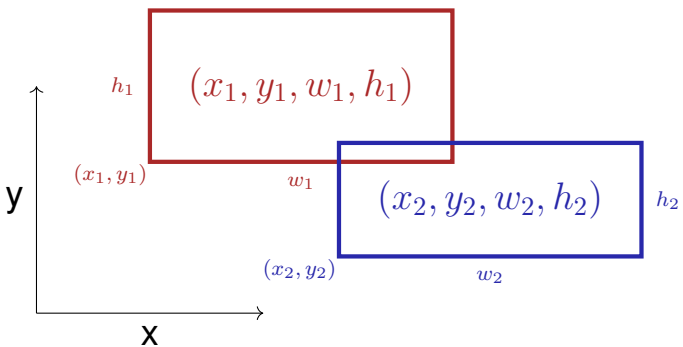


Grobe Lösung

(Include-Direktiven ausgelassen)

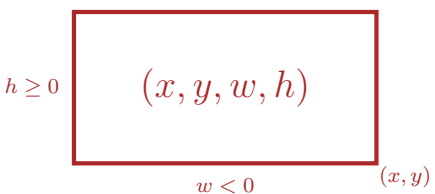
```
int main()
{
    // Eingabe Rechtecke

    // Schnitt?

    // Ausgabe der Loesung

    return 0;
}
```

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
}
```

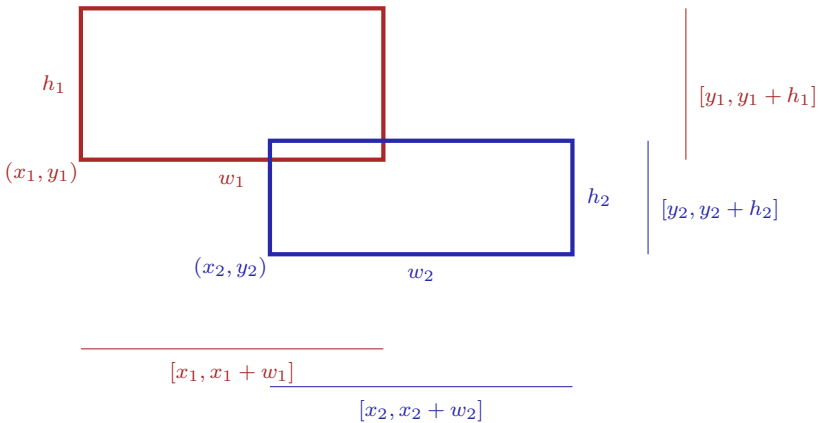
Funktion main ✓

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: [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

```
// 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!

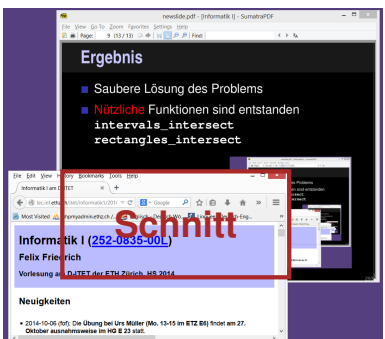
```cpp
#include <iostream>
#include <algorithm>

// 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 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);
}
```

```cpp
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
```



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;
}
```

Gültigkeitsbereich einer Funktion

- ist der Teil des Programmes, in dem die Funktion aufgerufen werden kann
- ist definiert als die Vereinigung der Gültigkeitsbereiche aller ihrer Deklarationen (es kann mehrere geben)

Deklaration einer Funktion: wie Definition aber ohne `{...}`.

```
double pow(double b, int e);
```

So geht's also nicht...

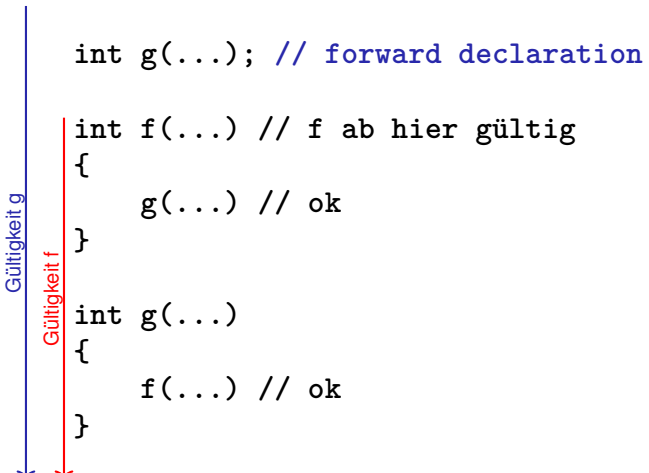
```
#include <iostream>


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

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

...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 g(...); // forward declaration

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

int g(...)
{
    f(...) // ok
}
```

Wiederverwendbarkeit

- Funktionen wie `rectangles_intersect` und `pow` sind in vielen Programmen nützlich.
- „Lösung:“ Funktion einfach ins Hauptprogramm hineinkopieren, wenn wir sie brauchen!
- Hauptnachteil: wenn wir die Funktionsdefinition ändern wollen, müssen wir *alle* Programme ändern, in denen sie vorkommt.

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: Inkludieren der Funktion

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

#include <iostream>
#include "mymath.cpp"

int main()
{
  std::cout << pow( 2.0, -2) << "\n";
  std::cout << pow( 1.5, 2) << "\n";
  std::cout << pow( 5.0, 1) << "\n";
  std::cout << pow(-2.0, 9) << "\n";

  return 0;
}
```

Datei im Arbeitsverzeichnis

Nachteil des Inkludierens

- `#include` kopiert die Datei (`mymath.cpp`) in das Hauptprogramm (`callpow2.cpp`).
- Der Compiler muss die Funktionsdefinition für jedes Programm neu übersetzen.
- Das kann bei sehr vielen und grossen Funktionen sehr lange dauern.

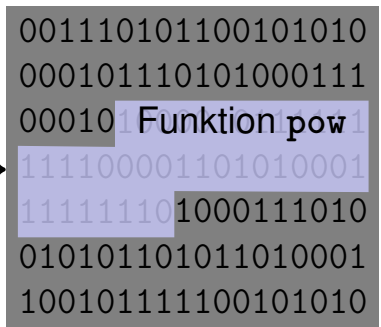
Level 2: Getrennte Übersetzung

von `mymath.cpp` unabhängig vom Hauptprogramm:

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

mymath.cpp

`g++ -c mymath.cpp`



mymath.o

Level 2: Getrennte Übersetzung

Deklaration aller benötigten Symbole in sog. *Header* Datei.

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

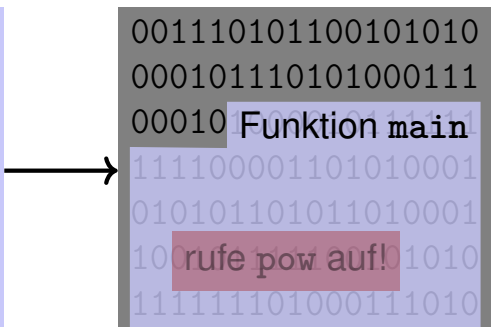
mymath.h

Level 2: Getrennte Übersetzung

des Hauptprogramms unabhängig von `mymath.cpp`, wenn eine *Deklaration* aus `mymath` inkludiert wird.

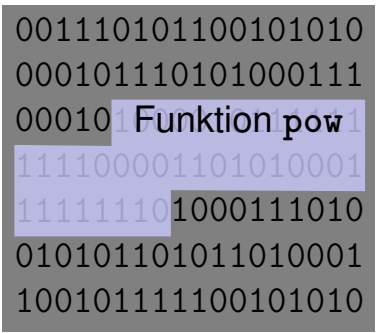
```
#include <iostream>
#include "mymath.h"
int main()
{
  std::cout << pow(2,-2) << "\n";
  return 0;
}
```

callpow3.cpp



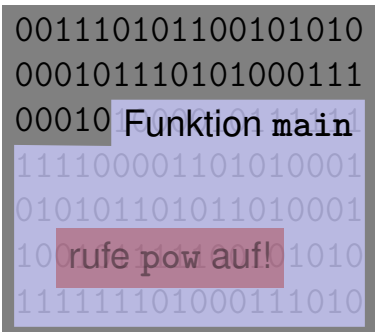
callpow3.o

Der Linker vereint...



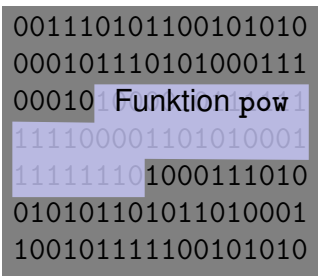
mymath.o

+

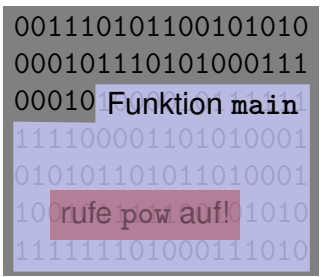


callpow3.o

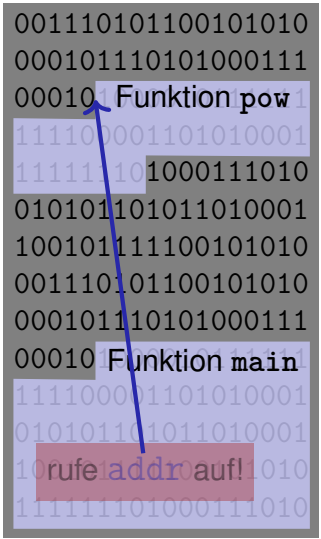
... was zusammengehört



+



=



Ausführbare Datei callpow3

Verfügbarkeit von Quellcode?

Beobachtung

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

Viele Anbieter von Funktionsbibliotheken liefern dem Benutzer keinen Quellcode.

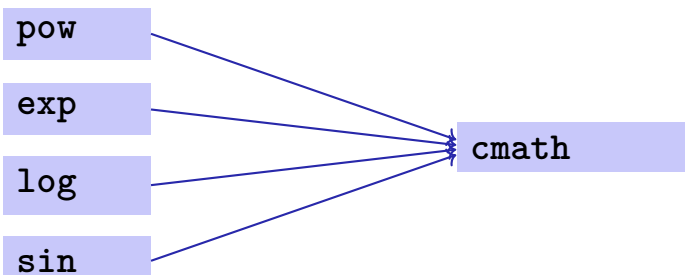
Header-Dateien sind dann die *einzigen* lesbaren Informationen.

Open-Source-Software

- Alle Quellcodes sind verfügbar.
- Nur das erlaubt die Weiterentwicklung durch Benutzer und engagierte „Hacker“.
- Selbst im kommerziellen Bereich ist Open-Source-Software auf dem Vormarsch.
- Lizenzen erzwingen die Nennung der Quellen und die offene Weiterentwicklung. Beispiel: GPL (GNU General Public License).
- Bekannte Open-Source-Softwares: Linux (Betriebssystem), Firefox (Browser), Thunderbird (Email-Programm)

Bibliotheken

- Logische Gruppierung ähnlicher Funktionen



Namensräume...

```
// cmath
namespace std {

  double pow(double b, int e);

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

...vermeiden Namenskonflikte

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

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

Namensräume / Kompilationseinheiten

In C++ ist das Konzept der separaten Kompilation *unabhängig* vom Konzept der Namensräume.

In manchen anderen Sprachen, z.B. Modula / Oberon (zum Teil auch bei Java) definiert die Kompilationseinheit gerade einen Namensraum.

Funktionen aus der Standardbibliothek

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

Beispiel: Primzahltest mit `sqrt`

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \ldots, n-1\}$ 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);
```

- Das funktioniert, weil `std::sqrt` auf die nächste darstellbare `double`-Zahl rundet (IEEE Standard 754).

Primzahltest mit `sqrt`

```
// Test if a given natural number is prime.
#include <iostream>
#include <cassert>
#include <cmath>

int main ()
{
  // Input
  unsigned int n;
  std::cout << "Test if n>1 is prime for n =? ";
  std::cin >> n;
  assert (n > 1);

  // Computation: test possible divisors d up to sqrt(n)
  unsigned int bound = std::sqrt(n);
  unsigned int d;
  for (d = 2; d <= bound && n % d != 0; ++d);@

  // Output
  if (d <= bound)
    // d is a divisor of n in {2,...,[sqrt(n)]}
    std::cout << n << " = " << d << " * " << n / d << ".\n";
  else
    // no proper divisor found
    std::cout << n << " is prime.\n";

  return 0;
}
```

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); // ok!
}
```

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

11. Referenztypen

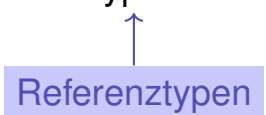
Referenztypen: Definition und Initialisierung, Pass By Value , Pass by Reference, Temporäre Objekte, Konstanten, Const-Referenzen

Swap!

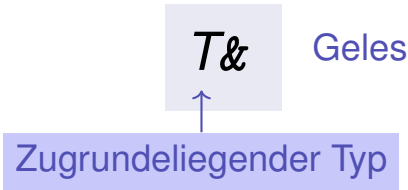
```
// POST: values of x and y are exchanged
void swap (int& x, int& y) {
 int t = x;
 x = y;
 y = t;
}
int main(){
    int a = 2;
    int b = 1;
    swap (a, b);
    assert (a == 1 && b == 2); // ok!
}
```

Referenztypen

- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern!
- Kein neues Konzept auf der Funktionenseite, sondern eine neue Klasse von Typen



Referenztypen: Definition



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

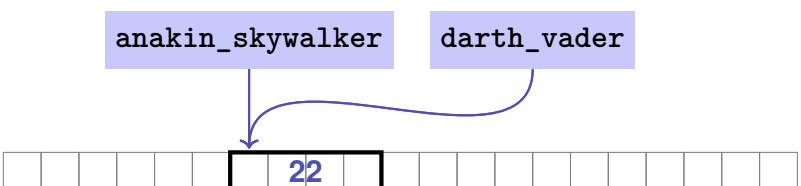
Anakin Skywalker alias Darth Vader

Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker; //  Alias
darth_vader = 22;
```

Zuweisung an den L-Wert hinter dem Alias

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



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

int& k = 5;  // Fehler: Das Literal 5 hat keine Adresse
```

Pass by Reference

Referenztypen erlauben Funktionen, die Werte ihrer Aufrufargumente zu ändern:

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

Pass by Reference

Formales Argument hat Referenztyp:

$\Rightarrow$ **Pass by Reference**

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

Pass by Value

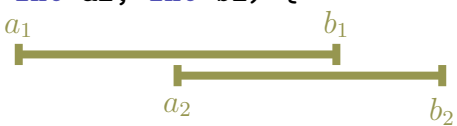
Formales Argument hat keinen Referenztyp:

$\Rightarrow$ **Pass by Value**

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

Referenzen im Kontext von intervals_intersect

```
// PRE:  [a1, b1], [a2, b2] are (generalized) intervals,
// POST: returns true if [a1, b1], [a2, b2] intersect, in which case
//       [l, h] contains the intersection of [a1, b1], [a2, b2]
bool intervals_intersect (int& l, int& h,
                          int a1, int b1, int a2, int b2) {
  sort (a1, b1);
  sort (a2, b2);
  l = std::max (a1, a2); // Zuweisungen
  h = std::min (b1, b2); // via Referenzen
  return l <= h;
}
...
int lo = 0; int hi = 0;
if (intervals_intersect (lo, hi, 0, 2, 1, 3)) // Initialisierung
    std::cout << "[" << lo << "," << hi << "]" << "\n"; // [1,2]
```



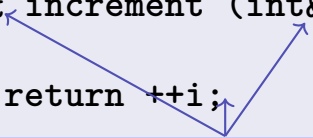
Referenzen im Kontext von intervals_intersect

```
// POST: a <= b
void sort (int& a, int& b) {
   if (a > b)
     std::swap (a, b); // Initialisierung ("Durchreichen" von a, b)
}

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

Return by 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;
}
```

Rückgabewert vom Typ `int&` wird Alias des formalen Arguments, 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
```

Die Referenz-Richtlinie

Referenz-Richtlinie

Wenn man eine Referenz erzeugt, muss das Objekt, auf das sie verweist, mindestens so lange „leben“ wie die Referenz selbst.

Const-Referenzen

- haben Typ `const` *T* `&`
- Typ kann verstanden werden als „(`const` *T*) `&`“
- können auch mit R-Werten initialisiert werden (Compiler erzeugt temporäres Objekt ausreichender Lebensdauer)

`const` *T*`&` `r` = *lvalue*;

r wird mit der Adresse von *lvalue* initialisiert (effizient)

`const` *T*`&` `r` = *rvalue*;

r wird mit der Adresse eines temporären Objektes vom Wert des *rvalue* initialisiert (pragmatisch)

Wann `const` *T*`&` ?

Regel

Argumenttyp `const` *T* `&` (pass by *read-only* reference) wird aus Effizienzgründen anstatt *T* (pass 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

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.

```
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
```

12. Vektoren und Strings I

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

Vektoren: Motivation

- Wir können jetzt über Zahlen iterieren

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

- Oft muss man aber über *Daten* iterieren (Beispiel: Finde ein Kino in Zürich, das heute „C++ Runner 2049“ zeigt)
- Vektoren dienen zum Speichern *gleichartiger* Daten (Beispiel: Spielpläne aller Zürcher Kinos)

Vektoren: erste Anwendung

Das Sieb des Erathostenes

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

Am Ende des Streichungsprozesses bleiben nur die Primzahlen übrig.

- Frage: wie streichen wir Zahlen aus ??
- Antwort: mit einem *Vektor*.

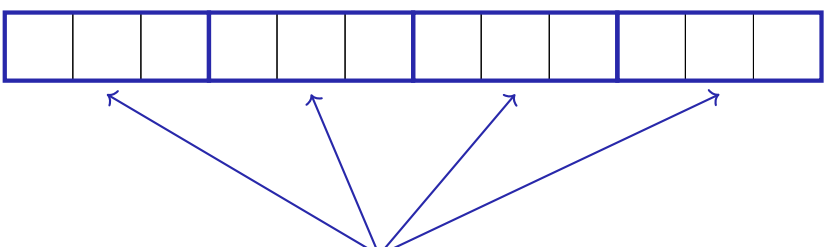
Sieb des Eratosthenes mit Vektoren

```
#include <iostream>
#include <vector> // standard containers with vector functionality
int main() {
  // input
  std::cout << "Compute prime numbers in {2,...,n−1} for n =? ";
  unsigned int n;
  std::cin >> n;

  // definition and initialization: provides us with Booleans
  // crossed_out[0],..., crossed_out[n−1], initialized to false
  std::vector<bool> crossed_out (n, false);

  // computation and output
  std::cout << "Prime numbers in {2,...," << n−1 << "}:\n";
  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;
    }
  std::cout << "\n";
  return 0;
```

Speicherlayout eines Vektors

- Ein Vector belegt einen *zusammenhängenden* Speicherbereich

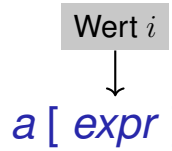
Beispiel: ein Vektor mit 4 Elementen



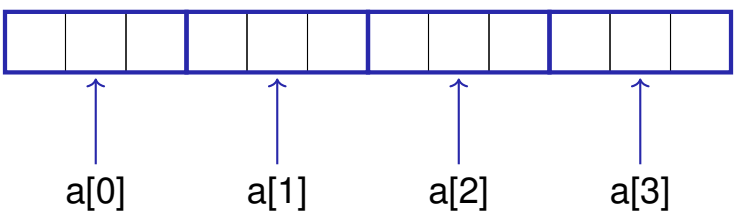
Speicherzellen für jeweils einen Wert vom Typ `T`

Wahlfreier Zugriff (Random Access)

Der L-Wert



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



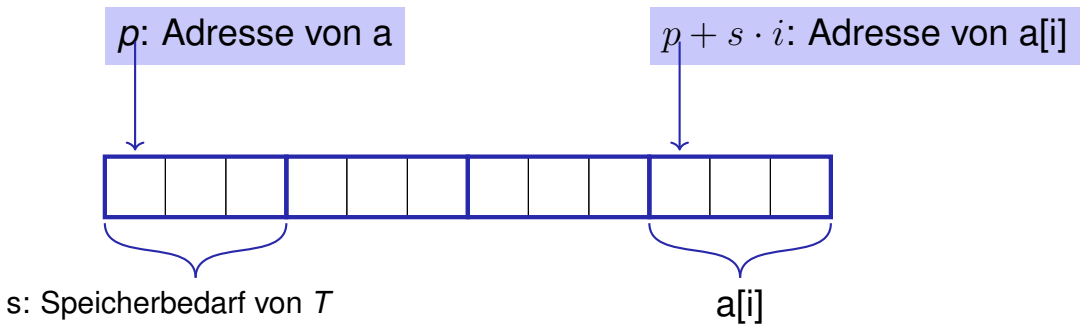
Wahlfreier Zugriff (Random Access)

a [*expr*]

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

Wahlfreier Zugriff (Random Access)

- Wahlfreier Zugriff ist sehr effizient:



Vektor Initialisierung

- `std::vector<int> a (5);`
  Die 5 Elemente von `a` werden mit null initialisiert
- `std::vector<int> a (5, 2);`
  Die 5 Elemente von `a` werden mit 2 initialisiert.
- `std::vector<int> a {4, 3, 5, 2, 1};`
  Der Vektor wird mit einer *Initialisierungsliste* initialisiert.
- `std::vector<int> a;`
  Ein leerer Vektor wird erstellt.

Achtung

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

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

Achtung

Prüfung der Indexgrenzen

Bei Verwendung des Indexoperators auf einem Vektor ist es die alleinige *Verantwortung des Programmierers*, die Gültigkeit aller Elementzugriffe zu prüfen.

Vektoren bieten Komfort

```
std::vector<int> v (10);
v.at(5) = 3; // with bound check
v.push_back(8); // 8 is appended
std::vector<int> w = v; // w is initialized with v
int sz = v.size(); // sz = 11
```

Zeichen und Texte

- Texte haben wir schon gesehen:

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

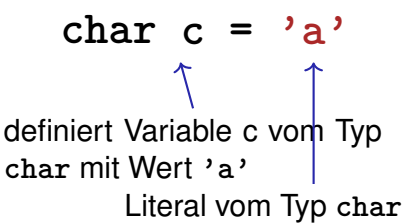
String-Literal

- Können wir auch „richtig“ mit Texten arbeiten? Ja:

Zeichen:	Wert des fundamentalen Typs `char`
Text:	`std::string` $\approx$ Vektor von `char` Elementen

Der Typ `char` („character“)

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



Der Typ `char` („character“)

ist formal ein ganzzahliger Typ

- Werte konvertierbar nach **`int`** / **`unsigned int`**
- Alle arithmetischen Operatoren verfügbar (Nutzen zweifelhaft: was ist **`'a'/'b'`** ?)
- Werte belegen meistens 8 Bit

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

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.
- 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`

Interessante Eigenschaft: bei jedem Byte kann entschieden werden, ob ein UTF8 Zeichen beginnt.

Einige Zeichen in UTF-8

Symbol	Codierung (jeweils 16 Bit)
ئ	11101111 10101111 10111001
	11100010 10011000 10100000
	11100010 10011000 10000011
	11100010 10011000 10011001
A	01000001

P.S.: Suchen Sie mal nach `apple "unicode of death"`

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: `shift`-Funktion

```
// pre: divisor > 0
// post: return the remainder of dividend / divisor
//       with 0 <= result < divisor
int mod(int dividend, int divisor);

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

"- 32" transforms interval $[32, 126]$ to $[0, 94]$
"32 +" transforms interval $[0, 94]$ back to $[32, 126]$
mod(x,95) is the representative of $x (\mathrm{mod} 95)$ in interval $[0, 94]$

Caesar-Code: `caesar`-Funktion

```
// POST: Each character read from std::cin was shifted cyclically
//       by s characters and afterwards written to std::cout
void caesar(int s) {
  std::cin >> std::noskipws; // #include <ios>

  char next;
  while (std::cin >> next) {
    std::cout << shift(next, s);
  }
}
```

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

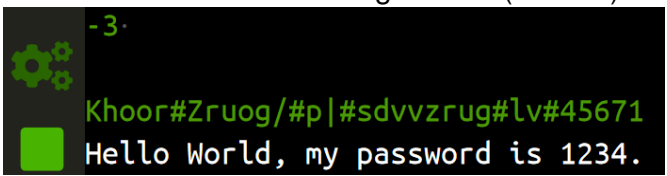
Verschiebt nur druckbare Zeichen.

Caesar-Code: Hauptprogramm

```
int main() {
  int s;
  std::cin >> s;

  // Shift input by s
  caesar(s);

  return 0;
}
```

Verschlüsseln: Verschiebung um n (hier: 3)

Entschlüsseln: Verschiebung um $-n$ (hier: -3)

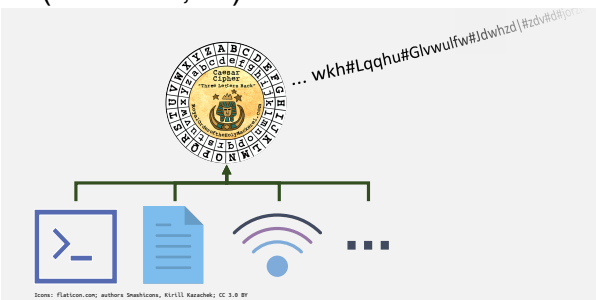


Caesar-Code: Generalisierung

```
void caesar(int s) {
  std::cin >> std::noskipws;

  char next;
  while (std::cin >> next) {
    std::cout << shift(next, s);
  }
}
```

- Momentan nur von `std::cin` nach `std::cout`

- Besser: von beliebiger Zeichenquelle (Konsole, Datei, ...) zu beliebiger Zeichensenke (Konsole, ...)



Caesar-Code: Generalisierung

```
void caesar(std::istream& in,
            std::ostream& out,
            int s) {

  in >> std::noskipws;

  char next;
  while (in >> next) {
    out << shift(next, s);
  }
}
```

- `std::istream`/`std::ostream` ist ein *generischer Eingabe-/Ausgabestrom* an `chars`
- Aufruf der Funktion erfolgt mit *spezifischen Strömen*, z.B.: Konsole (`std::cin`/`cout`), Dateien (`std::i/ofstream`), Strings (`std::i/ostringstream`)

Caesar-Code: Generalisierung, Beispiel 1

```
#include <iostream>
...

// in void main():
caesar(std::cin, std::cout, s);
```

Aufruf der generischen `caesar`-Funktion: Von `std::cin` nach `std::cout`

Caesar-Code: Generalisierung, Beispiel 2

```
#include <iostream>
#include <fstream>
...

// in void main():
std::string from_file_name = ...; // Name of file to read from
std::string to_file_name = ...; // Name of file to write to
std::ifstream from(from_file_name); // Input file stream
std::ofstream to(to_file_name); // Output file stream

caesar(from, to, s);
```

Aufruf der generischen `caesar`-Funktion: Von Datei zu Datei

Caesar-Code: Generalisierung, Beispiel 3

```
#include <iostream>
#include <sstream>
...

// in void main():
std::string plaintext = "My password is 1234";
std::istringstream from(plaintext);

caesar(from, std::cout, s);
```

Aufruf der generischen `caesar`-Funktion: Von einem String nach `std::cout`

13. Vektoren und Strings II

Strings, Mehrdimensionale Vektoren/Vektoren von Vektoren,
Kürzeste Wege, Vektoren als Funktionsargumente

Texte

- Text „`Sein oder nicht sein`“ könnte als `vector<char>` repräsentiert werden
- Texte sind jedoch allgegenwärtig, daher existiert in der Standardbibliothek ein eigener Typ für sie: `std::string` (Zeichenkette)
- Benutzung benötigt `#include <string>`

Benutzung von `std::string`

- Deklaration und Initialisierung mittels Literal:

```
std::string text = "Essen ist fertig!"
```

- Mit variabler Länge initialisieren:

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

`text` wird mit n `'a'`s gefüllt

- Texte vergleichen:

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

`true` wenn zeichenweise gleich

Benutzung von `std::string`

- Grösse auslesen:

```
for (unsigned int i = 0; i < text.size(); ++i) ...
```

Grösse ungleich Textlänge für Multibytekodierungen, z.B. UTF-8

- Einzelne Zeichen lesen:

```
if (text[0] == 'a') ... // or text.at(0)
```

`text[0]` prüft Indexgrenzen nicht, `text.at(0)` hingegen schon

- Einzelne Zeichen schreiben:

```
text[0] = 'b'; // or text.at(0)
```

Benutzung von `std::string`

- Strings konkatenieren (zusammensetzen):

```
text = ":-";
text += ")";
assert(text == ":-)");
```

- Viele weitere Operationen, bei Interesse siehe `https://en.cppreference.com/w/cpp/string`

Mehrdimensionale Vektoren

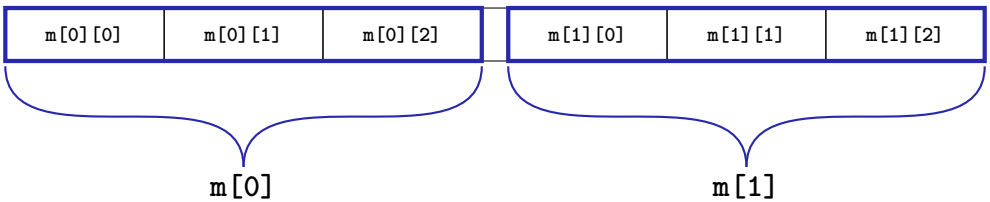
- Zum Speichern von mehrdimensionalen Strukturen wie Tabellen, Matrizen, ...

- ... können *Vektoren von Vektoren* verwendet werden:

```
std::vector<std::vector<int>> m; // An empty matrix
```

Mehrdimensionale Vektoren

Im Speicher: flach



Im Kopf: Matrix

Spalten →

Zeilen ↓	0	1	2
0	`m[0][0]`	`m[0][1]`	`m[0][2]`
1	`m[1][0]`	`m[1][1]`	`m[1][2]`

Mehrdimensionale Vektoren: Initialisierungsbeispiele

Mittels Literalen[7]:

```
// A 3-by-5 matrix
std::vector<std::vector<std::string>> m = {
  {"ZH", "BE", "LU", "BS", "GE"},
  {"FR", "VD", "VS", "NE", "JU"},
  {"AR", "AI", "OW", "IW", "ZG"}
};

assert(m[1][2] == "VS");
```

[7] eigentlich *Initialisierungslisten*

Mehrdimensionale Vektoren: Initialisierungsbeispiele

Auf bestimmte Grösse füllen:

```
unsigned int a = ...;
unsigned int b = ...;

// An a-by-b matrix with all ones
std::vector<std::vector<int>>
  m(a, std::vector<int>(b, 1));
```

`m` (Typ `std::vector<std::vector<int>>`) ist ein Vektor der Länge `a`, dessen Elemente (Typ `std::vector<int>`) Vektoren der Länge `b` sind, deren Elemente (Typ `int`) alles Einsen sind

(Es gibt noch viele weitere Wege, Vektoren zu initialisieren)

Mehrdimensionale Vektoren und Typ-Aliasse

- Auch möglich: Vektoren von Vektoren von Vektoren von ...:
  `std::vector<std::vector<std::vector<...>>>`
- Typnamen können offensichtlich laaaaaaaang werden
- Dann hilft die Deklaration eines *Typ-Alias*:

`using` *Name* = *Typ*;

Name, unter dem der Typ neu auch angesprochen werden kann

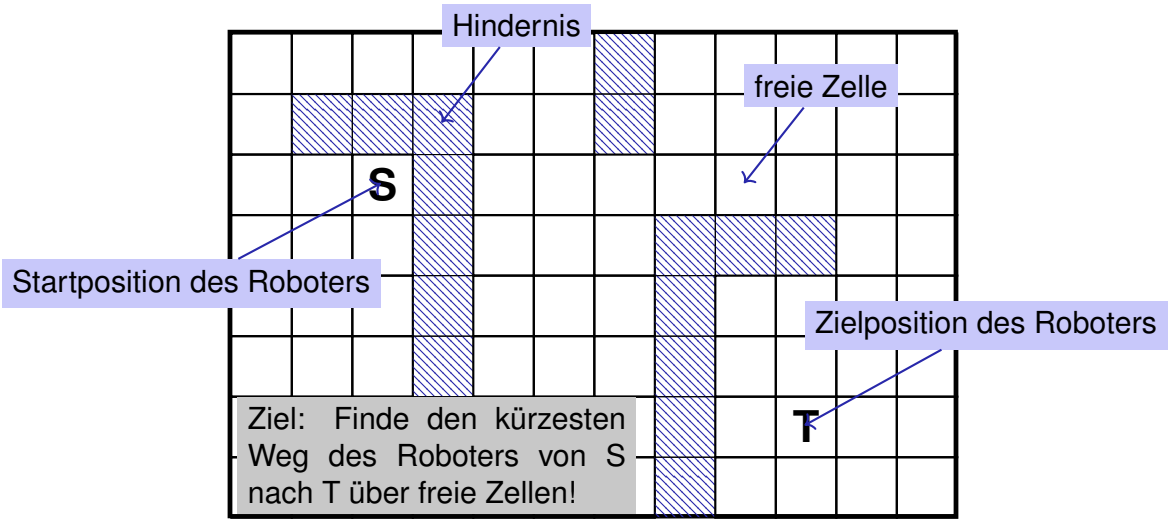
bestehender Typ

Typ-Aliasse: Beispiel

```
#include <iostream>
#include <vector>
using imatrix = std::vector<std::vector<int>>;

// POST: Matrix 'm' was printed to stream 'to'
void print(imatrix m, std::ostream to);

int main() {
  imatrix m = ...;
  print(m, std::cout);
}
```

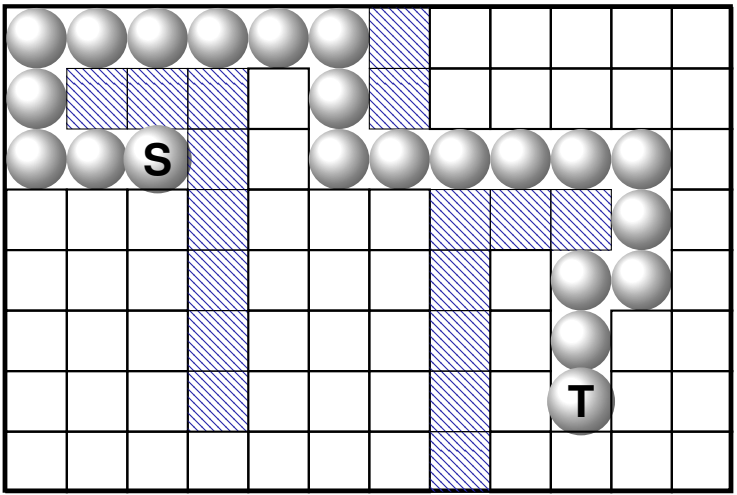
Anwendung: Kürzeste Wege

Fabrik-Halle ($n \times m$ quadratische Zellen)



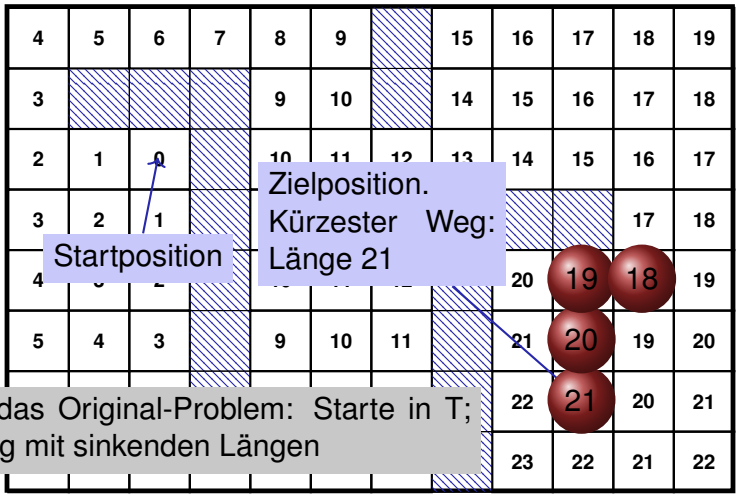
Anwendung: Kürzeste Wege

Lösung



Ein (scheinbar) anderes Problem

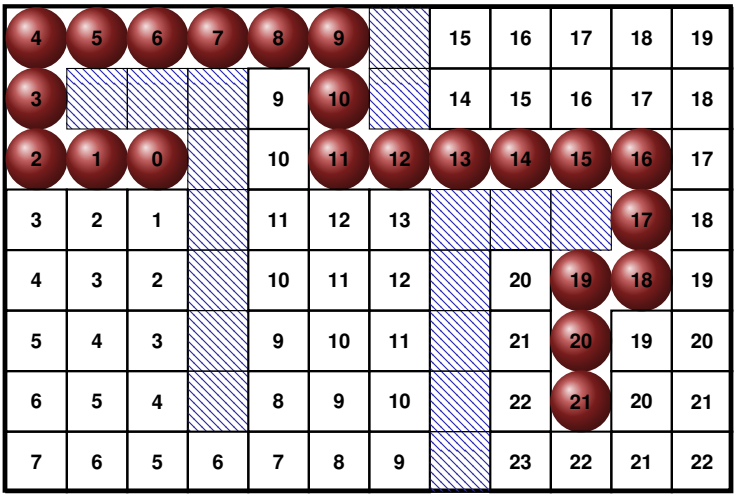
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



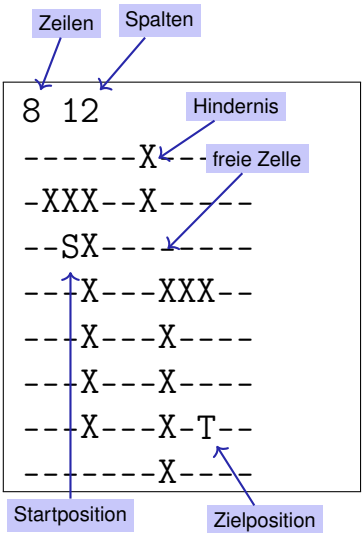
Das löst auch das Original-Problem: Starte in T; folge einem Weg mit sinkenden Längen

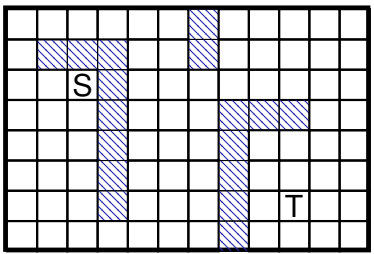
Ein (scheinbar) anderes Problem

Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen

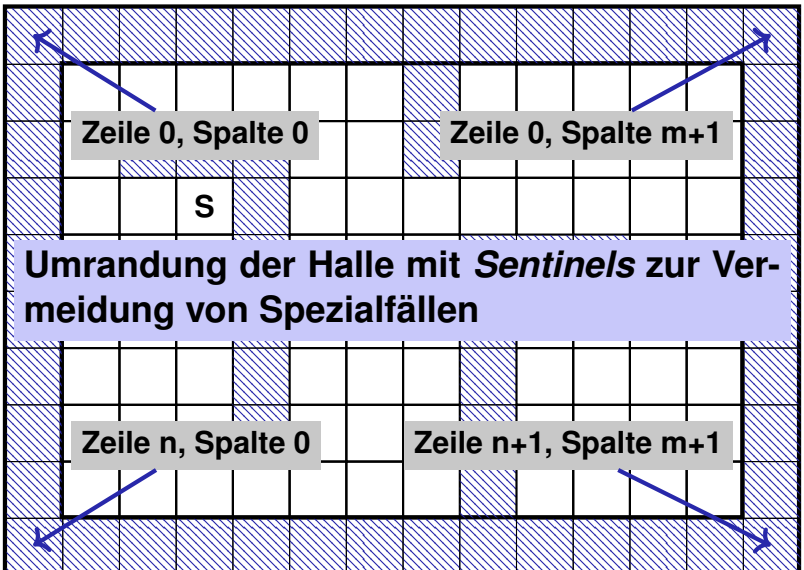


Vorbereitung: Eingabeformat

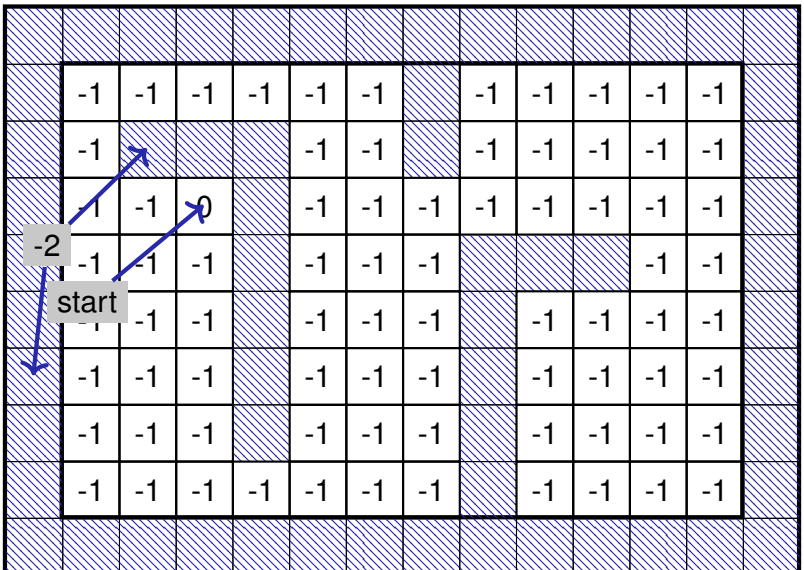




Vorbereitung: Wächter (*Sentinels*)



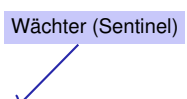
Vorbereitung: Initiale Markierung



Das Kürzeste-Wege-Programm

- Einlesen der Dimensionen und Bereitstellung eines zweidimensionalen Feldes für die Weglängen

```
#include<iostream>
#include<vector>

int main()
{
  // read floor dimensions
  int n; std::cin >> n; // number of rows
  int m; std::cin >> m; // number of columns

  // 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));
```



Das Kürzeste-Wege-Programm

- Einlesen der Hallenbelegung und Initialisierung der Längen

```
int tr = 0;
int tc = 0;
for (int r=1; r<n+1; ++r)
  for (int c=1; c<m+1; ++c) {
    char entry = '-';
    std::cin >> entry;
    if      (entry == 'S') floor[r][c] = 0;
    else if (entry == 'T') floor[tr = r][tc = c] = -1;
    else if (entry == 'X') floor[r][c] = -2;
    else if (entry == '-') floor[r][c] = -1;
  }
```

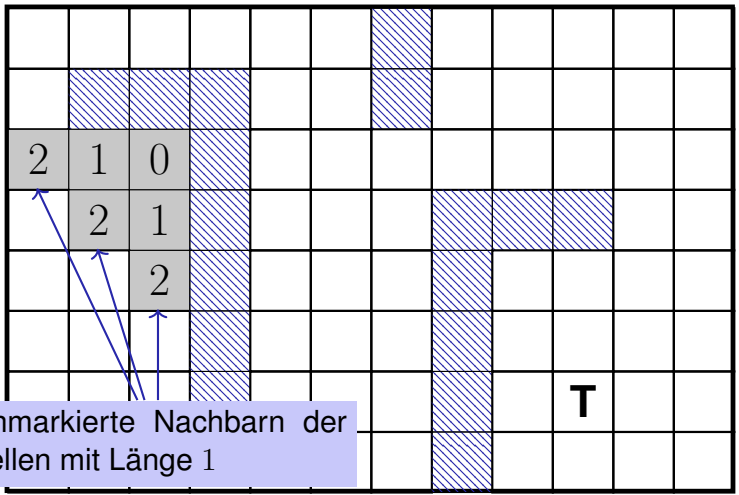
Das Kürzeste-Wege-Programm

- Hinzufügen der umschliessenden „Wände”

```
for (int r=0; r<n+2; ++r)
  floor[r][0] = floor[r][m+1] = -2;

for (int c=0; c<m+2; ++c)
  floor[0][c] = floor[n+1][c] = -2;
```

Markierung aller Zellen mit ihren Weglängen

Schritt 2: Alle Zellen mit Weglänge 2



Hauptschleife

Finde und markiere alle Zellen mit Weglängen $i = 1, 2, 3...$

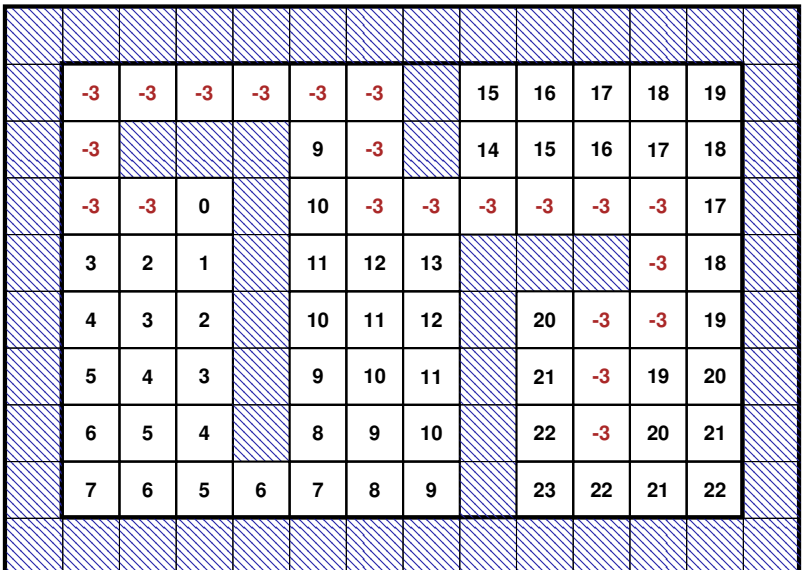
```
for (int i=1;; ++i) {
  bool progress = false;
  for (int r=1; r<n+1; ++r)
     for (int c=1; c<m+1; ++c) {
      if (floor[r][c] != -1) continue;
      if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||
          floor[r][c-1] == i-1 || floor[r][c+1] == i-1 ) {
        floor[r][c] = i; // label cell with i
        progress = true;
      }
    }
  if (!progress) break;
}
```

Das Kürzeste-Wege-Programm

Markieren des kürzesten Weges durch „Rückwärtslaufen“ vom Ziel zum Start

```
int r = tr; int c = tc;
while (floor[r][c] > 0) {
  const int d = floor[r][c] - 1;
  floor[r][c] = -3;
  if      (floor[r-1][c] == d) --r;
  else if (floor[r+1][c] == d) ++r;
  else if (floor[r][c-1] == d) --c;
  else ++c; // (floor[r][c+1] == d)
}
```

Markierung am Ende



Das Kürzeste-Wege-Programm: Ausgabe

Ausgabe

```
for (int r=1; r<n+1; ++r) {
  for (int c=1; c<m+1; ++c)
    if (floor[r][c] == 0)
        std::cout << 'S';
    else if (r == tr && c == tc)
        std::cout << 'T';
    else if (floor[r][c] == -3)
        std::cout << 'o';
    else if (floor[r][c] == -2)
        std::cout << 'X';
    else
        std::cout << '-';
  std::cout << "\n";
}
```

```
ooooooX-----
oXXX-oX-----
ooSX-oooooo-
---X---XXXo-
---X---X-oo-
---X---X-o--
---X---X-T--
-------X----
```

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 $i - 1$
- Verbesserung: Stoppe sobald das Ziel erreicht wurde

Ausgeben einer Matrix: Version 1

- Zur Erinnerung:

```
// POST: Matrix 'm' was printed to std::cout
void print(std::vector<std::vector<int>> m);
...
print(m);
```

- Nachteil: Beim Aufruf `print(m)` wird die (potentiell grosse) Matrix `m` kopiert (*call-by-value*) $\Rightarrow$ ineffizient

Ausgeben einer Matrix: Version 2

- Besser: Übergabe als Referenz (*call-by-reference*)

```
// POST: Matrix 'm' was printed to std::cout
void print(std::vector<std::vector<int>>& m);
...
print(m);
```

- Nachteil: `print(m)` könnte die Matrix verändern $\Rightarrow$ potentiell fehleranfällig

Ausgeben einer Matrix: Version 3

- Besser: Übergabe als `const`-Referenz

```
// POST: Matrix 'm' was printed to std::cout
void print(const std::vector<std::vector<int>>& m);
...
print(m);
```

- Jetzt: Effizient und trotzdem nicht fehleranfälliger

14. Rekursion 1

Mathematische Rekursion, Terminierung, der Aufrufstapel, Beispiele, Rekursion vs. Iteration, n-Damen Problem, Lindenmayer System

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...
- ...nur noch schlechter („verbrennt“ Zeit und Speicher)

```
void f()
{
  f(); // f() -> f() -> ... stack overflow
}
```

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)`

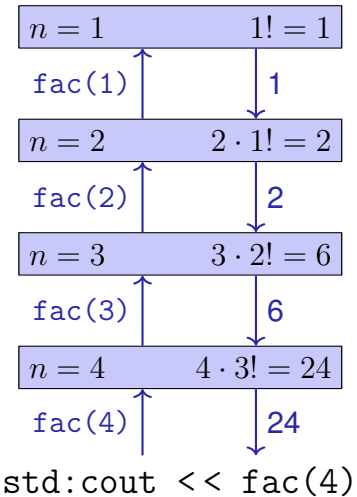
```
// POST: return value is n!
unsigned int fac (unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialisierung des formalen Arguments: $n = 4$
Rekursiver Aufruf mit Argument $n - 1 == 3$

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
- 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);
}
```

Terminierung: $a \bmod b < b$, also wird b in jedem rekursiven Aufruf kleiner.

Fibonacci-Zahlen

$$F_n := \begin{cases} 0, & \text{falls } n = 0 \\ 1, & \text{falls } n = 1 \\ F_{n-1} + F_{n-2}, & \text{falls } n > 1 \end{cases}$$

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \ldots$

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
}
```

Korrektheit und Terminierung sind klar.

Schnelle Fibonacci-Zahlen

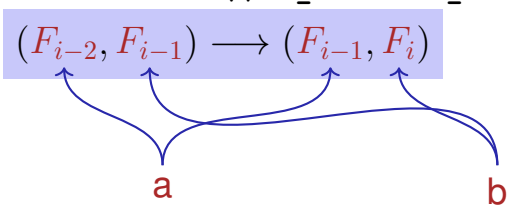
Idee:

- 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;
}
```

sehr schnell auch bei `fib(50)`



Rekursion und Iteration

Rekursion kann *immer* simuliert werden durch

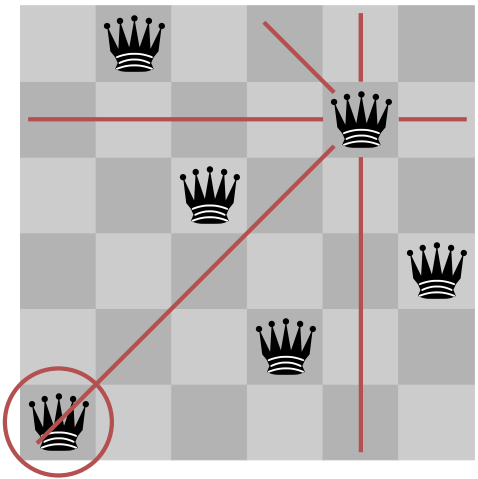
- Iteration (Schleifen)
- expliziten „Aufrufstapel“ (z.B. Feld).

Oft sind rekursive Formulierungen einfacher, aber manchmal auch weniger effizient.

Die Macht der Rekursion

- Einige Probleme scheinen ohne Rekursion kaum lösbar zu sein. Mit Rekursion werden sie plötzlich einfacher lösbar.
- Beispiele: *das n-Damen-Problem*, Die Türme von Hanoi, Parsen von Ausdrücken, *Sudoku-Löser*, Umgekehrte Aus- oder Eingabe, Suchen in Bäumen, Divide-And-Conquer (z.B. Sortieren)

Das n-Damen Problem

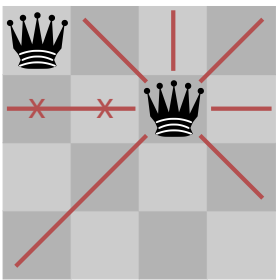


- Gegeben sei ein $n \times n$ Schachbrett
- Zum Beispiel $n = 6$
- Frage: ist es möglich n Damen so zu platzieren, dass keine zwei Damen sich bedrohen?
- Wenn ja, wie viele Lösungen gibt es?

Lösung?

- Durchprobieren aller Möglichkeiten?
- $\binom{n^2}{n}$ Möglichkeiten. Zu viele!
- Nur eine Dame pro Zeile. n^n Möglichkeiten, besser – aber auch noch zu viele.
- Idee: Unsinnige Pfade nicht weiterverfolgen. (Backtracking)

Lösung mit Backtracking

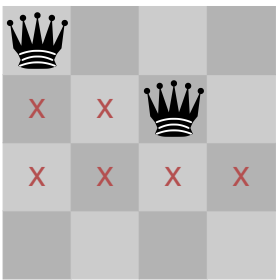


Nächste Dame in nächster Zeile (keine Kollision)

queens

0
2
0
0

Lösung mit Backtracking

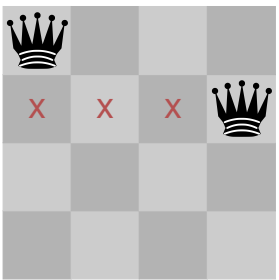


Alle Felder in nächster Zeile verboten. Zurück! (Backtracking!)

queens

0
2
4
0

Lösung mit Backtracking

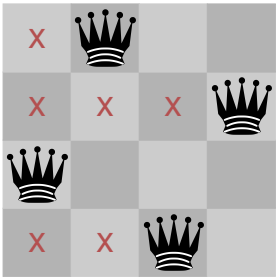


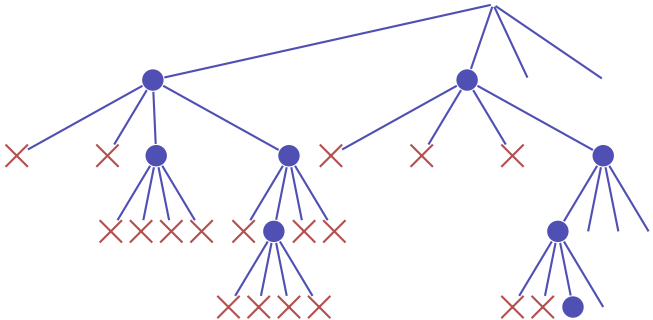
Dame eins weiter setzen und wieder versuchen

queens

0
3
0
0

Suchstrategie als Baum visualisiert





Prüfe Dame

```
using Queens = std::vector<unsigned int>;

// post: returns if queen in the given row is valid, i.e.
//       does not share a common row, column or diagonal
//       with any of the queens on rows 0 to row−1
bool valid(const Queens& queens, unsigned int row){
  unsigned int col = queens[row];
  for (unsigned int r = 0; r != row; ++r){
    unsigned int c = queens[r];
    if (col == c || col − row == c0 − r || col + row == c + r)
      return false; // same column or diagonal
  }
  return true; // no shared column or diagonal
}
```

Rekursion: Finde eine Lösung

```
// pre: all queens from row 0 to row−1 are valid,
//      i.e. do not share any common row, column or diagonal
// post: returns if there is a valid position for queens on
//      row .. queens.size(). if true is returned then the
//      queens vector contains a valid configuration.
bool solve(Queens& queens, unsigned int row){
  if (row == queens.size())
    return true;
  for (unsigned int col = 0; col != queens.size(); ++col){
    queens[row] = col;
    if (valid(queens, row) && solve(queens,row+1))
        return true; // (else check next position)
  }
  return false; // no valid configuration found
}
```

Rekursion: Zähle alle Lösungen

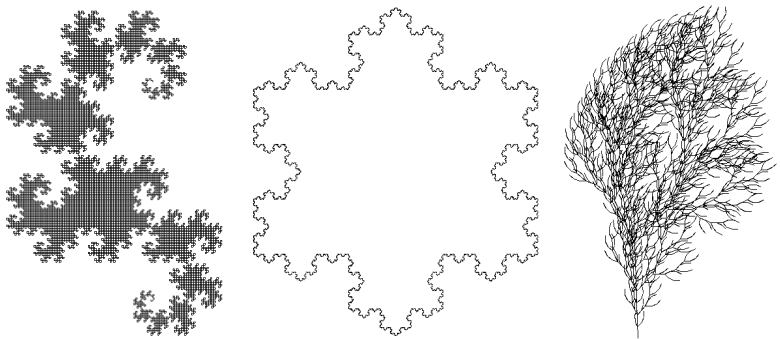
```
// pre: all queens from row 0 to row−1 are valid,
//   i.e. do not share any common row, column or diagonal
// post: returns the number of valid configurations of the
//   remaining queens on rows row ... queens.size()
int nSolutions(Queens& queens, unsigned int row){
  if (row == queens.size())
    return 1;
  int count = 0;
  for (unsigned int col = 0; col != queens.size(); ++col){
    queens[row] = col;
    if (valid(queens, row))
      count += nSolutions(queens,row+1);
  }
  return count;
}
```

Hauptprogramm

```
// pre: positions of the queens in vector queens
// post: output of the positions of the queens in a graphical way
void print(const Queens& queens);

int main(){
  int n;
  std::cin >> n;
  Queens queens(n);
  if (solve(queens,0)){
    print(queens);
    std::cout << "# solutions:" << nSolutions(queens,0) << std::endl;
  } else
    std::cout << "no solution" << std::endl;
  return 0;
}
```

Lindenmayer-Systeme (L-Systeme)

Fraktale aus Strings und Schildkröten



L-Systeme wurden vom ungarischen Biologen Aristid Lindenmayer (1925–1989) zur Modellierung von Pflanzenwachstum erfunden.

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

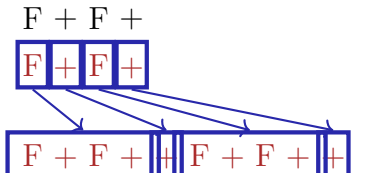
Das Tripel $\mathcal{L} = (\Sigma, P, s_0)$ ist ein L-System.

Die beschriebene Sprache

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

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

$$
\begin{array}{lcl}
w_0 & := & s_0 \\
w_1 & := & P(w_0) \\
w_2 & := & P(w_1) \\
 & \vdots &
\end{array}
\qquad
\begin{array}{lcl}
w_0 & := & \mathrm{F} \\
 & & \mathrm{F}+\mathrm{F}+ \\
w_1 & := & \mathrm{F}+\mathrm{F}+ \\
 & & P(\mathrm{F})P(+)P(\mathrm{F})P(+) \\
w_2 & := & \mathrm{F}+\mathrm{F}+\,+\,\mathrm{F}+\mathrm{F}+\,+ \\
 & \vdots &
\end{array}
$$

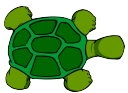


Definition

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

Turtle-Grafik

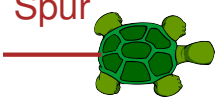
Schildkröte mit Position und Richtung



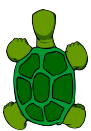
Schildkröte versteht 3 Befehle:

F: Gehe einen Schritt vorwärts ✓

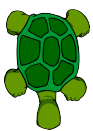
Spur



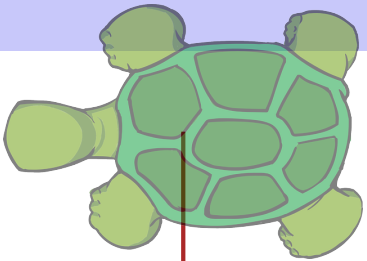
$+$: Drehe dich um 90 Grad ✓



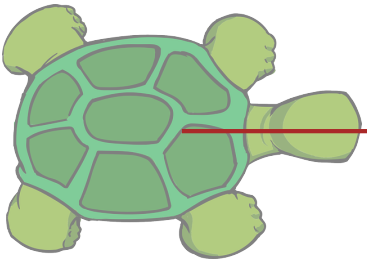
$-$: Drehe dich um -90 Grad ✓



Wörter zeichnen!



$w_1 = \mathrm{F} + \mathrm{F} + \checkmark$



lindenmayer: Hauptprogramm

Wort $w_0 \in \Sigma^*$:

```
int main () {
  std::cout << "Maximal Recursion Depth =? ";
  unsigned int n;
  std::cin >> n;

  std::string w = "F"; // w_0
  produce(w,n);

  return 0;
}
```

$w = w_0 = \mathrm{F}$

lindenmayer: production

```
// POST: recursively iterate over the production of the characters
//       of a word.
//       When recursion limit is reached, the word is "drawn"
void produce(std::string word, int depth){
  if (depth > 0){
    for (unsigned int k = 0; k < word.length(); ++k)
      produce(replace(word[k]), depth−1);
  } else {
    draw_word(word);
  }
}
```

$w = w_i \rightarrow w = w_{i+1}$

Zeichne $w = w_n$!

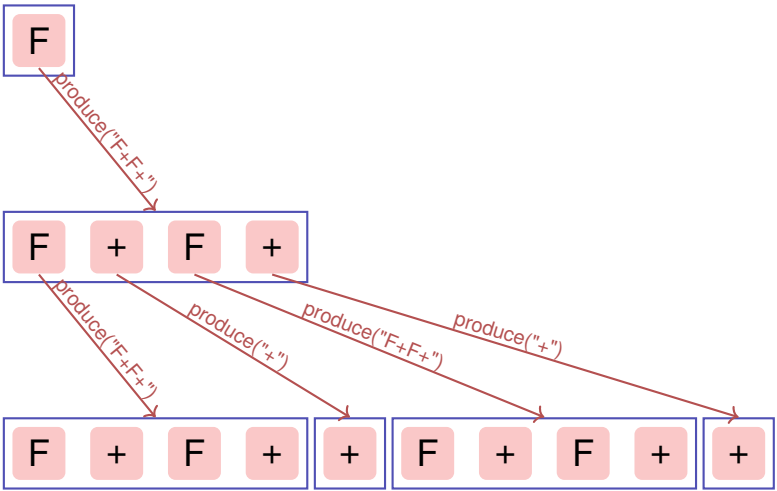
lindenmayer: replace

```
// POST: returns the production of c
std::string replace (const char c)
{
  switch (c) {
  case 'F':
    return "F+F+";
  default:
    return std::string (1, c); // trivial production c -> c
  }
}
```

lindenmayer: draw

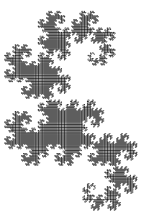
```
// POST: draws the turtle graphic interpretation of word
void draw_word (const std::string& word)
{
  for (unsigned int k = 0; k < word.length(); ++k)
    switch (word[k]) {
    case 'F':
      turtle::forward(); // move one step forward
      break;
    case '+':
      turtle::left(90); // turn counterclockwise by 90 degrees
      break;
    case '-':
      turtle::right(90); // turn clockwise by 90 degrees
    }
}
```

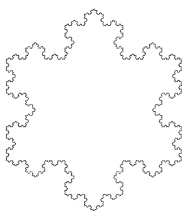
Die Rekursion



L-Systeme: Erweiterungen

- Beliebige Symbole ohne grafische Interpretation (dragon)
- Beliebige Drehwinkel (snowflake)
- Sichern und Wiederherstellen des Schildkröten-Zustandes $\rightarrow$ Pflanzen (bush)





15. Rekursion 2

Bau eines Taschenrechners, Formale Grammatiken, Extended Backus Naur Form (EBNF), Parsen von Ausdrücken

Motivation: Taschenrechner

Ziel: Bau eines Kommandozeilenrechners

Beispiel

Eingabe: `3 + 5`
Ausgabe: `8`
Eingabe: `3 / 5`
Ausgabe: `0.6`
Eingabe: `3 + 5 * 20`
Ausgabe: `103`
Eingabe: `(3 + 5) * 20`
Ausgabe: `160`
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";
```

```
Eingabe 2 + 3 * 3 =
Ergebnis 15
```

Analyse des Problems

Beispiel

Eingabe:

$$\underbrace{13 + 4 * (15 - 7*}_{} 3) =$$

Muss gespeichert bleiben, damit jetzt ausgewertet werden kann!

Beispiel

Diese Vorlesung ist insgesamt recht rekursiv.

Analyse des Problems

$$13 + 4 * (15 - 7 * 3)$$

Das “Verstehen” eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Wir werden die Symbole elegant mittels Rekursion zwischenspeichern.

Wir brauchen ein neues formales (von C++unabhängiges) Handwerkszeug.

Formale Grammatiken

- Alphabet: endliche Menge von Symbolen
- Sätze: endlichen Folgen von Symbolen

Eine formale Grammatik definiert, welche Sätze gültig sind.

Zur Beschreibung der Grammatik verwenden wir:

Extended Backus Naur Form (EBNF)

What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?

Niklaus Wirth
Federal Institute of Technology (ETH), Zürich, and Xerox Palo Alto Research Center

Key Words and Phrases: syntactic description language, extended BNF
CR Categories: 4.20

The population of programming languages is steadily growing, and there is no end of this growth in sight. Many language definitions appear in journals, many are found in technical reports, and perhaps an even greater number remains confined to proprietory circles. After frequent exposure to these definitions, one cannot fail to notice the lack of "common denominators." The only widely accepted fact is that the language structure is defined by a syntax. But even notation for syntactic description eludes any commonly agreed standard form, although the underlying ancestor is invariably the Backus-Naur Form of the Algol 60 report. As variations are often only slight, they become annoying for their very lack of an apparent motivation.

Out of sympathy with the troubled reader who is weary of adapting to a new variant of BNF each time another language definition appears, and without any claim for originality, I venture to submit a simple notation that has proven valuable and satisfactory in use. It has the following properties to recommend it:

1. The notation distinguishes clearly between meta-, terminal, and nonterminal symbols.
2. It does not exclude characters used as metasymbols from use as symbols of the language (as e.g. "|" in BNF).
3. It contains an explicit iteration construct, and thereby avoids the heavy use of recursion for expressing simple repetition.
4. It avoids the use of an explicit symbol for the empty string (such as ⟨empty⟩ or ϵ).
5. It is based on the ASCII character set.

This meta language can therefore conveniently be used to define its own syntax, which may serve here as an example of its use. The word *identifier* is used to denote *nonterminal symbol*, and *literal* stands for *terminal symbol*. For brevity, *identifier* and *character* are not defined in further detail.

```
syntax     = {production}.
production = identifier "=" expression ".".
expression = term {"|" term}.
term       = factor {factor}.
factor     = identifier | literal | "(" expression ")" |
             "[" expression "]" | "{" expression "}".
literal    = """" character {character} """".
```

Repetition is denoted by curly brackets, i.e. {a} stands for ϵ | a | aa | aaa | . . . . Optionality is expressed by square brackets, i.e. [a] stands for a | ϵ. Parentheses merely serve for grouping, e.g. (a|b)c stands for ac | bc. Terminal symbols, i.e. literals, are enclosed in quote marks (and, if a quote mark appears as a literal itself, it is written twice), which is consistent with common practice in programming languages.

Received January 1977; revised February 1977

Author's present address: Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

Ausdrücke

```
-(3-(4-5))*(3+4*5)/6
```

Was benötigen wir in einer Grammatik?

- Zahl , (Ausdruck)
  −Zahl, −(Ausdruck)
- Faktor ∗ Faktor, Faktor
  Faktor / Faktor , ...
- Term + Term, Term
  Term − Term, ...

Faktor

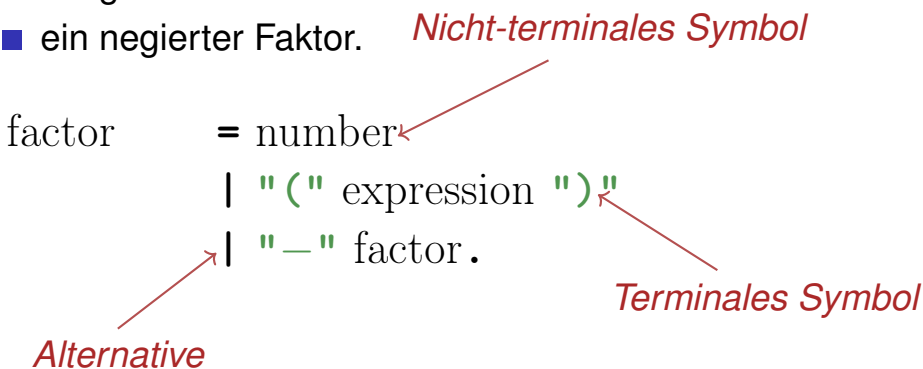
Term

Ausdruck

Die EBNF für Ausdrücke

Ein Faktor ist

- eine Zahl,
- ein geklammerter Ausdruck oder
- ein negierter Faktor.



Die EBNF für Ausdrücke

Ein Term ist

- Faktor,
- Faktor ∗ Faktor, Faktor / Faktor,
- Faktor ∗ Faktor ∗ Faktor, Faktor / Faktor ∗ Faktor, ...
- ...

term = factor { "∗" factor | "/" factor }.

Optionale Repetition

Die EBNF für Ausdrücke

```
factor     = number
           | "(" expression ")"
           | "-" factor.

term       = factor { "*" factor | "/" factor }.

expression = term { "+" term |"-" term }.
```

Zahlen

Eine *Ganzzahl hat mindestens eine Ziffer, gefolgt von beliebig vielen Ziffern.*

```
number = digit { digit }.
digit  = '0' | '1' | '2' | ... |'9'.
```

Parsen

- **Parsen:** Feststellen, ob ein Satz nach der EBNF gültig ist.
- **Parser:** Programm zum Parsen
- **Praktisch:** Aus der EBNF kann (fast) automatisch ein Parser generiert werden

Parser bauen

- 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 (ohne `number`)

```
factor     = 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 input, and the first non-whitespace character
//       input returned (0 if there input no such character)
char lookahead (std::istream& input)
{
  input >> std::ws;          // skip whitespaces
  if (input.eof())
    return 0;              // end of stream
  else
    return input.peek();     // next character in input
}
```

Rosinenpickerei

. . . um jeweils nur das gewünschte Zeichen zu extrahieren.

```
// POST: if ch matches the next lookahead then consume it and return true
//       otherwise return false
bool consume (std::istream& input, char c)
{
  if (lookahead (input) == c) {
    input >> c;
    return true;
  } else
    return false;
}
```

Faktoren auswerten

```
double factor (std::istream& input)
{
  double value;
  if (consume (input, '(')) {
    value = expression (input);        // "(" expression
    consume (input, ')');              // ")"
  } else if (consume (input, '-'))
    value = -factor (input);           // - factor
  else
    value = number(input);
  return value;
}
```

```
factor = "(" expression ")"
       | "-" factor
       | number.
```

Terme auswerten

```
double term (std::istream& input)
{
  double value = factor (input);          // factor
  while (true) {
    if (consume (input, '*'))
      value *= factor (input);            // "*" factor
    else if (consume (input, '/'))
      value /= factor (input);            // "/" factor
    else
      return value;
  }
}
```

term = factor { "*" factor | "/" factor }.

Ausdrücke auswerten

```
double expression (std::istream& input)
{
  double value = term (input);          // term
  while (true) {
    if (consume (input, '+'))
      value += term (input);            // "+" term
    else if (consume (input, '-'))
      value -= term (input);            // "-" term
    else
      return value;
  }
}
```

expression = term { "+" term | "−" term }.

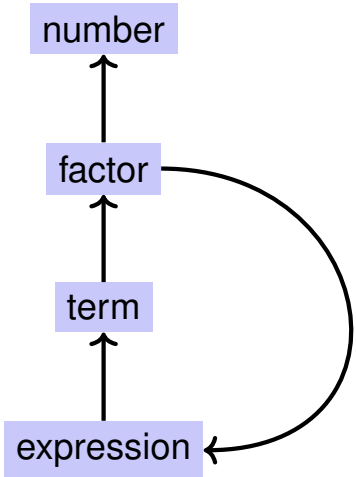
Ziffern ...

```
// POST: returns the digit that could be consumed from a stream
//       (0 if no digit available)
// digit = '0' | '1' | ... | '9'.
char digit(std::istream& input){
  char ch = input.peek(); // one symbol lookahead
  if (input.eof()) return 0; // nothing available on the stream
  if  (ch >= '0' && ch <= '9'){
    input >> ch; // consume
    return ch;
  }
  return 0;
}
```

... und Zahlen

```
// POST: returns an unsigned integer consumed from the stream
// number = digit {digit}.
unsigned int number (std::istream& input){
  input >> std::skipws;// skip whitespaces before the first digit
  char ch = digit(input);
  input >> std::noskipws; // no whitespaces allowed within a number
  unsigned int num = 0;
  while(ch > 0){ // skip remaining digits
    num = num * 10 + ch - '0';
    ch = digit(input);
  }
  return num;
}
```

Rekursion!



EBNF — Und es funktioniert!

EBNF (`calculator.cpp`, Auswertung von links nach rechts):

```
factor     = number
           | "(" expression ")"
           | "−" factor.

term       = factor { "∗" factor | "/" factor }.

expression = term { "+" term |"−" term }.
```

```
std::stringstream input ("1−2−3");
std::cout << expression (input) << "\n"; // −4
```

16. Structs

Rationale Zahlen, Struct-Definition, Funktions- und Operatorüberladung

Rechnen mit rationalen Zahlen

- Rationale Zahlen ($\mathbb{Q}$) sind von der Form $\frac{n}{d}$ mit n und d in $\mathbb{Z}$
- $\mathrm{C{+}{+}}$hat keinen „eingebauten” Typ für rationale Zahlen

Ziel

Wir bauen uns selbst einen $\mathrm{C{+}{+}}$-Typ für rationale Zahlen! 🙂

Vision

So könnte (wird) es aussehen

```
// 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

Invariante: spezifiziert gültige Wertkombinationen (informell).

```
struct rational {
  int n;
  int d; // INV: d != 0
};
```

Member-Variable (**n**umerator)

Member-Variable (**d**enominator)

- `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;
}
```

$$\frac{r_n}{r_d} := \frac{a_n}{a_d} + \frac{b_n}{b_d} = \frac{a_n \cdot b_d + a_d \cdot b_n}{a_d \cdot b_d}$$

Ein erstes Struct: Funktionalität

Ein `struct` definiert einen *Typ*, keine *Variable*!

```
// new type rational
struct rational {
    int n;
    int d; // INV: d != 0
};

// POST: return value is the sum of a and b
rational add (const rational a, const rational b)
{
  rational result;
  result.n =  a.n * b.d +  a.d * b.n;
  result.d = a.d * b.d;
  return result;
}
```

Bedeutung: jedes Objekt des neuen Typs ist durch zwei Objekte vom Typ `int` repräsentiert, die die Namen `n` und `d` tragen.

Member-Zugriff auf die `int`-Objekte von `a`.

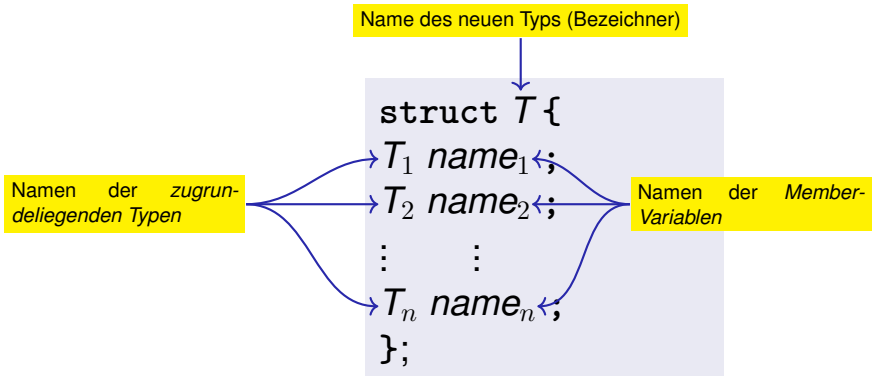
Eingabe

```
// 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



Wertebereich von T: $T_1 \times T_2 \times ... \times T_n$

Struct-Definitionen: Beispiele

```
struct rational_vector_3 {
  rational x;
  rational y;
  rational z;
};
```

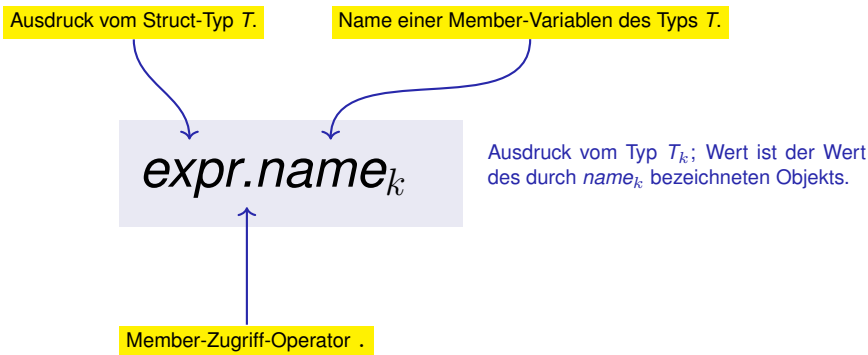
Zugrundeliegende Typen können fundamentale aber auch benutzerdefinierte Typen sein.

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: Member-Zugriff



Ausdruck vom Typ T_k; Wert ist der Wert des durch $name_k$ bezeichneten Objekts.

Structs: Initialisierung und Zuweisung

Default-Initialisierung:

```
rational t;
```

- Member-Variablen von `t` werden default-initialisiert
- für Member-Variablen fundamentaler Typen passiert dabei nichts (Wert undefiniert)

Structs: Initialisierung und Zuweisung

Initialisierung:

```
rational t = {5, 1};
```

- Member-Variablen von `t` werden mit den Werten der Liste, entsprechend der Deklarationsreihenfolge, initialisiert.

Structs: Initialisierung und Zuweisung

Zuweisung:

```
rational s;
...
rational t = s;
```

- Den Member-Variablen von `t` werden die Werte der Member-Variablen von `s` zugewiesen.

Structs: Initialisierung und Zuweisung

```
t.n          .n
t.d = add (r, s)  .d ;
```

Initialisierung:

```
rational t = add (r, s);
```

- `t` wird mit dem Wert von `add(r, s)` initialisiert

Structs: Initialisierung und Zuweisung

Zuweisung:

```
rational t;
t = add (r, s);
```

- `t` wird default-initialisiert
- Der Wert von `add (r, s)` wird `t` zugewiesen

Structs: Initialisierung und Zuweisung

```
rational s;
```
← Member-Variablen uninitialisiert (wird sich bald ändern)

```
rational t = {1,5};
```
← *Memberweise* Initialisierung: `t.n = 1, t.d = 5`

```
rational u = t;
```
← Memberweise Kopie

```
t = u;
```
← Memberweise Kopie

```
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?

- Memberweiser Vergleich ergibt im allgemeinen keinen Sinn,...
- ...denn dann wäre z.B. $\frac{2}{3} \neq \frac{4}{6}$

Structs als Funktionsargumente

```
void increment(rational dest, const rational src)
{
    dest = add (dest, src); // veraendert nur lokale Kopie
}
```

Call by Value !

```
rational a;
rational b;
a.d = 1; a.n = 2;
b = a;
increment (b, a); // kein Effekt!
std::cout << b.n << "/" << b.d; // 1 / 2
```

Structs als Funktionsargumente

```
void increment(rational & dest, const rational src)
{
    dest = add (dest, src);
}
```

Call by Reference

```
rational a;
rational b;
a.d = 1; a.n = 2;
b = a;
increment (b, a);
std::cout << b.n << "/" << b.d; // 2 / 2
```

Benutzerdefinierte Operatoren

Statt

```
rational t = add(r, s);
```

würden wir lieber

```
rational t = r + s;
```

schreiben.

Das geht mit *Operator-Überladung*.

Überladen von Funktionen

- Funktionen sind durch Ihren Namen im Gültigkeitsbereich ansprechbar
- Es ist sogar möglich, mehrere Funktionen des gleichen Namens zu definieren und zu deklarieren
- Die „richtige” Version wird aufgrund der *Signatur* der Funktion ausgewählt

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” (wir vertiefen das nicht)

```
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 (Operator Overloading)

- Operatoren sind spezielle Funktionen und können auch überladen werden
- Name des Operators *op*:

```
operatorop
```

- Wir wissen schon, dass z.B. `operator+` für verschiedene Typen existiert

`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;
```

Infix-Notation

Andere binäre Operatoren für rationale Zahlen

```
// POST: return value is difference of a and b
rational operator- (rational a, rational b);

// POST: return value is the product of a and b
rational operator* (rational a, rational b);

// POST: return value is the quotient of a and b
// PRE: b != 0
rational operator/ (rational a, rational b);
```

Unäres Minus

Hat gleiches Symbol wie binäres Minus, aber nur ein Argument:

```
// POST: return value is -a
rational operator- (rational a)
{
    a.n = -a.n;
    return a;
}
```

Vergleichsoperatoren

Sind für Structs nicht eingebaut, können aber definiert werden:

```
// 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+= Erster Versuch

```
rational operator+= (rational a, rational b)
{
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

Das funktioniert nicht! Warum?

- Der Ausdruck `r += s` hat zwar den gewünschten Wert, weil die Aufrufargumente R-Werte sind (call by value!) jedoch nicht den gewünschten Effekt der Veränderung von `r`.
- Das Resultat von `r += s` stellt zudem entgegen der Konvention von C++ keinen L-Wert dar.

Operator +=

```
rational& operator+= (rational& a, rational b)
{
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

Das funktioniert!

- Der L-Wert `a` wird um den Wert von `b` erhöht und als L-Wert zurückgegeben.

`r += s;` hat nun den gewünschten Effekt.

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;
}
```

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";
```

operator >>

operator +

operator<<

17. Klassen

Datenkapselung, Klassen, Memberfunktionen, Konstruktoren

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
- Lehre
- Technologietransfer

Wir gründen die Startup-Firma RAT PACK®!

- Verkauf der `rational`-Bibliothek an Kunden
- Weiterentwicklung nach Kundenwünschen

Der Kunde ist zufrieden

. . . und programmiert fleissig mit `rational`.

- Ausgabe als `double`-Wert $(\frac{3}{5} \to 0.6)$

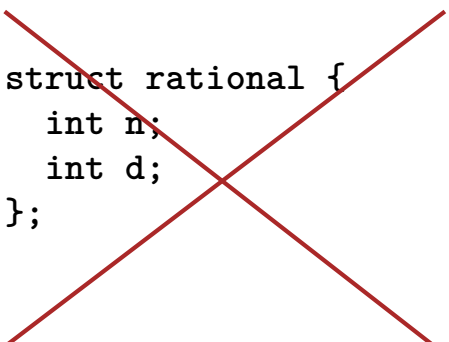
```
// POST: double approximation of r
double to_double (rational r)
{
  double result = r.n;
  return result / r.d;
}
```

Der Kunde will mehr

“Können wir rationale Zahlen mit erweitertem Wertebereich bekommen?”

- 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!

- 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;
}
```

`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`)
- 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.
- 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++
- 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.

Einziger Unterschied:

- `struct`: standardmässig wird *nichts* versteckt
- `class` : standardmässig wird *alles* versteckt

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 instance
  int numerator () const {
    return n;
  }
  // POST: return value is the denominator of this instance
  int denominator () const {
    return d;
  }
private:
   int n;
   int d; // INV: d!= 0
};
```

öffentlicher Bereich

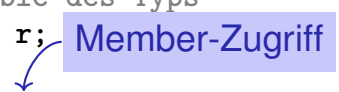
Memberfunktion

Memberfunktionen haben Zugriff auf private Daten

Gültigkeitsbereich von Membern in einer Klasse ist die ganze Klasse, unabhängig von der Deklarationsreihenfolge

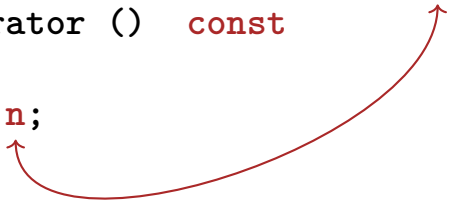
Memberfunktionen: Aufruf

```
// 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 instance
int numerator ()  const
{
  return n;
}
```



- Eine Memberfunktion wird für einen Ausdruck der Klasse aufgerufen. In der Funktion: `this` ist der Name dieses impliziten Arguments. `this` selbst ist ein Zeiger darauf.
- Das const bezieht sich auf die Instanz `this`, verspricht also, dass das implizite Argument nicht im Wert verändert wird.
- `n` ist Abkürzung in der Memberfunktion für `this->n` (genaue Erklärung von „`->`“ nächste Woche)

`const` und Memberfunktionen

```
class rational {
public:
  int numerator () const
  { return n; }
  void set_numerator (int N)
  { n = N;}
...
}
```

```
rational x;
x.set_numerator(10); // ok;
const rational y = x;
int n = y.numerator(); // ok;
y.set_numerator(10); // error;
```

Das `const` an einer Memberfunktion liefert das Versprechen, dass eine Instanz nicht über diese Funktion verändert wird.

`const` Objekte dürfen nur `const` Memberfunktionen aufrufen!

This rational vs. dieser Bruch

So wäre es **in etwa** ...

```
class rational {
    int n;
    ...
public:
    int numerator () const
    {
        return this->n;
    }
};

rational r;
...
std::cout << r.numerator();
```

... ohne Memberfunktionen

```
struct bruch {
    int n;
    ...
};

int numerator (const bruch& dieser)
{
    return dieser.n;
}

bruch r;
..
std::cout << numerator(r);
```

Member-Definition: In-Class vs. Out-of-Class

```
class rational {
    int n;
    ...
public:
    int numerator () const
    {
        return n;
    }
    ....
};
```

- Keine Trennung zwischen Deklaration und Definition (schlecht für Bibliotheken)

```
class rational {
    int n;
    ...
public:
    int numerator () const;
    ...
};

int rational::numerator () const
{
  return n;
}
```

- So geht's auch.

Konstruktoren

- sind spezielle *Memberfunktionen* einer Klasse, die den Namen der Klasse tragen.
- können wie Funktionen überladen werden, also in der Klasse mehrfach, aber mit verschiedener *Signatur* vorkommen.
- werden bei der Variablendeklaration wie eine Funktion aufgerufen. Der Compiler sucht die „naheliegendste“ passende Funktion aus.
- wird kein passender Konstruktor gefunden, so gibt der Compiler eine *Fehlermeldung* aus.

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.

Konstruktoren: Aufruf

- direkt

```
rational r (1,2); // initialisiert r mit 1/2
```

- indirekt (Kopie)

```
rational r = rational (1,2);
```

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
```

Benutzerdefinierte Konversionen

sind definiert durch Konstruktoren mit genau *einem* Argument

```
rational (int num)
    : n (num), d (1)
  {}
```

Benutzerdefinierte Konversion von `int` nach `rational`. Damit wird `int` zu einem Typ, dessen Werte nach `rational` konvertierbar sind.

```
rational r = 2; // implizite Konversion
```

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!

Alternative: Default-Konstruktor löschen

```
class rational
{
public:
    ...
    rational () = delete;
...
};
...
rational r;     // error: use of deleted function 'rational::rational()
```

$\Rightarrow$ Es gibt keine uninitialisierten Variablen vom Typ rational mehr!

Der Default-Konstruktor

- wird automatisch aufgerufen bei Deklarationen der Form `rational r;`
- ist der eindeutige Konstruktor mit leerer Argumentliste (falls existent)
- muss existieren, wenn `rational r;` kompilieren soll
- wenn in einem Struct keine Konstruktoren definiert wurden, wird der Default-Konstruktor automatisch erzeugt (wegen der Sprache C)

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;
};
```

```
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;
  }
}
```

- 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
};
```

- Wir legen uns auf Zähler-/Nennertyp `int` fest.
- Lösung: Nicht nur Daten, auch Typen kapseln.

Fix: „Unser“ Typ `rational::integer`

Die Sicht des Kunden (`rational.h`):

```
public:
   using integer = long int; // might change
   // POST: returns numerator of *this
   integer numerator () const;
```

- Wir stellen einen eigenen Typ zur Verfügung!
- Festlegung nur auf Funktionalität, z.B.:
  - implizite Konversion `int` $\rightarrow$ `rational::integer`
  - Funktion `double to_double (rational::integer)`

RAT PACK® Revolutions

Endlich ein Kundenprogramm, das stabil bleibt:

```
// POST: double approximation of r
double to_double (const rational r)
{
  rational::integer n = r.numerator();
  rational::integer d = r.denominator();
  return to_double (n) / to_double (d);
}
```

Deklaration und Definition getrennt

```
class rational {
public:
   rational (int num, int denum);
   using integer = long int;
   integer numerator () const;
   ...
private:
 ...
};
```

rational.h

```
rational::rational (int num, int den):
    n (num), d (den) {}
rational::integer rational::numerator () const
{
    return n;
}
```

rational.cpp

Klassenname :: Membername

18. Dynamische Datenstrukturen I

Dynamischer Speicher, Adressen und Zeiger, Const-Zeiger, Arrays, Array-basierte Vektoren

Wiederholung: `vector<`T`>`

- Kann mit beliebiger Grösse `n` initialisiert werden
- Unterstützt diverse Operationen:

```
e = v[i];          // Get element
v[i] = e;          // Set element
l = v.size();      // Get size
v.push_front(e); // Prepend element
v.push_back(e);  // Append element
...
```

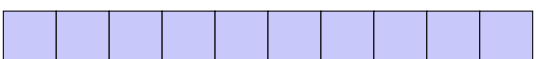
- Ein Vektor ist eine *dynamische Datenstruktur*, deren Grösse sich zur Laufzeit ändern kann

Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<`T`>` (später, nur kurz angeschnitten)

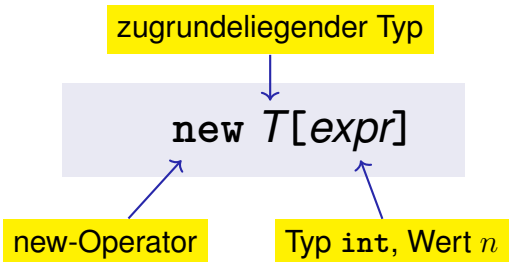
Vektoren im Speicher

Bereits bekannt: Ein Vektor belegt einen *zusammenhängenden* Speicherbereich



Frage: Wie *alloziert* (reserviert) man einen Speicherblock *beliebiger* Grösse zur Laufzeit, also *dynamisch*?

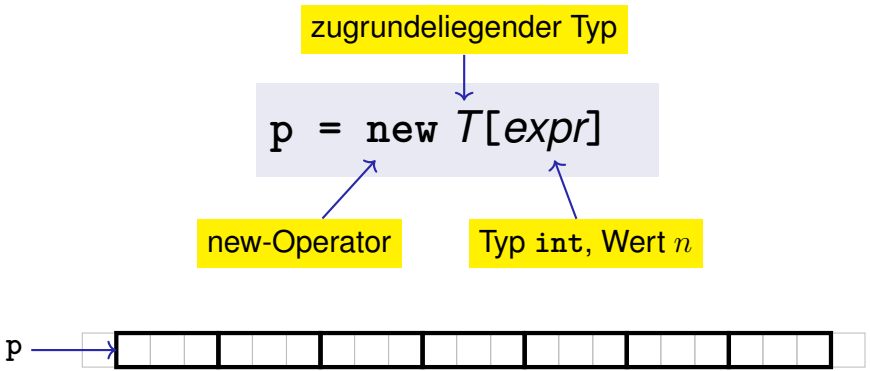
Der `new`-Ausdruck für Arrays



- **Effekt**: Neuer zusammenhängender Bereich im Speicher für n Elemente vom Typ *T* wird alloziert

- Dieser Speicherbereich wird *Array* (der Länge n) genannt

Der `new`-Ausdruck für Arrays



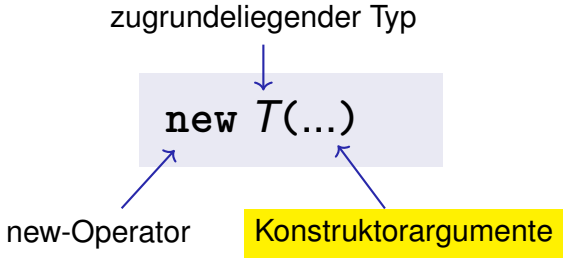
- **Typ**: Ein Zeiger $T*$ (mehr dazu gleich)
- **Wert**: Die Startadresse des Speicherbereichs

Ausblick: `new` und `delete`

`new` *T*`[`*expr*`]`

- Bisher: Speicher (lokale Variablen, Funktionsargumente) „lebt“ nur innerhalb eines Funktionsaufrufs
- Aber jetzt: Speicherblock im Vektor darf nicht vor dem Vektor selbst „sterben“
- Mittels `new` allozierter Speicher wird *nicht* automatisch deallokiert (= freigegeben)
- Zu jedem `new` gehört ein `delete`, das den Speicher explizit freigibt → **in zwei Wochen**

Der `new`-Ausdruck (ohne Arrays)



- **Effekt**: Speicher für ein neues Objekt vom Typ T wird alloziert ...
- ... und mit Hilfe des passenden Konstruktors initialisiert
- **Wert**: Adresse des neuen T-Objekt, **Typ**: Zeiger $T*$
- Auch hier gilt: Objekt „lebt“ bis es explizt gelöscht wird (Nutzen wird später klarer werden)

Zeiger-Typen

`T*` Zeiger-Typ für zugrunde liegenden Typ `T`

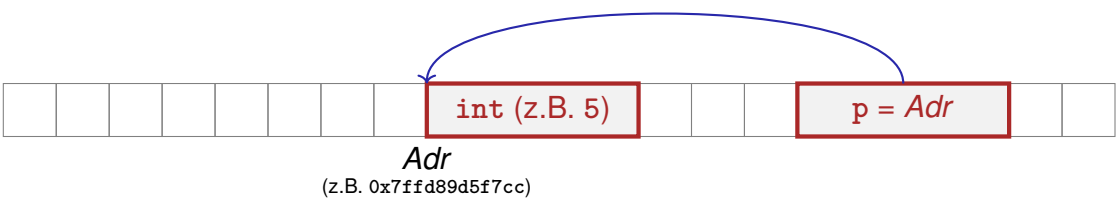
Ein Ausdruck vom Typ `T*` heisst *Zeiger (auf `T`)*

```
int* p; // Zeiger auf einen int
std::string* q; // Zeiger auf einen std::string
```

Zeiger-Typen

Wert eines Zeigers auf `T` ist die *Adresse* eines Objektes vom Typ `T`

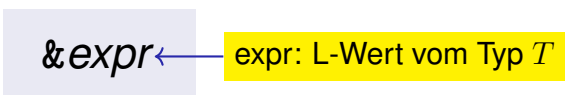
```
int* p = ...;
std::cout << p; // z.B. 0x7ffd89d5f7cc
```



Adress-Operator

Frage: Wie kommt man an die Adresse eines Objekts?

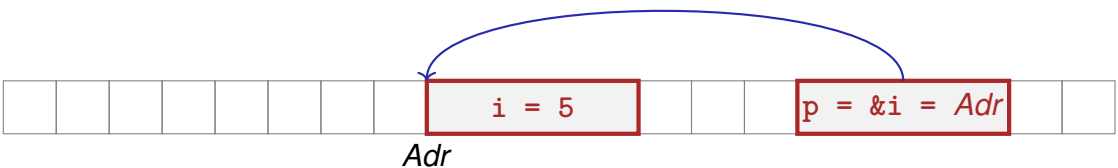
1. Direkt beim Erzeugen eines neuen Objekts mittels `new`
2. Für bestehende Objekte: Mittels des *Adress-Operators* `&`



- **Wert** des Ausdrucks: Die *Adresse* des Objekts (L-Wertes) *expr*
- **Typ** des Ausdrucks: Ein Zeiger $T*$ (vom Typ T)

Adress-Operator

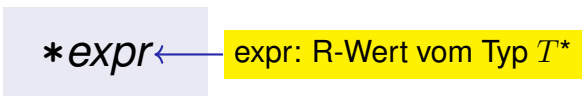
```
int i = 5; // i initialisiert mit 5
!1int* p = &i; // p initialisiert mit Adresse von i
```



Nächste Frage: Wie „folgt“ man einem Zeiger?

Dereferenz-Operator

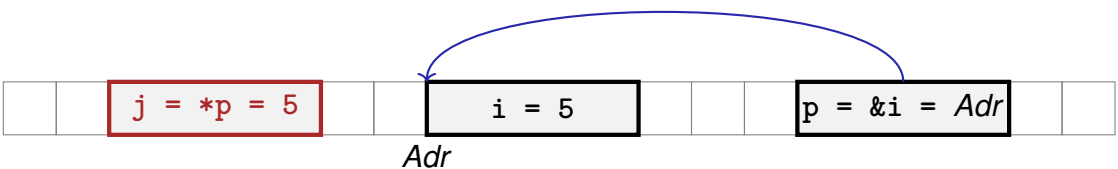
Antwort: Mittels des *Dereferenz-Operators* $*$



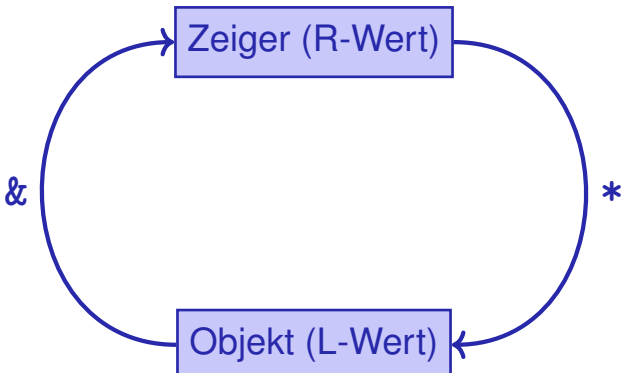
- **Wert** des Ausdrucks: Der *Wert* des Objekts an der durch *expr* bestimmten Adresse
- **Typ** des Ausdrucks: T

Dereferenz-Operator

```
int i = 5;
int* p = &i; // p = Adresse von i
!1int j = *p; // j = 5
```



Adress- und Dereferenzoperator



Zeiger-Typen

Wo ein $T*$ draufsteht, muss auch ein T drin sein

```
int* p = ...;  // p zeigt auf einen int
double* q = p; // aber q auf einen double →
   Kompilerfehler!
```

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!

Null-Zeiger

- Spezieller Zeigerwert, der angibt, dass (noch) auf kein Objekt gezeigt wird
- Repräsentiert durch das Literal `nullptr` (konvertierbar nach `T*`)

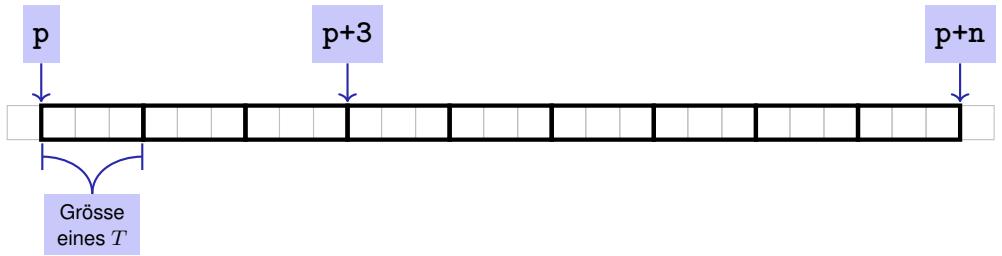
```
int* p = nullptr;
```

- Kann nicht dereferenziert werden (Laufzeitfehler)
- Dient der Vermeidung undefinierten Verhaltens

```
int* p; // p could point to anything
int* q = nullptr; // q explicitly points nowhere
```

Zeiger-Arithmetik: Zeiger plus `int`

```
T* p = new T[n]; // p points to first array element
```

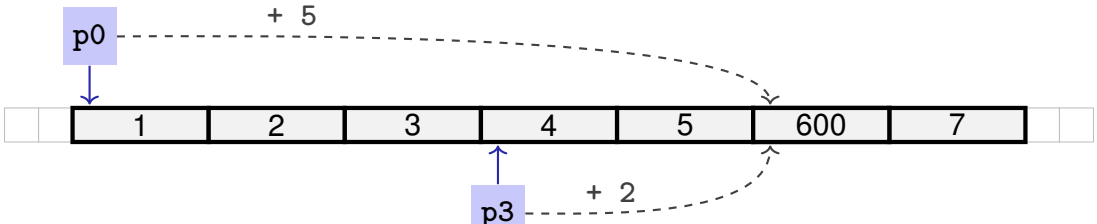


Wie zeigt man auf hintere Elemente? $\rightarrow$ *Zeiger-Arithmetik*:

- `p` gibt die *Adresse* des *ersten* Array-Elements, `*p` dessen *Wert*
- `*(p + i)` gibt den Wert des `i`-*ten* Array-Elements, für $0 \leq$ `i` $<$ `n`
- `*p` ist äquivalent zu `*(p + 0)`

Zeiger-Arithmetik: Zeiger plus `int`

```
int* p0 = new int[7]{1,2,3,4,5,6,7}; // p0 points to
   1st element
!1int* p3 = p0 + 3; // p3 points to 4th element
!1−2*(p3 + 2) = 600; // set value of 6th element to
   600
!1−3std::cout << *(p0 + 5); // output 6th element's
   value (i.e. 600)
```



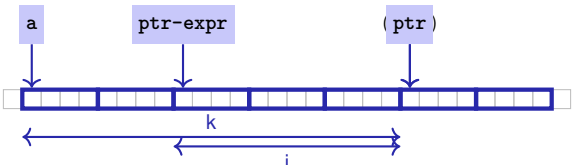
Zeiger-Arithmetik: Zeiger minus `int`

- Wenn *ptr* ein Zeiger auf das Element mit Index k in einem Array a der Länge n ist
- und der Wert von *expr* eine ganze Zahl i ist, $0 \leq k - i \leq n$,

dann liefert der Ausdruck

ptr - expr

einen Zeiger zum Element von a mit Index $k - i$.



Zeigersubtraktion

- Wenn *p1* und *p2* auf Elemente desselben Arrays `a` mit Länge n zeigen
- und $0 \leq k_1, k_2 \leq n$ die Indizes der Elemente sind, auf die *p1* und *p2* zeigen, so gilt

p1 - p2 hat den Wert k_1 - k_2

Nur gültig, wenn *p1* und *p2* ins gleiche Array zeigen.

- Die Zeigerdifferenz beschreibt, „wie weit die Elemente voneinander entfernt sind“

Zeigeroperatoren

Beschreibung	Op	Stelligkeit	Prä-zedenz	Assoziativität	Zuordnung
Subskript	`[]`	2	17	links	R-Werte → L-Wert
Dereferenzierung	`*`	1	16	rechts	R-Wert → L-Wert
Adresse	`&`	1	16	rechts	L-Wert → R-Wert

Präzedenzen und Assoziativitäten von +, −, ++ (etc.) wie in Kapitel 2

Zeigerwerte sind keine Ganzzahlen

- Adressen können als „Hausnummern des Speichers“, also als Zahlen interpretiert werden.
- Ganzzahl- und Zeigerarithmetik verhalten sich aber unterschiedlich.

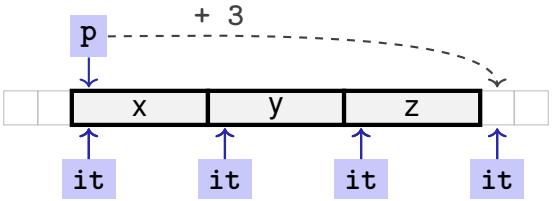
`ptr + 1` ist *nicht* die nächste Hausnummer, sondern die s-nächste, wobei s der Speicherbedarf eines Objekts des Typs ist, der `ptr` zugrundeliegt.

- Zeiger und Ganzzahlen sind nicht kompatibel:

```
int* ptr = 5; // Fehler: invalid conversion from int to int*
int a = ptr;  // Fehler: invalid conversion from int* to int
```

Sequenzielle Zeiger-Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```




```
for (3char* it = p;
     4,7,10,13it != p +
     ++it){

  std::cout << *it << ' ';
}
```

`it` zeigt aufs erste Element

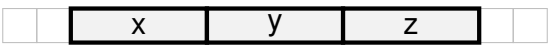
Abbruch falls Ende erreicht

Zeiger elementweise voranschieben

Aktuelles Element ausgeben: 'x'

Wahlfreier Zugriff auf Arrays

```
char* p = new char[3]{'x', 'y', 'z'};
```



- Der Ausdruck `*(p + i)`
- kann auch geschrieben werden als `p[i]`
- z.B. `p[1] == *(p + 1) == 'y'`

Wahlfreier Zugriff auf Arrays

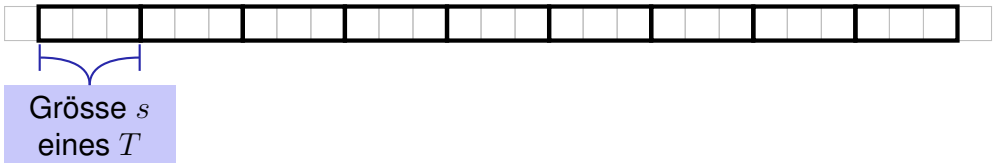
Iteration über ein Array mittels Indizes und *wahlfreiem Zugriff*:

```
char* p = new char[3]{'x', 'y', 'z'};

for (int i = 0; i < 3; ++i)
  std::cout << p[i] << ' ';
```

Aber: Dies ist weniger *effizient* als der vorher gezeigte *sequenzielle* Zugriff mittels Zeiger-Iteration

Wahlfreier Zugriff auf Arrays

```
T* p = new T[n];
```



- Zugriff `p[i]`, also `*(p + i)`, „kostet“ Berechnung $p + i \cdot s$
- Iteration mittels *wahlfreiem Zugriff* (`p[0]`, `p[1]`, ...) kostet eine Addition und eine Multiplikation pro Zugriff
- Iteration mittels *sequentiellem Zugriff* (`++p`, `++p`, ...) kostet nur eine Addition pro Zugriff
- Sequenzieller Zugriff ist daher für Iterationen zu bevorzugen

Ein Buch lesen . . . mit wahlfreiem Zugriff . . . mit sequenziellem Zugriff

Wahlfreier Zugriff

- öffne Buch auf S.1
- klappe Buch zu
- öffne Buch auf S.2-3
- klappe Buch zu
- öffne Buch auf S.4-5
- klappe Buch zu
- ....

Sequenzieller Zugriff

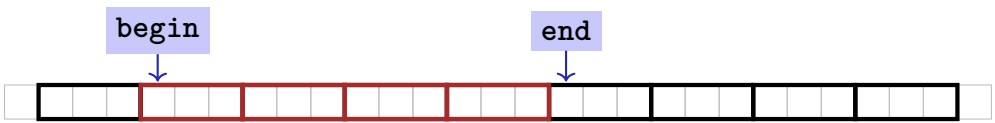
- öffne Buch auf S.1
- blättere um
- blättere um
- blättere um
- blättere um
- blättere um
- ...

Statische Arrays

- `int* p = new int[`$expr$`]` erzeugt ein dynamisches Array der Grösse $expr$
- C++hat noch *statische* Arrays von der Vorgängersprache C geerbt: `int a[`$cexpr$`]`
- Statische Arrays haben unter anderem den Nachteil, dass ihre Grösse $cexpr$ eine Konstante sein muss. D.h. $cexpr$ kann z.B. `5` oder `4*3+2` sein, aber kein von der Tastatur eingelesener Wert `n`.
- Eine statisches Array-Variable `a` kann wie ein Zeiger verwendet werden
- Faustregel: Vektoren sind besser als dynamische Arrays, welche besser als statische Arrays sind

Arrays in Funktionen

Konvention in C++: Übergabe eines Arrays (oder eines Array-Ausschnitts) mit zwei Zeigern



- `begin`: Zeiger auf das erste Element
- `end`: Zeiger *hinter* das letzte Element
- `[begin, end)` bezeichnet die Elemente des Array-Ausschnitts
- `[begin, end)` ist leer, wenn `begin == end`
- `[begin, end)` muss ein *gültiger Bereich* sein, d.h. ein echter (evtl. leerer) Array-Ausschnitt

Arrays in (mutierenden) Funktionen: `fill`

```
// PRE: [begin, end) ist ein gültiger Bereich
// POST: Jedes Element in [begin, end) wurde auf
   value gesetzt
void fill(1-int* begin, 1-int* end, int value) {
  for (int* p = begin; p != end; ++p)
    *p = value;
}


...
int* p = new int[5];
fill(2-p, 2-p+5, 1); // Array bei p wird
   zu {1, 1, 1, 1, 1}
```

Funktionen mit/ohne Effekt

- Zeiger können, wie auch Referenzen, für Funktionen mit Effekt verwendet werden. Beispiel: `fill`
- Aber viele Funktionen haben keinen Effekt, sie lesen Daten nur
- $\Rightarrow$ Verwendung von `const`
- Bisher, zum Beispiel:

```
const int zero = 0;
const int& nil = zero;
```

Positionierung von Const

Zur Determinierung der Zugehörigkeit des `const` Modifiers:

`const` T ist äquivalent zu T `const` (und kann auch so geschrieben werden):

```
const int zero = ...   ⟺   int const zero = ...
const int& nil = ...   ⟺   int const& nil = ...
```

Beide Schreibweisen werden in der Praxis genutzt

Const und Zeiger

Lies Deklaration von rechts nach links

`int const p;`	`p` ist eine konstante Ganzzahl
`int const* p;`	`p` ist ein Zeiger auf eine konstante Ganzzahl
`int* const p;`	`p` ist ein konstanter Zeiger auf eine Ganzzahl
`int const* const p;`	`p` ist ein konstanter Zeiger auf eine konstante Ganzzahl

Nicht-mutierende Funktionen: `print`

Es gibt auch *nicht* mutierende Funktionen, die nur lesend auf Elemente eines Arrays zugreifen

```
// PRE: [begin, end) ist ein gültiger Bereich
// POST: Die Werte in [begin, end) wurden ausgegeben
void print(
    2—int const* const begin,
    2—const int* const end) {

  for (3—int const* p = begin; p != end; ++p)
    std::cout << *p << ' ';
}
```

Const-Zeiger auf const-int

Ebenfalls (aber andere Schreibweise)

Zeiger, *nicht const*, auf const-int

Zeiger `p` kann selbst nicht `const` sein, da er verändert wird (`++p`)

`const` ist nicht absolut

- Der Wert an einer Adresse kann sich ändern, auch wenn ein `const`-Zeiger diese Adresse speichert.

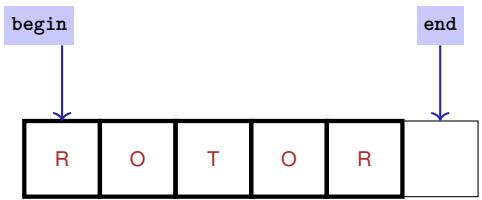
beispiel

```
int a[5];
const int* begin1 = a;
int*       begin2 = a;
*begin1 = 1;    // Fehler: *begin1 ist const
*begin2 = 1;    // ok, obwohl sich damit auch *begin1 ändert
```

- `const` ist ein Versprechen lediglich aus Sicht des `const`-Zeigers, keine absolute Garantie.

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;
}
```



Arrays, `new`, Zeiger: Abschluss

- Arrays sind kontinuierliche Speicherblöcke statisch unbestimmter Grösse
- `new` T`[`n`]` alloziert ein T-Array der Grösse n
- T`* p = new` T`[`n`]`: Zeiger `p` zeigt auf das erste Array-Element
- Zeigerarithmetik ermöglicht Zugriff auf hintere Array-Elemente
- Sequenzielle Iteration über Arrays mittels Zeigern ist effizienter als wahlfreier Zugriff
- `new` T alloziert Speicher für (und initialisiert) ein einzelnes T-Objekt und liefert einen Zeiger darauf
- Zeiger können auf etwas (nicht) `const`es zeigen und selbst (nicht) `const` sein
- Mittels `new` allozierter Speicher wird *nicht* automatisch freigegeben (mehr dazu demnächst)
- Zeiger und Referenzen sind verwandt, beide „verweisen“ auf Objekte im Speicher. Siehe auch die Extrafolien `pointers.pdf`)

Array-basierter Vektor

- Vektoren ... da war doch was 🤔
- Nun wissen wir, wie man Speicherblöcke beliebiger Grösse allozieren kann . . .
- . . . und können einen Vektor, auf einem solchen Speicherblock aufbauend, implementieren
- `avec` – ein Array-basierter Vektor für `int`-Elemente

Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<`T`>` (später, nur kurz angeschnitten)

Array-basierter Vektor `avec`: Klassensignatur

```
class avec {
  // Private (internal) state:
  1int* elements; // Pointer to first element
  2unsigned int count; // Number of elements

public: // Public interface:
  3avec(unsigned int size); // Constructor
  4unsigned int size() const; // Size of vector
  5int& operator[](int i); // Access an element
  6void print(std::ostream& sink) const; // Output elems.
}
```

Konstruktor `avec::avec()`

```
avec::avec(unsigned int size)
        : 1count(size) {

  2elements = new int[size];
}
```

Grösse speichern

Speicher allozieren

Nebenbemerkung: Vektor wird nicht mit einem Standardwert initialisiert

Exkurs: Zugriff auf Membervariablen

```
avec::avec(unsigned int size): count(size) {
  this->elements = new int[size];
}
```

- `elements` ist eine Membervariable unserer `avec`-Instanz
- Diese Instanz ist mittels des *Zeigers* `this` zugreifbar
- `elements` ist eine Kurzschreibweise von `(*this).elements`
- Dereferenzieren eines Zeigers (`*this`) gefolgt von einem Memberzugriff (`.elements`) ist eine so häufig genutzte Operation, dass sie verkürzt geschrieben werden kann als `this->elements`
- Eselsbrücke: „Folge dem Zeiger zur Membervariablen“

Funktion `avec::size()`

```
int avec::size() 1const {
  2return this->count;
}
```

Verändert den Vektor nicht

Grösse zurückgeben

Anwendungsbeispiel:

```
avec v = avec(7);
assert(v.size() == 7); // ok
```

Funktion `avec::operator[]`

```
int& avec::operator[](int i) {
  1return this->elements[i];
}
```

Elementzugriff mit Indexüberprüfung:

```
int& avec::at(int i) const {
  assert(0 <= i && i < this->count);

  return this->elements[i];
}
```

Funktion `avec::operator[]`

```
int& avec::operator[](int i) {
  return this->elements[i];
}
```

Anwendungsbeispiel:

```
avec v = avec(7);
std::cout << v[6]; // Outputs a "random" value
v[6] = 0;
std::cout << v[6]; // Outputs 0
```

Funktion `avec::operator[]` braucht's doppelt

```
int& avec::operator[](int i) { return elements[i]; }
const int& avec::operator[](int i) const { return
   elements[i]; }
```

- Die erste Memberfunktion ist *nicht const* und gibt eine *nicht-const*-Referenz zurück

```
avec v = ...; // A non-const vector
std::cout << v.get[0]; // Reading elements is
   allowed
v.get[0] = 123; // Modifying elements is allowed
```

- Sie wird auf nicht-const-Vektoren aufgerufen

Funktion `avec::operator[]` braucht's doppelt

```
int& avec::operator[](int i) { return elements[i]; }
const int& avec::operator[](int i) const { return
   elements[i]; }
```

- Die zweite Memberfunktion *ist const* und gibt eine *const*-Referenz zurück

```
const avec v = ...; // A const vector
std::cout << v.get[0]; // Reading elements is
   allowed
v.get[0] = 123; // Compiler error: modifications
   are not allowed
```

- Sie wird auf const-Vektoren aufgerufen

Funktion `avec::print()`

Elemente mittels sequenziellem Zugriff ausgeben:

```
void avec::print(std::ostream& sink) const {
  for (1int* p = this->elements;
       2p != this->elements + this->count;
       3++p)
  {
    4sink << *p << ' ';
  }
}
```

Zeiger auf erstes Element

Abbruch falls hinter letztem Element

Zeiger elementweise voranschieben

Aktuelles Element ausgeben

Funktion `avec::print()`

Abschluss: Ausgabeoperator überladen:

```
_____________________ operator<<(_____________________ sink,
                                 _____________________ vec) {
  vec.print(sink);
  return ____________________;
}
```

```
std::ostream& operator<<(std::ostream& sink,
                         const avec& vec) {
  vec.print(sink);
  return sink;
}
```

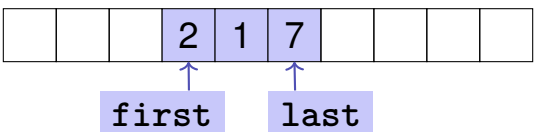
Weitere Funktionen

```
class avec {
  ...
  void push_front(int e)       // Prepend e to vector
  void push_back(int e)        // Append e to vector
  void remove(unsigned int i) // Cut out ith element
  ...
}
```

Gemeinsamkeit: Diese Operationen müssen die *Grösse* des Vektors verändern

Arrays vergrössern/verkleiner

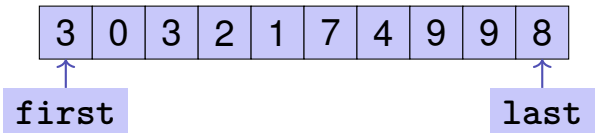
Ein allozierter Speicherblock (z.B. `new int[3]`) kann nicht nachträglich vergrössert/verkleinert werden



Möglichkeit:

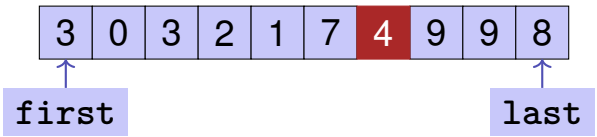
- Mehr Speicher als initial nötig allozieren
- Befüllen aus der Mitte heraus, mittels Zeigern auf erstes und letztes Element

Arrays vergrössern/verkleiner

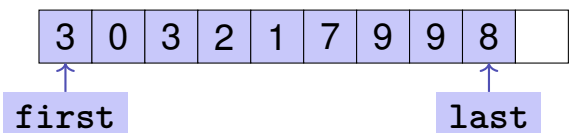


- Aber irgendwann sind alle Plätze belegt
- Dann nötig: Grösseren Speicherblock allozieren und Daten umkopieren

Arrays vergrössern/verkleiner



Elemente löschen erfordert verschieben (via kopieren) aller vorhergehenden oder nachfolgenden Elemente



Ähnlich: Einfügen an beliebiger Position

19. Dynamische Datenstrukturen II

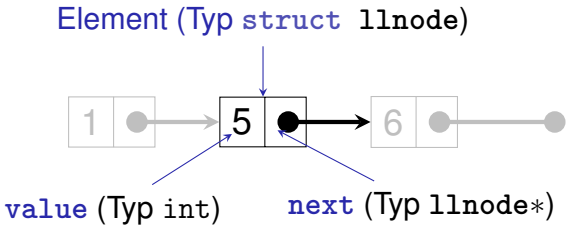
Verkettete Listen, Vektoren als verkettete Listen

Anderes Speicherlayout: Verkettete Liste

- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element zeigt auf seinen Nachfolger
- Einfügen und Löschen *beliebiger* Elemente ist einfach

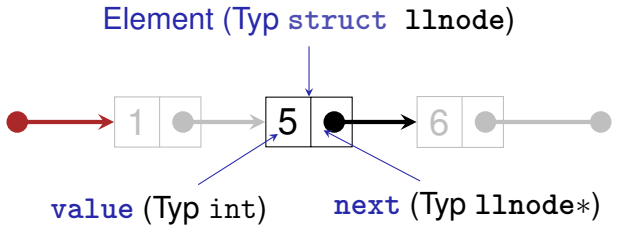
$\Rightarrow$ Unser Vektor kann als verkettete Liste realisiert werden

Verkettete Liste: Zoom




```
struct llnode {
  int value;
  llnode* next;

  llnode(int v, llnode* n): value(v), next(n) {} //
     Constructor
};
```

Vektor = Zeiger aufs erste Element




```
class llvec {
  llnode* head;
public:
  // Public interface identical to avec's
  llvec(unsigned int size);
  unsigned int size() const;
  ...
```

Funktion `llvec::print()`

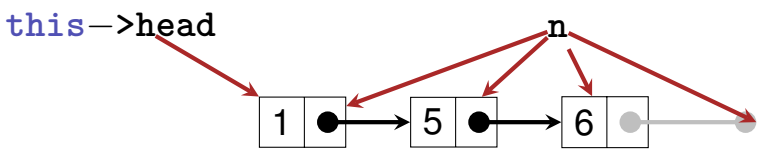
```
struct llnode {
  int value;
  llnode* next;
  ...
};
```

```
void llvec::print(std::ostream& sink) const {
  for (1llnode* n = this->head;
       2n != nullptr;
       3n = n->next)
  {
    4sink << n->value << ' ';
  }
}
```

1. Zeiger auf erstes Element
2. Abbruch falls Ende erreicht
3. Zeiger elementweise voranschieben
4. Aktuelles Element ausgeben

Funktion `llvec::print()`

```
void llvec::print(std::ostream& sink) const {
  for (llnode* n = this->head;
       258n != nullptr;
       n = n->next)
  {
    sink << n->value << ' ';  // 1 5 6
  }
}
```



Funktion `llvec::operator[]`

Zugriff auf i-tes Element ähnlich implementiert wie `print()`:

```
int& llvec::operator[](unsigned int i) {
  1llnode* n = this->head;

  2for (; 0 < i; --i)
    2n = n->next;

  3return n->value;
}
```

Zeiger auf erstes Element

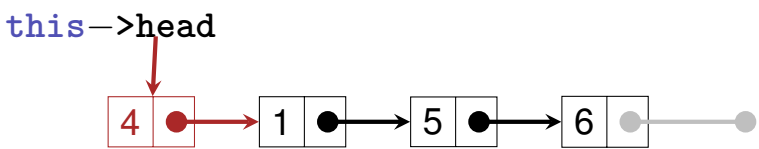
Bis zum `i`-ten voranschreiten

`i`-tes Element zurückgeben

Funktion `llvec::push_front()`

Vorteil `llvec`: Elemente am Anfang anfügen ist sehr einfach:

```
void llvec::push_front(int e) {
  4this->head =
    2new llnode{3e, this->head};
}
```



Achtung: Wäre der neue `llnode` nicht *dynamisch* alloziert, dann würde er am Ende von `push_front` sofort wieder gelöscht (= Speicher dealloziert)

Funktion `llvec::llvec()`

Konstruktor kann mittels `push_front()` implementiert werden:

```
llvec::llvec(unsigned int size) {
  1this->head = nullptr;

  2for (; 0 < size; --size)
    2this->push_front(0);
}
```

`head` zeigt zunächst ins Nichts

`size` mal 0 vorne anfügen

Anwendungsbeispiel:

```
llvec v = llvec(3);
std::cout << v; // 0 0 0
```

Funktion `llvec::push_back()`

Einfach, aber ineffizient: Verkettete Liste bis ans Ende traversieren und neues Element anhängen

```
void llvec::push_back(int e) {
  1llnode* n = this->head;

  2for (; n->next != nullptr; n = n->next);

  3n->next =
    3new llnode{e, nullptr};
}
```

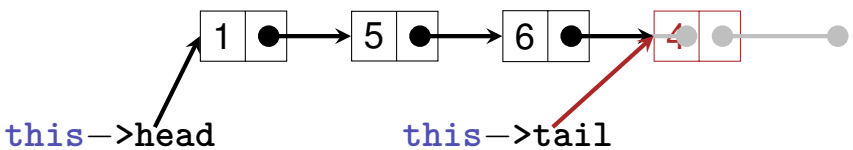
Beim ersten Element beginnen ...

... und bis zum letzten Element gehen

Neues Element an bisher letztes anhängen

Funktion `llvec::push_back()`

- Effizienter, aber auch etwas komplexer:
  1. Zweiter Zeiger, der auf das letzte Element zeigt: `this->tail`
  2. Mittels dieses Zeigers kann direkt am Ende angehängt werden



- Aber: Verschiedene Grenzfälle, z.B. Vektor noch leer, müssen beachtet werden

Funktion `llvec::size()`

Einfach, aber ineffizient: Grösse durch abzählen *berechnen*

```
unsigned int llvec::size() const {
  1unsigned int c = 0;

  2for (llnode* n = this->head;
       2n != nullptr;
       2n = n->next)
    2++c;

  3return c;
}
```

Zähler initial 0

Länge der Kette abzählen

Zähler zurückgeben

Funktion `llvec::size()`

Effizienter, aber etwas komplexer: Grösse als Membervariable *nachhalten*

1. Membervariable `unsigned int count` zur Klasse `llvec` hinzufügen
2. `this->count` muss nun bei *jeder* Operation, die die Grösse des Vektors verändert (z.B. `push_front`), aktualisiert werden

Effizienz: Arrays vs. Verkettete Listen

- Speicher: Unser `avec` belegt ungefähr n int s (Vektorgrösse n), unser `llvec` ungefähr $3n$ int s (ein Zeiger belegt i.d.R. 8 Byte)

- Laufzeit (mit `avec` = `std::vector`, `llvec` = `std::list`):

20. Container, Iteratoren und Algorithmen

Container, Mengen, Iteratoren, const-Iteratoren, Algorithmen, Templates

Vektoren sind Container

- Abstrakt gesehen ist ein Vektor
  1. Eine Ansammlung von Elementen
  2. Plus Operationen auf dieser Ansammlung
- In C++ heissen `vector<`T`>` und ähnliche „Ansammlungs“-Datenstrukturen *Container*
- In manchen Sprachen, z.B. Java, *Collections* genannt

Container-Eigenschaften

- Jeder Container hat bestimmte *charakteristische Eigenschaften*
- Ein Array-basierter Vektor z.B. die folgenden:
  - Effizienter, index-basierter Zugriff (`v[i]`)
  - Effiziente Speichernutzung: Nur die Elemente selbst belegen Platz (plus Elementezähler)
  - Einfügen/Entfernen an beliebigem Index ist potenziell ineffizient
  - Suchen eines bestimmten Elements ist potenziell ineffizient
  - Kann Elemente mehrfach enthalten
  - Elemente sind in Einfügereihenfolge enthalten (geordnet aber unsortiert)

Container in C++

- Fast jede Anwendung erfordert die Verwaltung und Manipulation von beliebig vielen Datensätzen
- Aber mit unterschiedlichen Anforderungen (z.B. Elemente nur hinten anhängen, fast nie entfernen, oft suchen, . . .)
- Deswegen enthält die Standardbibliothek von C++diverse Container mit unterschiedlichen Eigenschaften, siehe `https://en.cppreference.com/w/cpp/container`
- Viele weitere sind über Bibliotheken Dritter verfügbar, z.B. `https://www.boost.org/doc/libs/1_68_0/doc/html/container.html`, `https://github.com/abseil/abseil-cpp`

Beispiel-Container: `std::unordered_set<`T`>`

- Eine *mathematische Menge* ist eine ungeordnete, duplikatfreie Zusammenfassung von Elementen:

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- In C++: **`std::unordered_set<`**T**`>`**
- Eigenschaften:
  - Kann kein Element doppelt enthalten
  - Elemente haben keine bestimmte Reihenfolge
  - Kein indexbasierter Zugriff (`s[i]` nicht definiert)
  - Effiziente „Element enthalten?“-Prüfung
  - Effizientes Einfügen und Löschen von Elementen
- Randbemerkung: Implementiert als Hash-Tabelle

Anwendungsbeispiel `std::unordered_set<`T`>`

Problem:

- Gegeben eine Sequenz an Paaren *(Name, Prozente)* von Code-Expert-Submissions . . .

```
// Input: file submissions.txt
Friedrich 90
Schwerhoff 10
Lehner 20
Schwerhoff 11
```

- . . . bestimme die Abgebenden, die mindestens 50% erzielt haben

```
// Output
Friedrich
```

Anwendungsbeispiel `std::unordered_set<`T`>`

```
1std::ifstream in("submissions.txt");
2std::unordered_set<std::string> names;

3std::string name;
3unsigned int score;

while (4in >> name >> score) {
  5if (50 <= score)
    5names.insert(name);
}

6std::cout << "Unique submitters: "
          6<< names << '\n';
```

Öffne `submissions.txt`

Namen-Menge, initial leer

Paar *(Name, Punkte)*

Nächstes Paar einlesen

Namen merken falls Punkte ausreichen

Gemerkte Namen ausgeben

Beispiel-Container: `std::set<`T`>`

- Fast gleich wie `std::unordered_set<`T`>`, aber die Elemente sind *geordnet*

$$\{1, 2, 1\} = \{1, 2\} \neq \{2, 1\}$$

- Elemente suchen, einfügen und löschen weiterhin effizient (besser als bei `std::vector<`T`>`), aber weniger effizient als bei `std::unordered_set<`T`>`
- Denn das Beibehalten der Ordnung zieht etwas Aufwand nach sich
- Randbemerkung: Implementiert als Rot-Schwarz-Baum

Anwendungsbeispiel `std::set<`T`>`

```
std::ifstream in("submissions.txt");
1std::set<std::string> names;

std::string name;
unsigned int score;

while (in >> name >> score) {
  if (50 <= score)
    names.insert(name);
}

2std::cout << "Unique submitters: "
          2<< names << '\n';
```

`set` statt `unsorted_set` ...

... und die Ausgabe erfolgt alphabetisch sortiert

Container Ausgeben

- Bereits gesehen: `avec::print()` und `llvec::print()`
- Wie sieht's mit der Ausgabe von `set`, `unordered_set`, ... aus?
- Gemeinsamkeit: Über Container-Elemente iterieren und diese ausgeben

Ähnliche Funktionen

- Viele weitere nützliche Funktionen können mittels Container-Iteration implementiert werden:
- `contains(c, e)`: wahr gdw. Container `c` Element `e` enthält
- `min/max(c)`: Gibt das grösste/kleinste Element zurück
- `sort(c)`: Sortiert die Elemente von `c`
- `replace(c, e1, e2)`: Ersetzt alle `e1` in `c` mit `e2`
- `sample(c, n)`: Wählt zufällig `n` Elemente aus `c` aus
- ...

Zur Erinnerung: Iterieren mit Zeigern

- Iteration über ein *Array*:
  - Auf Startelement zeigen: `p = this->arr`
  - Auf aktuelles Element zugreifen: `*p`
  - Überprüfen, ob Ende erreicht: `p == p + size`
  - Zeiger vorrücken: `p = p + 1`

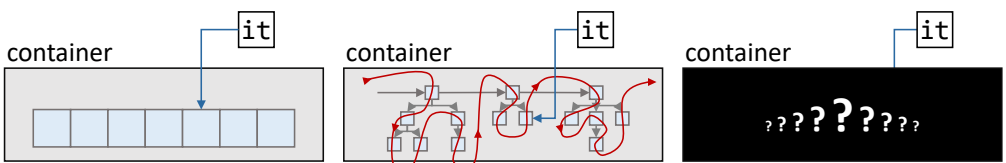
- Iteration über eine *verkettete Liste*:
  - Auf Startelement zeigen: `p = this->head`
  - Auf aktuelles Element zugreifen: `p->value`
  - Überprüfen, ob Ende erreicht: `p == nullptr`
  - Zeiger vorrücken: `p = p->next`

Iteratoren

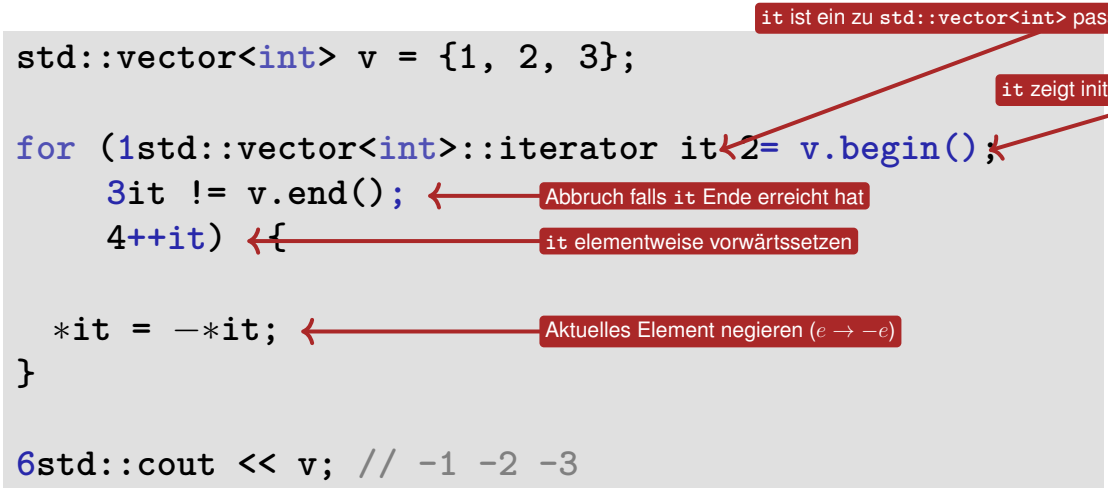
- Iteration erfordert nur die vier eben gesehenen Operationen
- Aber deren Implementierung hängt vom Container ab
- $\Rightarrow$ Jeder C++-Container implementiert seinen eigenen *Iterator*
- Gegeben ein Container `c`:
  - `it = c.begin()`: Iterator aufs erste Element
  - `it = c.end()`: Iterator *hinters* letzte Element
  - `*it`: Zugriff aufs aktuelle Element
  - `++it`: Iterator um ein Element verschieben
- Iteratoren sind quasi gepimpte Zeiger

Iteratoren

- Iteratoren ermöglichen Zugriff auf verschiedene Container auf *uniforme* Weise: `*it`, `++it`, etc.
- Nutzer bleiben unabhängig von der Container-Implementierung
- Iterator weiss, wie man die Elemente „seines“ Containers abläuft
- Nutzer müssen und sollen interne Details nicht kennen
- $\Rightarrow$ Containerimplementierung kann geändert werden, ohne das Nutzer Code ändern müssen



Beispiel: Iteration über `std::vector`



Beispiel: Iteration über `std::vector`

Zur Erinnerung: Type-Aliasse können genutzt werden um oft genutzte Typnamen abzukürzen

```
1using ivit = std::vector<int>::iterator; // int-
   vector iterator

for (1ivit it = v.begin();
     ...
```

Negieren als Funktion

```
void neg(std::vector<int>& v) {
  for (std::vector<int>::iterator it = v.begin();
       it != v.end();
       ++it) {

    *it = -*it;
  }
}

// in main():
std::vector<int> v = {1, 2, 3};
neg(v); // v = {-1, -2, -3}
```

Negieren als Funktion

Besser: Innerhalb eines bestimmten *Bereichs (Intervalls)* negieren

```
void neg(1std::vector<int>::iterator begin;
         1std::vector<int>::iterator end) {

  for (std::vector<int>::iterator it = 1begin;
       it != 1end;
       ++it) {

    *it = -*it;
  }
}
```

Elemente im Interv
[begin, end) negie

Negieren als Funktion

Besser: Innerhalb eines bestimmten *Bereichs (Intervalls)* negieren

```
void neg(std::vector<int>::iterator start;
         std::vector<int>::iterator end);

// in main():
std::vector<int> v = {1, 2, 3};
1neg(v.begin(), v.begin() + (v.size() / 2));
```

← Erste Hälfte negieren

Algorithmen-Bibliothek in C++

- Die C++-Standardbibliothek enthält viele nützliche Algorithmen (Funktionen), die auf durch Iteratoren bestimmten Intervallen [*Anfang, Ende*) arbeiten
- Zum Beispiel **`find`**, **`fill`** and **`sort`**
- Siehe auch `https://en.cppreference.com/w/cpp/algorithm`

Ein Iterator für `llvec`

Wir brauchen:

1. Einen `llvec`-spezifischen Iterator mit mindestens folgender Funktionalität:
   - Zugriff aktuelles Element: `operator*`
   - Iterator vorwärtssetzen: `operator++`
   - Ende-Erreicht-Prüfung: `operator!=` (oder `operator==`)
2. Memberfunktionen `begin()` und `end()` für `llvec` um einen Iterator auf den Anfang bzw. hinter das Ende zu erhalten

Iterator `avec::iterator` (Schritt 1/2)

```
1class llvec {
  ...
public:
  1class iterator {
  1  ...
  1};

  ...
}
```

- Der Iterator gehört zu unserem Vektor, daher ist `iterator` eine öffentliche *innere Klasse* von `llvec`
- Instanzen unseres Iterators sind vom Typ `llvec::iterator`

Iterator `llvec::iterator` (Schritt 1/2)

```
class iterator {
  1llnode* node;

public:
  2iterator(llnode* n);
  3iterator& operator++();
  4int& operator*() const;
  5bool operator!=(const iterator& other) const;
};
```

Zeiger auf aktuelles Vektor-Element

Erzeuge Iterator auf bestimmtes Element

Iterator ein Element vorwärtssetzen

Zugriff auf aktuelles Element

Vergleich mit anderem Iterator

Iterator `llvec::iterator` (Schritt 1/2)

```
// Constructor
llvec::iterator::iterator(llnode* n): 2node(n) {}

// Pre-increment
llvec::iterator& llvec::iterator::operator++() {
  assert(this->node != nullptr);

  4this->node = this->node->next;

  5return *this;
}
```

Iterator initial auf `n` zeigen lassen

Iterator ein Element vorwärtssetzen

Referenz auf verschobenen Iterator zurückgeben

Iterator `llvec::iterator` (Schritt 1/2)

```
// Element access
int& llvec::iterator::operator*() const {
  2return this->node->value;
}


// Comparison
bool llvec::iterator::operator!=(const llvec::
   iterator& other) const {
  4return this->node != other.node;
}
```

Zugriff auf aktuelles Element

`this` Iterator verschieden von `other` falls sie auf unterschiedliche Elemente zeigen

Ein Iterator für `llvec` (Wiederholung)

Wir brauchen:

1. Einen `llvec`-spezifischen Iterator mit mindestens folgender Funktionalität:
   - Zugriff aktuelles Element: `operator*`
   - Iterator vorwärtssetzen: `operator++`
   - Ende-Erreicht-Prüfung: `operator!=` (oder `operator==`)

2. Memberfunktionen **`begin()`** und **`end()`** für `llvec` um einen Iterator auf den Anfang bzw. hinter das Ende zu erhalten

Iterator `avec::iterator` (Schritt 2/2)

```
1class llvec {
  ...
public:
  class iterator {...};

  1iterator begin();
  1iterator end();

  ...
}
```

`llvec` braucht Memberfunktionen um Iteratoren *auf den Anfang* bzw. *hinter das Ende* des Vektors herausgeben zu können

Iterator `llvec::iterator` (Schritt 2/2)

```
llvec::iterator llvec::begin() {
  1return llvec::iterator(this->head);
}

llvec::iterator llvec::end() {
  2return llvec::iterator(nullptr);
}
```

Iterator auf erstes Vektorelement

Iterator hinter letztes Vektorelement

Const-Iteratoren

- Neben `iterator` sollte jeder Container auch einen *Const-Iterator* `const_iterator` bereitstellen
- Const-Iteratoren gestatten nur Lesezugriff auf den darunterliegenden Container
- Zum Beispiel für `llvec`:

```
llvec::1-const_iterator llvec::1-cbegin() const;
llvec::1-const_iterator llvec::1-cend() const;

1-const int& llvec::const_iterator::operator*()
   const;
...
```

- Daher nicht möglich (Compilerfehler): `*(v.cbegin()) = 0`

Const-Iteratoren

Const-Iterator *kann* verwendet werden um nur Lesen zu erlauben:

```
llvec v = ...;
for (llvec::1const_iterator it = v.1cbegin(); ...)
  std::cout << *it;
```

Hier könnte auch der nicht-const `iterator` verwendet werden

Const-Iteratoren

Const-Iterator *muss* verwendet werden falls Vektor selbst const ist:

```
1const llvec v = ...;
for (llvec::1const_iterator it = 1v.cbegin(); ...)
  std::cout << *it;
```

Hier kann nicht der `iterator` verwendet werden (Compilerfehler)

Exkurs: Templates

- Ziel: Ein *generischer* Ausgabe-Operator << für *iterierbare Container*: `llvec`, `avec`, `std::vector`, `std::set`, ...
- D.h. `std::cout << c << 'n'` soll für jeden solchen Container `c` funktionieren

Exkurs: Templates

Templates ermöglichen *Typ-generische* Funktionen und Klassen:

- Templates ermöglichen die Nutzung von *Typen als Argumenten*

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container);
```

Die spitzen Klammern kennen wir schon von `std::vector<int>`. Vektoren sind auch als Templates realisiert.

Intuition: Operator funktioniert für jeden Ausgabestrom `sink` vom Typ `S` und jeden Container `container` vom Typ `C`

Exkurs: Templates

Templates ermöglichen *Typ-generische* Funktionen und Klassen:

- Templates ermöglichen die Nutzung von *Typen als Argumenten*

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container);
```

- Der Compiler *inferiert* passende Typen aus den Aufrufargumenten

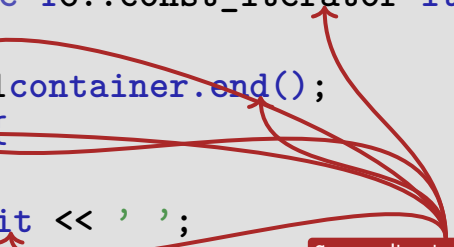
```
std::set<int> s = ...;
std::cout << s << '\n';
```

S = std::ostream, C = std::set<int>

Exkurs: Templates

Implementierung von << *schränkt* `S` und `C` *ein* (Compilerfehler falls nicht erfüllt):

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container) {
  for (typename 1C::const_iterator it = 1container.
     begin();
       it 2!= 1container.end();
       2++it) {

    sink << 2*it << ' ';
  }
```



Exkurs: Templates

Implementierung von << *schränkt* `S` und `C` *ein* (Compilerfehler falls nicht erfüllt):

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container) {
  for (typename C::const_iterator it = container.
     begin();
       it != container.end();
       ++it) {

    1sink << *it << ' ';
  }
```

`S` muss Ausgabe von Elementen (`*it`) und Zeichen (`' '`) unterstützen

21. Dynamische Datentypen und Speicherverwaltung

Problem

Letzte Woche: Dynamischer Datentyp

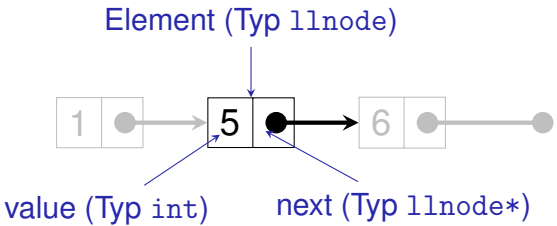
Haben im Vektor dynamischen Speicher angelegt, aber nicht wieder freigegeben. Insbesondere: keine Funktionen zum Entfernen von Elementen aus `llvec`.

Heute: Korrektes Speichermanagement!

Ziel: Stapelklasse mit Speichermanagement

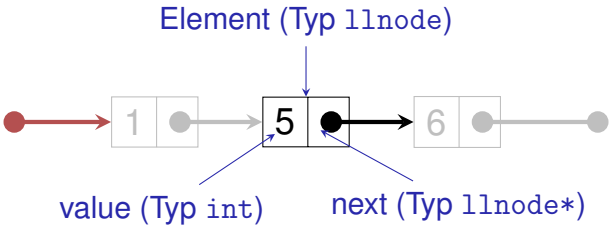
```
class stack{
public:
  // post: Element auf den Stapel legen
  void push(int value);
  // pre: Stack nicht leer
  // post: Entfernt oberstes Element vom Stapel
  void pop();
  // pre: Stack nicht leer
  // post: Gibt Wert des obersten Elementes zurück
  int top() const;
  // post: gibt zurück, ob Stack leer ist
  bool empty() const;
  // post: gibt den Stapel aus
  void print(std::ostream& out) const;
...
```

Erinnerung: Verkettete Liste



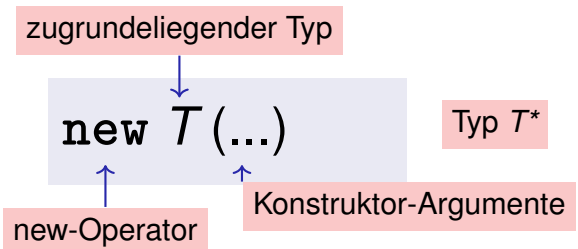

```
struct llnode {
  int value;
  llnode* next;
  // constructor
  llnode (int v, llnode* n) : value (v), next (n) {}
};
```

Stapel = Zeiger aufs oberste Element




```
class stack {
public:
  void push (int value);
  ...
private:
  llnode* topn;
};
```

Erinnerung: der `new`-Ausdruck

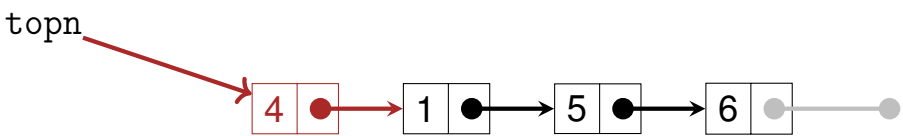


- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.
- **Wert:** Adresse des neuen Objekts

Der `new`-Ausdruck: `push(4)`

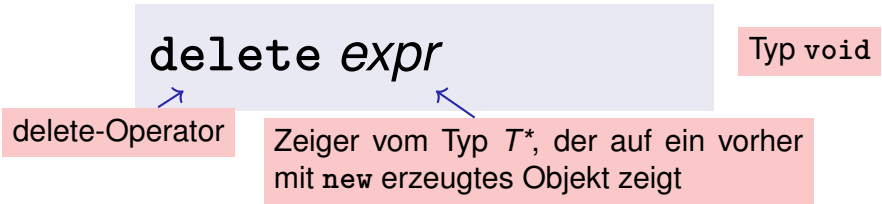
- **Effekt:** Neues Objekt vom Typ *T* wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.
- **Wert:** Adresse des neuen Objekts

```
void stack::push(int value){
  topn = new llnode (value, topn);
}
```



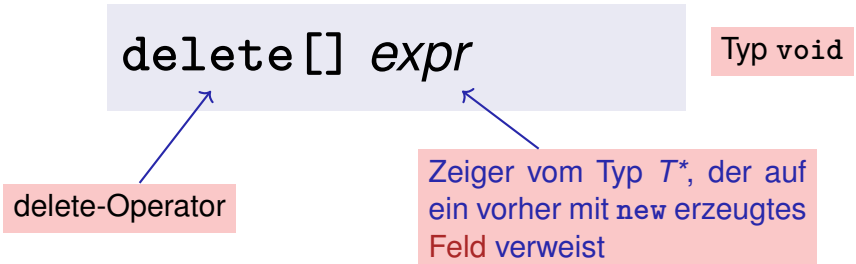
Der `delete`-Ausdruck

Objekte, die mit **`new`** erzeugt worden sind, haben *dynamische Speicherdauer:* sie “leben”, bis sie explizit *gelöscht* werden.



- **Effekt:** Objekt wird *dekonstruiert* (Erklärung folgt)
  ... und *Speicher wird freigegeben*.

Der `delete`-Ausdruck für Felder



- **Effekt:** Feld wird gelöscht, Speicher wird wieder freigegeben

Wer geboren wird, muss sterben...

Richtlinie “Dynamischer Speicher”

Zu jedem `new` gibt es ein passendes `delete`!

Nichtbeachtung führt zu *Speicherlecks*:

- “Alte” Objekte, die den Speicher blockieren...
- ...bis er irgendwann voll ist (heap overflow)

Aufpassen mit `new` und `delete`!

```
rational* t = new rational;
rational* s = t;
delete s;
int nominator = (*t).denominator();
```

Speicher für t wird angelegt

Auch andere Zeiger können auf das Objekt zeigen..

... und zur Freigabe verwendet werden.

Fehler: Speicher freigegeben!

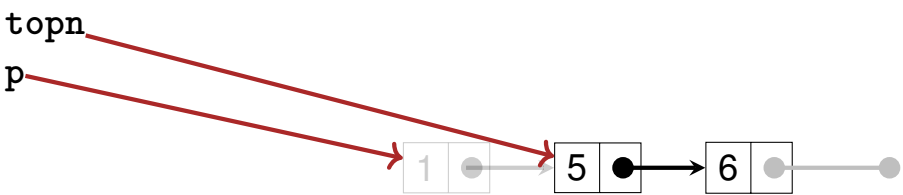
Dereferenzieren eines „dangling pointers”

- Zeiger auf freigegebene Objekte: hängende Zeiger (*dangling pointers*)
- Mehrfache Freigabe eines Objektes mit `delete` ist ein ähnlicher schwerer Fehler.

Weiter mit dem Stapel: pop()

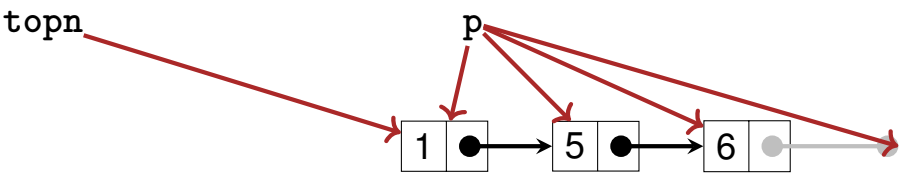
```
void stack::pop(){
    assert (!empty());
    llnode* p = topn;
    topn = topn->next;
    delete p;
}
```

Erinnerung: Abkürzung für `(*topn).next`



Stapel ausgeben: print()

```
void stack::print (std::ostream& out) const {
  for(const llnode* p = topn; p != nullptr; p = p->next)
    out << p->value << " "; // 1 5 6
}
```



Stapel ausgeben: operator<<

```
class stack {
public:
   void push (int value);
   void pop();
   void print (std::ostream& o) const;
   ...
private:
   llnode* topn;
};

// POST: s is written to o
std::ostream& operator<< (std::ostream& o, const stack& s){
   s.print (o);
   return o;
}
```

empty(),top()

```
bool stack::empty() const {
  return top == nullptr;
}

int stack::top() const {
  assert(!empty());
  return topn->value;
}
```

Leerer Stapel

```
class stack{
public:
  stack() : topn (nullptr) {} // default constructor

  void push(int value);
  void pop();
  void print(std::ostream& out) const;
  int top() const;
  bool empty() const;
private:
  llnode* topn;
}
```

Zombie-Elemente

```
{
  stack s1; // lokale Variable
  s1.push (1);
  s1.push (3);
  s1.push (2);
  std::cout << s1 << "\n"; // 2 3 1
}
// s1 ist gestorben (nicht mehr zugreifbar)
```

- . . . aber die drei *Elemente* des Stapels `s1` leben weiter (Speicherleck)!
- Sie sollten zusammen mit `s1` aufgeräumt werden!

Der Destruktor

- Der Destruktor einer Klasse T ist die eindeutige Memberfunktion mit Deklaration

$$\sim T\,(\,);$$

- Wird automatisch aufgerufen, wenn die Speicherdauer eines Klassenobjekts vom Typ `T` endet – z.B. bei Aufruf von `delete` auf einem Objekt vom Typ `T*` oder wenn der Gültigkeitsbereich eines Objektes vom Typ `T` endet.
- Falls kein Destruktor deklariert ist, so wird er automatisch erzeugt und ruft die Destruktoren für die Membervariablen auf (Zeiger `topn`, kein Effekt – Grund für Zombie-Elemente)

Mit dem Destruktor klappt's!

```
// POST: the dynamic memory of *this is deleted
stack::~stack(){
  while (topn != nullptr){
    llnode* t = topn;
    topn = t->next;
    delete t;
  }
}
```

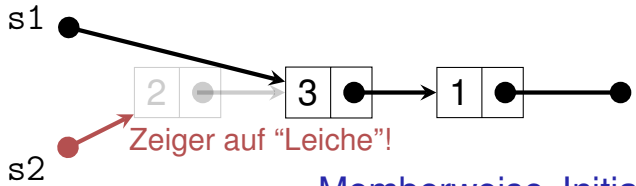
- löscht automatisch alle Stapelemente, wenn der Stapel ungültig wird
- Unsere Stapel-Klasse scheint jetzt die Richtlinie "Dynamischer Speicher" zu befolgen (?)

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?



Memberweise Initialisierung: kopiert nur den `topn`-Zeiger

```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();  // Oops, Programmabsturz!
```

Das eigentliche Problem

Schon das geht schief:

```
{
  stack s1;
  s1.push(1);
  stack s2 = s1;
}
```

Beim Verlassen des Gültigkeitsbereiches werden beide Stacks aufgeräumt (dekonstruiert). Aber beide Stacks versuchen dieselben Daten zu löschen, denn sie haben *Zugriff auf denselben Zeiger*.

Lösungsmöglichkeiten

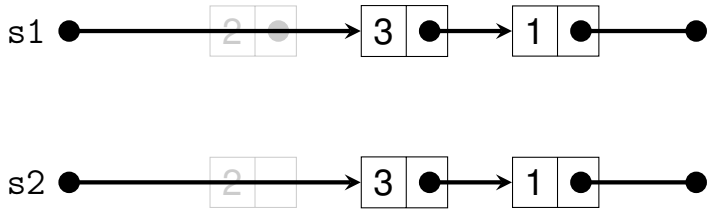
Smart-Pointers (werden hier nicht weiter vertieft):

- Zähle die Anzahl Zeiger, die auf ein Objekt verweisen. Lösche nur wenn diese Anzahl auf 0 zurückfällt: `std::shared_pointer`
- Verhindere, dass mehrere Zeiger auf ein Objekt zeigen können: `std::unique_pointer`.

oder:

- Wir erstellen eine echte Kopie aller Daten – wie folgt.

Wir erstellen eine echte Kopie!




```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();  // ok
```

Der Copy-Konstruktor

- Der Copy-Konstruktor einer Klasse *T* ist der eindeutige Konstruktor mit Deklaration

  T (`const` *T*`&` x);

- wird automatisch aufgerufen, wenn Werte vom Typ *T* mit Werten vom Typ `T` *initialisiert* werden

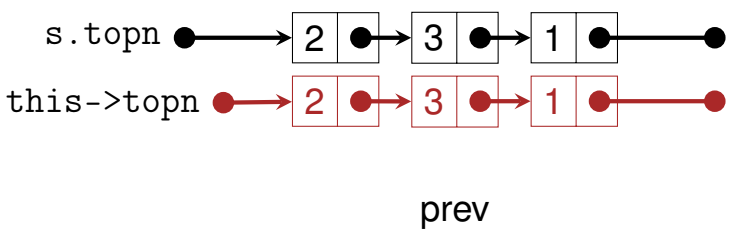
  T `x = t;` (t vom Typ T)

  T `x (t);`

- Falls kein Copy-Konstruktor deklariert ist, so wird er automatisch erzeugt (und initialisiert memberweise – Grund für obiges Problem)

Mit dem Copy-Konstruktor klappt's!

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
  if (s.topn == nullptr) return;
  topn = new llnode(s.topn->value, nullptr);
  llnode* prev = topn;
  for(llnode* n = s.topn->next; n != nullptr; n = n->next){
    llnode* copy = new llnode(n->value, nullptr);
    prev->next = copy;
    prev = copy;
  }
}
```



NB: rekursives Kopieren

```
llnode* copy (node* that){
  if (that == nullptr) return nullptr;
  return new llnode(that->value, copy(that->next));
}
```

Elegant, oder? Warum haben wir das nicht gleich so gemacht?

Grund: verkettete Listen können sehr lang werden. Dann könnte `copy` zum Stapelüberlauf[8] führen. Aufrufstapel ist nämlich meist kleiner als Heapspeicher.

[8] nicht von dem Stapel, den wir gerade implementieren, sondern vom Aufrufstapel der Rekursion

Initialisierung $\neq$ Zuweisung!

```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n"; // 2 3 1

stack s2;
s2 = s1; // Zuweisung

s1.pop ();
std::cout << s1 << "\n"; // 3 1
s2.pop (); // Oops, Programmabsturz!
```

Der Zuweisungsoperator

- Überladung von `operator=` als Memberfunktion
- Wie Copy-Konstruktor ohne Initialisierer, aber zusätzlich
  - Freigabe des Speichers für den „alten” Wert
  - Prüfen auf Selbstzuweisungen (`s1=s1`), die keinen Effekt haben sollen
- Falls kein Zuweisungsoperator deklariert ist, so wird er automatisch erzeugt (und weist memberweise zu – Grund für obiges Problem)

Mit dem Zuweisungsoperator klappt's!

```
// POST: *this (left operand) becomes a
//             copy of s (right operand)
stack& stack::operator= (const stack& s){
  if (topn != s.topn){ // keine Selbstzuweisung
    stack copy = s; // Kopierkonstruktor
    std::swap(topn, copy.topn); // copy hat nun den Müll!
  } // copy wird aufgeräumt -> Dekonstruktion
  return *this; // Rueckgabe als L-Wert (Konvention)
}
```

Cooooler Trick! ☺

Fertig

```
class stack{
public:
  stack(); // constructor
  ~stack(); // destructor
  stack(const stack& s); // copy constructor
  stack& operator=(const stack& s); // assignment operator

  void push(int value);
  void pop();
  int top() const;
  bool empty() const;
  void print(std::ostream& out) const;
private:
  llnode* topn;
}
```

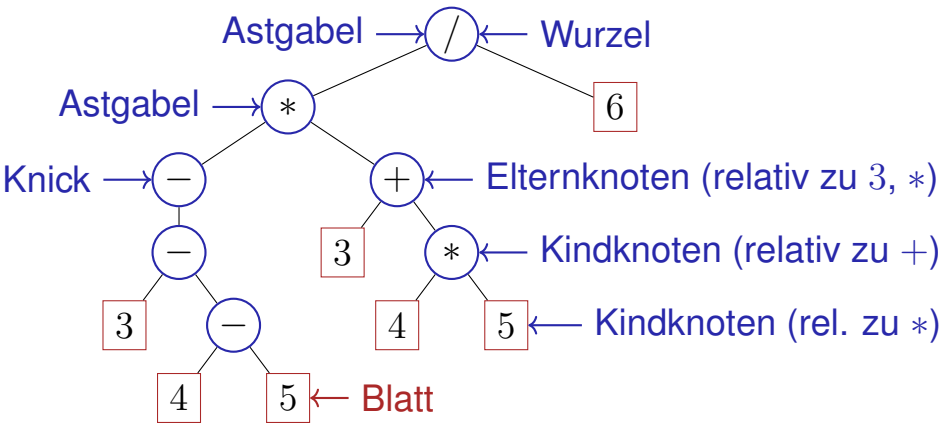
Dynamischer Datentyp

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Stapel)
- Mindestfunktionalität:
  - Konstruktoren
  - Destruktor
  - Copy-Konstruktor
  - Zuweisungsoperator

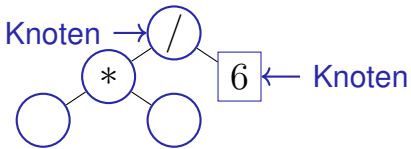
Dreierregel: definiert eine Klasse eines davon, so muss sie auch die anderen zwei definieren!

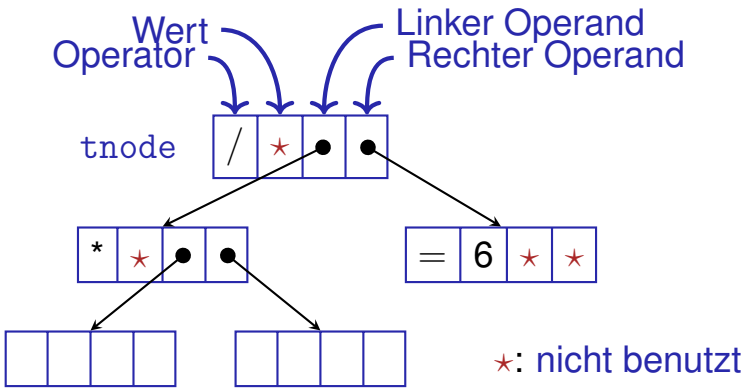
(Ausdrucks-)Bäume

```
-(3-(4-5))*(3+4*5)/6
```

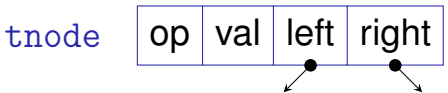


Astgabeln + Blätter + Knicke = Knoten



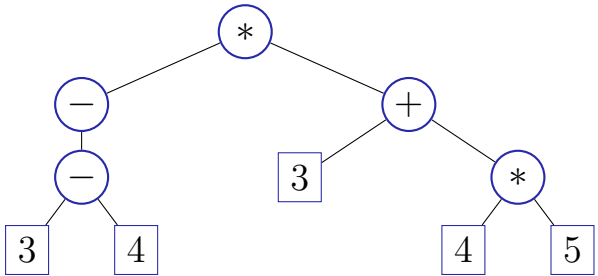


Knoten (`struct tnode`)




```
struct tnode {
  char op; // leaf node: op is '='
           // internal node: op is '+', '-', '*' or '/'
  double val;
  tnode* left; // == nullptr for unary minus
  tnode* right;

  tnode(char o, double v, tnode* l, tnode* r)
    : op(o), val(v), left(l), right(r) {}
};
```

Grösse = Knoten in Teilbäumen zählen



- Grösse eines Blattes: 1
- Grösse anderer Knoten: 1 + Gesamtgrösse aller Kindknoten
- Z.B. Grösse des „+“-Knoten ist 5

Knoten in Teilbäumen zählen

```
// POST: returns the size (number of nodes) of
//       the subtree with root n
int size (const tnode* n) {
  if (n){ // shortcut for n != nullptr
    return size(n->left) + size(n->right) + 1;
  }
  return 0;
}
```

Teilbäume auswerten

```
// POST: evaluates the subtree with root n
double eval(const tnode* n){
  assert(n);
  if (n->op == '=') return n->val;
  double l = 0;
  if (n->left) l = eval(n->left);
  double r = eval(n->right);
  switch(n->op){
    case '+': return l+r;
    case '-': return l-r;
    case '*': return l*r;
    case '/': return l/r;
    default: return 0;
  }
}
```

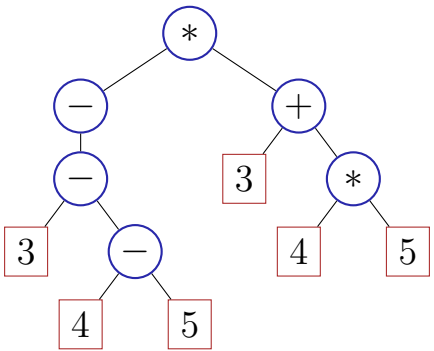
← Blatt...
...oder Astgabel:
← op unär, oder linker Ast
← rechter Ast

Teilbäume klonen

```
// POST: a copy of the subtree with root n is made
//       and a pointer to its root node is returned
tnode* copy (const tnode* n) {
  if (n == nullptr)
    return nullptr;
  return new tnode (n->op, n->val, copy(n->left), copy(n->right));
}
```

Teilbäume fällen

```
// POST: all nodes in the subtree with root n are deleted
void clear(tnode* n) {
  if(n){
    clear(n->left);
    clear(n->right);
    delete n;
  }
}
```



Teilbäume nutzen

```
// Construct a tree for 1 - (-(3 + 7))
tnode* n1 = new tnode('=', 3, nullptr, nullptr);
tnode* n2 = new tnode('=', 7, nullptr, nullptr);
tnode* n3 = new tnode('+', 0, n1, n2);
tnode* n4 = new tnode('-', 0, nullptr, n3);
tnode* n5 = new tnode('=', 1, nullptr, nullptr);
tnode* root = new tnode('-', 0, n5, n4);

// Evaluate the overall tree
std::cout << "1 - (-(3 + 7)) = " << eval(root) << '\n';

// Evaluate a subtree
std::cout << "3 + 7 = " << eval(n3) << '\n';

clear(root); // free memory
```

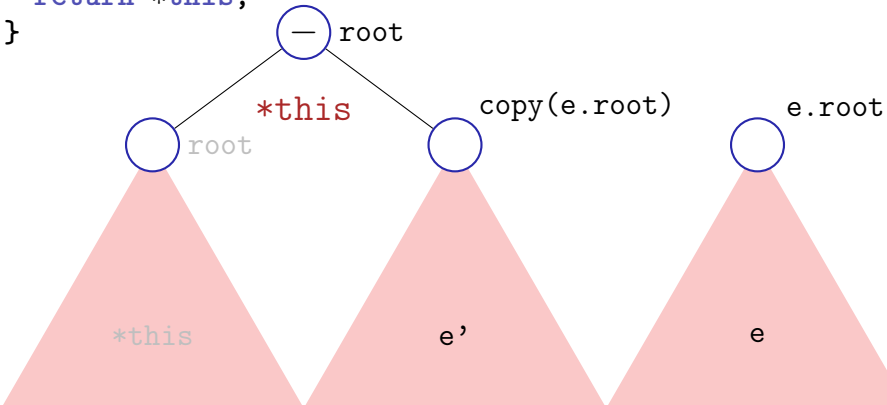
Bäume pflanzen

```
class texpression {
public:
    texpression (double d)
        : root (new tnode ('=', d, 0, 0)) {}
    ...
private:
    tnode* root;
};
```

erzeugt Baum mit einem Blatt

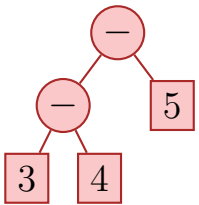
Bäume wachsen lassen

```
texpression& texpression::operator−= (const texpression& e)
{
  assert (e.root);
  root = new tnode ('−', 0, root, copy(e.root));
  return *this;
}
```



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;
```



Dreierregel: Bäume klonen, reproduzieren und fällen

```
texpression::~texpression(){
  clear(root);
}

texpresssion::texpression (const texpression& e)
     : root(copy(e.root)) { }

texpression::texpression& operator=(const texpression& e){
  if (root != e.root){
    texpression cp = e;
    std::swap(cp.root, root);
  }
  return *this;
}
```

Zusammengefasst

```
class texpression{
public:
  texpression (double d); // constructor
  ~texpression(); // destructor
  texpression (const texpression& e); // copy constructor
  texpression& operator=(const texpression& e); // assignment op
  texpression operator-();
  texpression& operator-=(const texpression& e);
  texpression& operator+=(const texpression& e);
  texpression& operator*=(const texpression& e);
  texpression& operator/=(const texpression& e);
  double evaluate();
private:
 tnode* root;
};
```

Werte zu Bäumen!

```
using number_type = texpression ;

// term = factor { "*" factor | "/" factor }
number_type term (std::istream& is){
  number_type value = factor (is);
  while (true) {
    if (consume (is, '*'))
      value *= factor (is);
    else if (consume (is, '/'))
      value /= factor (is);
  else
      return value;
  }
}
```

double_calculator.cpp
(Ausdruckswert)
$\rightarrow$
texpression_calculator.cpp
(Ausdrucksbaum)

Abschliessende Bemerkung

- Wir haben in dieser Vorlesung die Knoten für Liste und Baum bewusst ohne Memberfunktionen implementiert. Wir betrachten sie nämlich als reine Datencontainer ohne eigene Intelligenz.[9]
- Wenn Vererbung und Polymorhpie im Spiel ist, ist die Implementation der Funktionalität wie `evalute`, `print`, `clear`, `copy` (etc.) mit Memberfunktionen vorzuziehen.
- In jedem Falle implementiert man die Speicherverwaltung der zusammengesetzten Datenstruktur Liste / Baum nicht in den Knotenklassen.

[9] Teile der Implementation waren so sogar einfacher,da der Fall `n==nullptr` einfacher abgefangen werden kann

22. 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

Kapitelüberblick

- 1. Einführung
- 2. Ganze Zahlen
- 3. Wahrheitswerte
- 4. Defensives Programmieren
- 5./6. Kontrollanweisungen
- 7./8. Fliesskommazahlen
- 9./10. Funktionen
- 11. Referenztypen
- 12./13. Vektoren und Strings
- 14./15. Rekursion
- 16. Structs und Overloading
- 17. Klassen
- 18./19. Dynamische Datenstrukturen
- 20. Container, Iteratoren und Algorithmen
- 21. Dynamische Datentypen und Speicherverwaltung

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. Defensives Programmieren

Ⓚ
- Assertions und Konstanten

Ⓢ
- Die `assert`-Anweisung, `#include <cassert>`
- `const int speed_of_light=2999792458`

Ⓑ
- Assertions für den GGT

5./6. 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;`
- Switch Anweisung `switch(grade) {case 6: }`

Ⓑ
- Summenberechnung (Gauss), Primzahltest, Collatz-Folge, Fibonacci Zahlen, Taschenrechner, Notenausgabe

7./8. 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

9./10. 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

11. Referenztypen

Ⓜ
- Funktion Swap

Ⓚ
- Werte-/ Referenzsemantik, Pass by Value / Pass by Reference, Return by Reference
- Lebensdauer von Objekten / Temporäre Objekte
- Konstanten

Ⓢ
- Referenztyp `int& a`
- Call by Reference und Return by Reference `int& increment (int& i)`
- Const-Richtlinie, Const-Referenzen, Referenzrichtlinie

Ⓑ
- Swap, Inkrement

12./13. Vektoren und Strings

Ⓜ
- Iteration über Daten: Sieb des Eratosthenes

Ⓚ
- Vektoren, Speicherlayout, Wahlfreier Zugriff
- (Fehlende) Grenzenprüfung
- Vektoren
- Zeichen: ASCII, UTF8, Texte, Strings

Ⓢ
- Vektor Typen `std::vector<int> a {4,3,5,2,1};`
- Zeichen und Texte, der Typ char `char c = 'a';`, Konversion nach **`int`**
- Vektoren von Vektoren
- Ströme `std::istream, std::ostream`

Ⓑ
- Sieb des Eratosthenes, Caesar-Code, Kürzeste Wege

14./15. Rekursion

Ⓜ
- Rekursive math. Funktionen, Das n-Queen Problem, , Lindenmayer-Systeme, Kommandozeilenrechner

Ⓚ
- Rekursion
- Aufrufstapel, Gedächtnis der Rekursion
- Korrektheit, Terminierung,
- Rekursion vs. Iteration
- Backtracking, EBNF, Formale Grammatiken, Parsen

Ⓑ
- Fakultät, GGT, Sudoku-Löser, Taschenrechner

16. Structs und Overloading

Ⓜ
- 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

17. Klassen

Ⓜ
- Rationale Zahlen mit Kapselung

Ⓚ
- Kapselung, Konstruktion, Mitgliedsfunktionen

Ⓢ
- Klassen `class rational { ... };`
- Zugriffssteuerung `public:` / `private:`
- Mitgliedsfunktionen `int rational::denominator () const`
- Das implizite Argument der Memberfunktionen

Ⓑ
- Endlicher Ring, Komplexe Zahlen

18./19. Dynamische Datenstrukturen

Ⓜ
- Unser eigener Vektor

Ⓚ
- Allokation, Zeiger-Typen, Verkettete Liste, Allokation, Deallokation, Dynamischer Datentyp

Ⓢ
- Die `new` Anweisung
- Zeiger `int* x;`, Nullzeiger `nullptr`.
- Adress-, Dereferenzoperator `int *ip = &i; int j = *ip;`
- Zeiger und Const `const int *a;`

Ⓑ
- Verkettete Liste, Stack

20. Container, Iteratoren und Algorithmen

Ⓜ
- Vektoren sind Container

Ⓚ
- Iterieren mit Zeigern
- Container und Iteratoren
- Algorithmen

Ⓢ
- Iteratoren `std::vector<int>::iterator`
- Algorithmen der Standardbibliothek `std::fill (a, a+5, 1);`
- Einen Iterator implementieren
- Iteratoren und const

Ⓑ
- Ausgeben eines Vektors, einer Menge

21. Dynamische Datentypen und Speicherverwaltung

Ⓜ
- Stack
- Ausdrucksbaum

Ⓚ
- Richtlinie „Dynamischer Speicher”
- Gemeinsamer Zeiger-Zugriff
- Dynamischer Datentyp
- Baumstruktur

Ⓢ
- `new` und `delete`
- Desktruktor `stack::~stack()`
- Kopierkonstruktor `stack::stack(const stack& s)`
- Zuweisungsoperator `stack& stack::operator=(const stack& s)`
- Dreierregel

Ⓑ
- Binärer Suchbaum

Ende

Ende der Vorlesung.