Informatik I

Vorlesung am D-ITET der ETH Zürich

Felix Friedrich

HS 2017

Willkommen

zur Vorlesung Informatik I !

am ITET Department der ETH Zürich.

Ort und Zeit:

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

Vorlesungshomepage

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

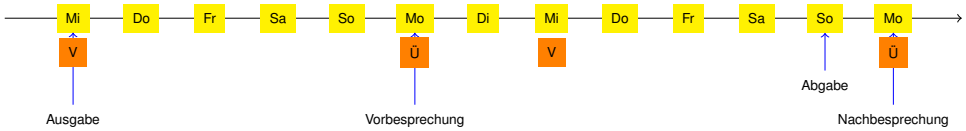
Team

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

Einschreibung in Übungsgruppen

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

Ablauf



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

Zu den Übungen

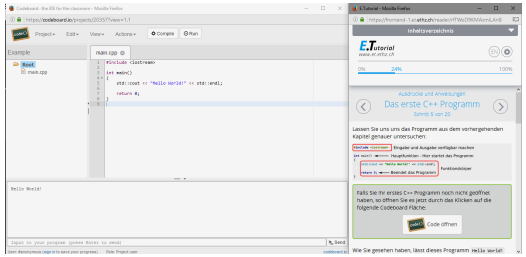
- An der ETH ist (seit HS 2013) für die Prüfungszulassung kein Testat erforderlich.
- Bearbeitung der wöchentlichen Uebungsserien ist also freiwillig, wird aber *dringend* empfohlen!

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 2018) schliesst ein

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

Prüfung ist schriftlich. Es sind keine Hilfsmittel zugelassen.

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

[1] soweit in schriftlicher Prüfung möglich

Unser Angebot

- Ihre Programmierübungen werden (halb)automatisch bewertet. Durch Bearbeitung der wöchentlichen Übungsserien kann ein Bonus von maximal 0.25 Notenpunkten erarbeitet werden, der an die Prüfung mitgenommen wird.
- Der Bonus ist proportional zur erreichten Punktzahl über alle Serien, wobei volle Punktzahl einem Bonus von 0.25 entspricht.

Akademische Lauterkeit

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

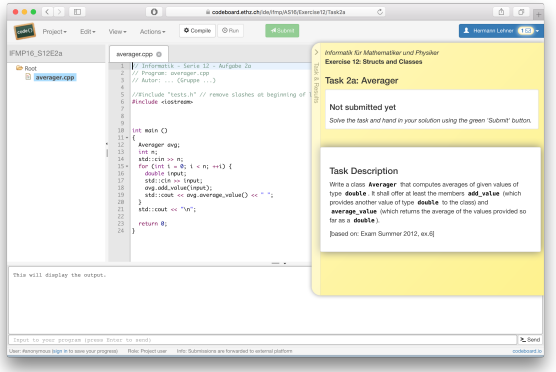
Wir prüfen das (zum Teil automatisiert) nach und behalten uns insbesondere mündliche Prüfgespräche vor.

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

Codeboard

Codeboard ist eine Online-IDE: Programmieren im Browser!

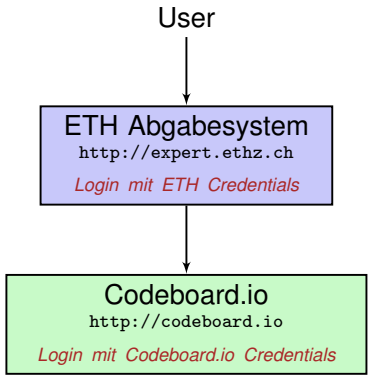
- Falls vorhanden, bringen Sie Ihren Laptop/Tablet/... mit in den Unterricht.
- Sie können direkt in der Vorlesung Beispiele ausprobieren, ohne dass Sie irgendwelche Tools installieren müssen.



Expert

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

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



Übungseinschreibung

Codeboard.io Registrierung

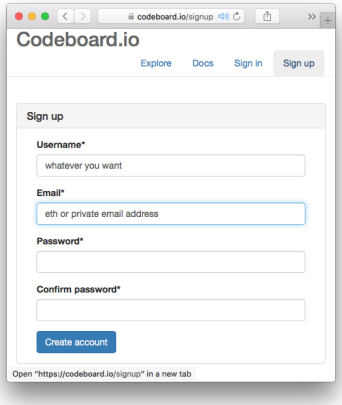
Gehen Sie auf `http://codeboard.io` und erstellen Sie dort ein Konto, bleiben Sie am besten eingeloggt.

Einschreibung in Übungsgruppen

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

Codeboard.io Registrierung

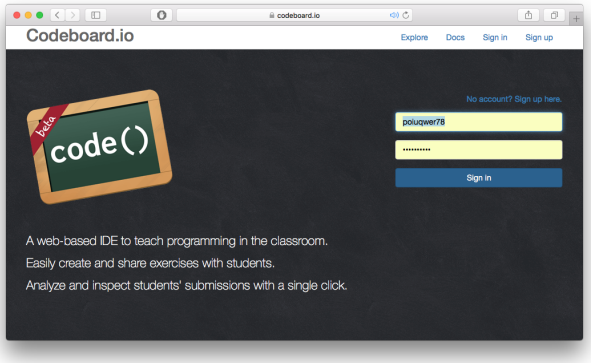
Falls Sie noch keinen **Codeboard.io** Account haben ...



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

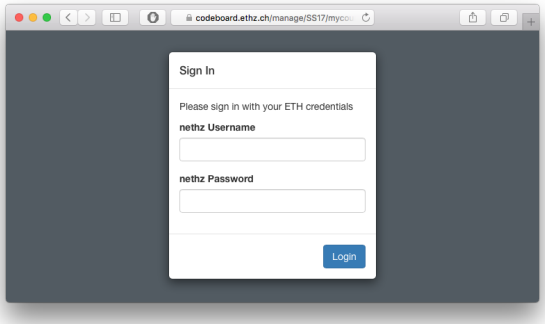
Codeboard.io Login

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



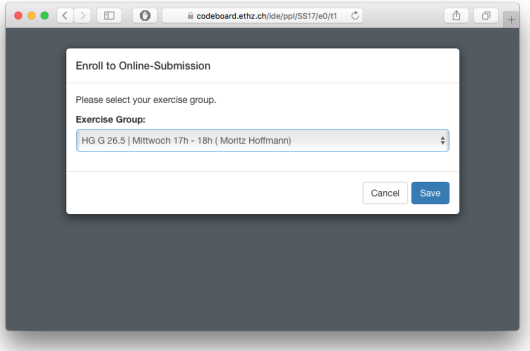
Einschreibung in Übungsgruppen - I

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



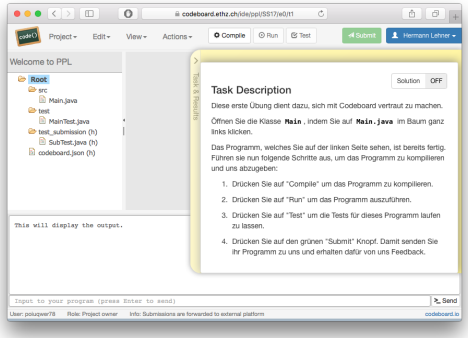
Einschreibung in Übungsgruppen - II

Schreiben Sie sich in diesem Dialog in eine Übungsgruppe ein.



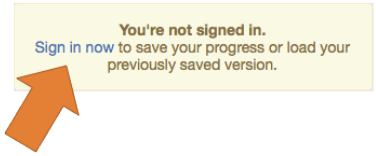
Die erste Übung

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



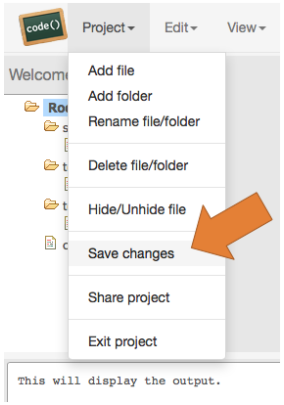
Die erste Übung - Codeboard.io Login

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



Die erste Übung - Fortschritt speichern!

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



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

- Aufbau dieser **Vorlesung und Folien** gemeinsam mit **Prof. Bernd Gärtner (ETH)** erarbeitet.
- Skript von Prof. Bernd Gärtner.

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 $\neq$ Computer Science

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

Computer Science $\subseteq$ Informatik

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

Informatik $\neq$ EDV-Kenntnisse

EDV-Kenntnisse: *Anwenderwissen*

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

Informatik: *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)

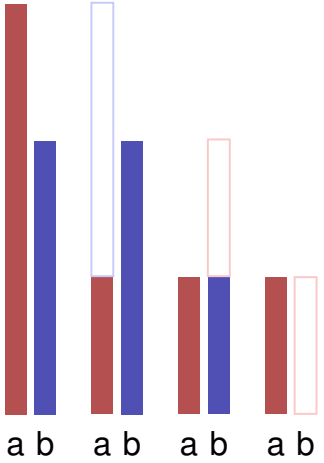
http://de.wikipedia.org/wiki/Algorithmus

Der älteste nichttriviale Algorithmus

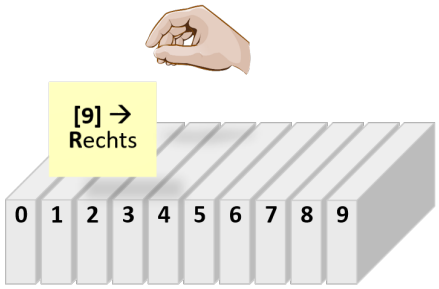
Euklidischer Algorithmus (aus Euklids *Elementen*, 3. Jh. v. Chr.)

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

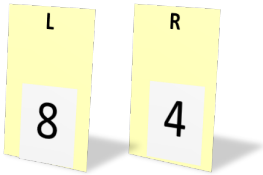
Solange $b \neq 0$
  Wenn $a > b$ dann
    $a \leftarrow a - b$
  Sonst:
    $b \leftarrow b - a$
Ergebnis: a.



Live Demo: Turing Maschine



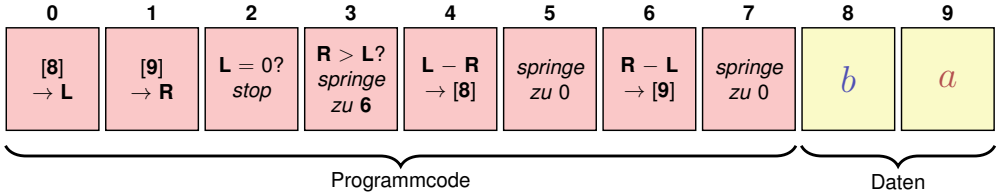
Speicher

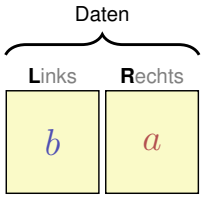


Register

Euklid in der Box

Speicher





Register

Solange $b \neq 0$

Wenn $a > b$ dann

$a \leftarrow a - b$

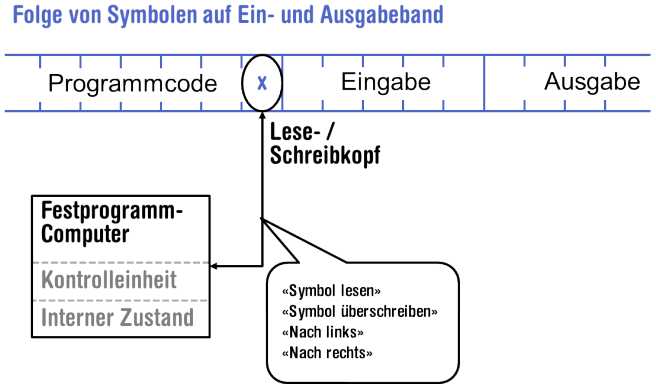
Sonst:

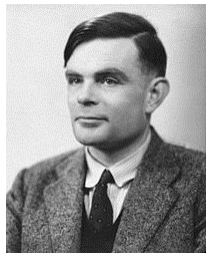
$b \leftarrow b - a$

Ergebnis: a.

Computer – Konzept

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



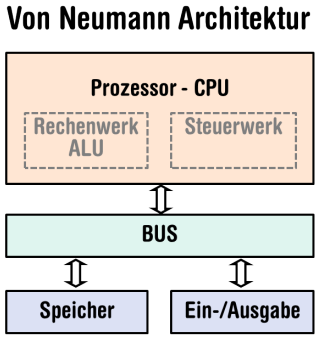


Alan Turing

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

Computer – Umsetzung

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



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

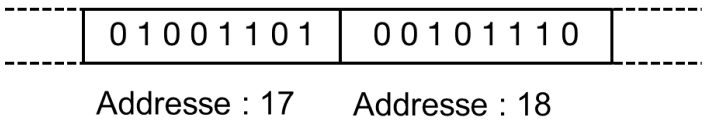
Computer

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.[2]

[2] Uniprozessor Computer bei 1GHz

Programmieren

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

The Harvard Computers, Menschliche Berufsrechner, ca.1890

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

Darum Programmieren!

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

Programmiersprachen

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

Höhere Programmiersprachen

darstellbar als Programmtext, der

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

Programmiersprachen – Einordnung

Unterscheidung in

- *Kompilierte* vs. interpretierte Sprachen
  - *C++*, C#, Pascal, Modula, Oberon, Java
    vs.
    Python, Tcl, 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, Modula, Oberon, Python ...

Allgemeiner Konsens

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

Warum C++?

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

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 → 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 gallopiert 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

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

Semantik

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

Syntax und Semantik von C++

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

- ist das “Gesetz” von C++
- legt Grammatik und Bedeutung von C++-Programmen fest
- enthält seit 2011 Neuerungen für *fortgeschrittenes* Programmieren. . .
- . . . weshalb wir auf diese Neuerungen hier auch nicht weiter eingehen werden.

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;
    // computation
    int b = a * a; // b = a^2
    b = b * b;     // b = a^4
    // output b * b, i.e., a^8
    std::cout << a << "^8 = " << b * b << "\n";
    return 0;
}
```

Anweisungen: Mache etwas (lies a ein)!

Ausdrücke: Berechne einen Wert (a^2)!

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

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;
}
```

Ausdrucksanweisungen

Rückgabeanweisung

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;
}
```

Typ-namen

Deklarationsanweisungen

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`:

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

In 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.

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

Ausdrücke — 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

```
// 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;
```

R-Wert

L-Wert (Ausdruck + Adresse)

L-Wert (Ausdruck + Adresse)

R-Wert

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

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

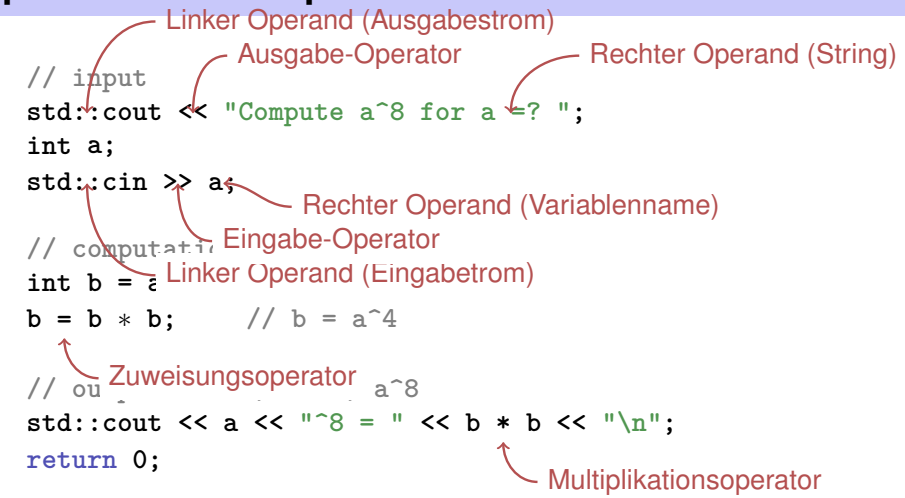
L-Werte und R-Werte

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*.

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 =

- Linker Operand ist L-Wert,
- Rechter 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.

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

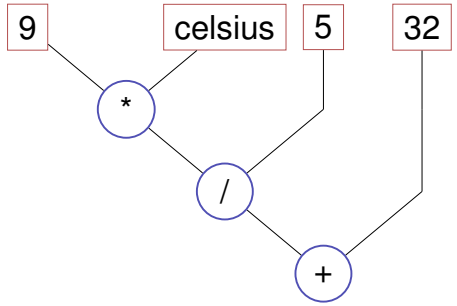
- Assoziativitäten
- Präzedenzen
- Stelligkeiten (Anzahl Operanden)

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

Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

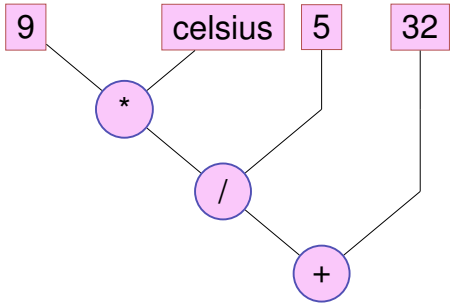
```
(((9 * celsius) / 5) + 32)
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

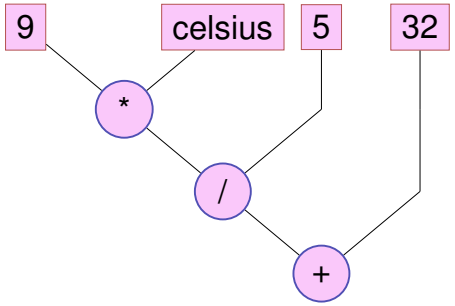
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

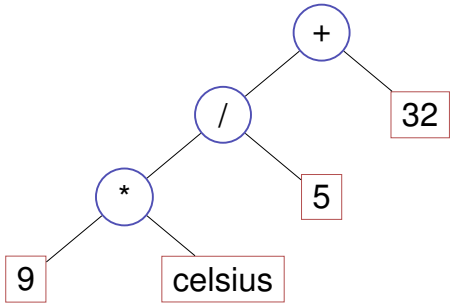
```
9 * celsius / 5 + 32
```



Ausdrucksbäume – Notation

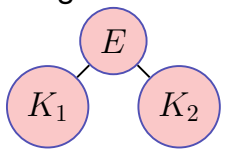
Üblichere Notation: Wurzel oben

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

Auswertungsreihenfolge

Richtlinie

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

Arithmetische Operatoren

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

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

Zuweisungsausdruck – nun genauer

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

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

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

Division und Modulus

- Operator `/` realisiert ganzzahlige Division

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

- In `fahrenheit.cpp`

```
9 * celsius / 5 + 32

15 degrees Celsius are 59 degrees Fahrenheit
```

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

```
9 / 5 * celsius + 32

15 degrees Celsius are  47 degrees Fahrenheit
```

Division und Modulus

- Modulus-Operator berechnet Rest der ganzzahligen Division

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

- 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 (Effekte!)

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 → R-Wert
Prä-Inkrement	`++expr`	1	16	rechts	L-Wert → L-Wert
Post-Dekrement	`expr--`	1	17	links	L-Wert → R-Wert
Prä-Dekrement	`--expr`	1	16	rechts	L-Wert → 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
```

⇔

```
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.

Niedrigstes Bit, Least Significant Bit (LSB)

Höchstes Bit, Most Significant Bit (MSB)

Binäre Zahlen: Zahlen der Computer?

Wahrheit: Computer rechnen mit Binärzahlen.


NEUE ZÜRCHER ZEITUNG

TECHNIK

Das programmgesteuerte Rechengerät an der Eidgenössischen Technischen Hochschule in Zürich

Merkmale des Gerätes

Das duale Zahlsystem

Binäre Zahlen: Zahlen der Computer?

Klischee: Computer reden 0/1-Kauderwelsch.

Rechentricks

- Abschätzung der Grössenordnung von Zweierpotenzen[3]:

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

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

Hexadezimale Zahlen

Zahlen zur Basis 16. Darstellung

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

entspricht der Zahl

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

Schreibweise in C++: vorangestelltes `0x`

Beispiel: `0xff` entspricht 255.

Hex Nibbles

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

Wozu Hexadezimalzahlen?

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

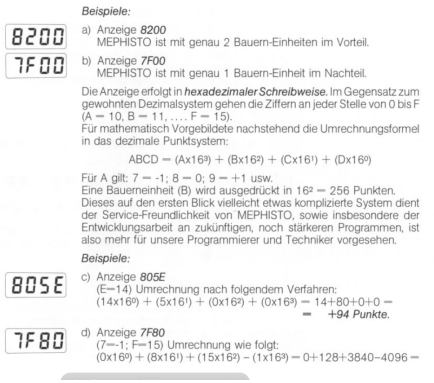
32-bit Zahlen bestehen aus acht Hex-Nibbles: `0x00000000 -- 0xffffffff` .
`0x400` $= 1Ki = 1'024$.
`0x100000` $= 1Mi = 1'048'576$.
`0x40000000` $= 1Gi = 1'073.741,824$.
`0x80000000`: höchstes Bit einer 32-bit Zahl gesetzt.
`0xffffffff`: alle Bits einer 32-bit Zahl gesetzt.
„`0x8a20aaf0` ist eine Addresse in den oberen 2G des 32-bit Addressraumes”

Beispiel: Hex-Farben

#00FF00
r g b

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

Wozu Hexadezimalzahlen?

Die NZZ hätte viel Platz sparen können...

Wertebereich des Typs `int`

```
// Program: limits.cpp
// Output the smallest and the largest value of type int.

#include <iostream>
#include <limits>

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

Zum Beispiel

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

Woher kommen diese Zahlen?

Wertebereich des Typs `int`

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

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

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

Woher kommt gerade diese Aufteilung?

Überlauf und Unterlauf

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

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

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

- Es gibt *keine Fehlermeldung!*

Der Typ `unsigned int`

- Wertebereich

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

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

Gemischte Ausdrücke

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

```
17 + 17u
```

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

Konversion

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

Bei Zweierkomplementdarstellung passiert dabei intern gar nichts

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.

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

- 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

2	0010
$+3$	$+0011$
5	0101

Einfache Subtraktion

5	0101
-3	-0011
2	0010

Rechnen mit Binärzahlen (4 Stellen)

Addition mit Überlauf

7	0111
$+9$	$+1001$
16	$(1)0000$

Negative Zahlen?

5	0101
$+(-5)$	$????$
0	$(1)0000$

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!

3	0011
$+(-4)$	$+1100$
-1	$1111 \mathrel{\widehat{=}} 2^B - 1$

a	a
$+(-a-1)$	$\bar{a}$
-1	$1111 \mathrel{\widehat{=}} 2^B - 1$

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[4]

$11 \equiv 23 \equiv -1 \equiv \ldots \mod 12$ $\quad$ $4 \equiv 16 \equiv \ldots \mod 12$ $\quad$ $3 \equiv 15 \equiv \ldots \mod 12$

[4] 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.

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”.
- 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)

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

R-Wert $\times$ R-Wert $\to$ 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”.
- 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)

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

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

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


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

- "Logisches Nicht"

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

- 0 entspricht „falsch".
- entspricht „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 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}

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

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`	→	`int`
true	→	1
false	→	0
`int`	→	**`bool`**
≠0	→	*true*
0	→	*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

Fehlerquellen

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

Fehlerquellen vermeiden

1. Genaue Kenntnis des gewünschten Programmverhaltens

   ≫ It's not a bug, it's a feature !!≪
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

DeMorgansche Regeln

Hinterfrage das Offensichtliche! Hinterfrage das scheinbar Offensichtliche!

```
// Prog: assertion.cpp
// use assertions to check De Morgan's laws

#include<cassert>

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


Assertions abschalten

```
// Prog: assertion2.cpp
// use assertions to check De Morgan's laws. To tell the
// compiler to ignore them, #define NDEBUG ("no debugging")
// at the beginning of the program, before the #includes

#define NDEBUG
#include<cassert>

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

Div-Mod Identität `a/b * b + a%b == a`

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

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

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

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

Eingabe der Argumente für die Berechnung

Vorbedingung für die weitere Berechnung

Div-Mod Identität `a/b * b + a%b == a`

...und hinterfrage das Offensichtliche!

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

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

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

Vorbedingung für die weitere Berechnung

Div-Mod Identität

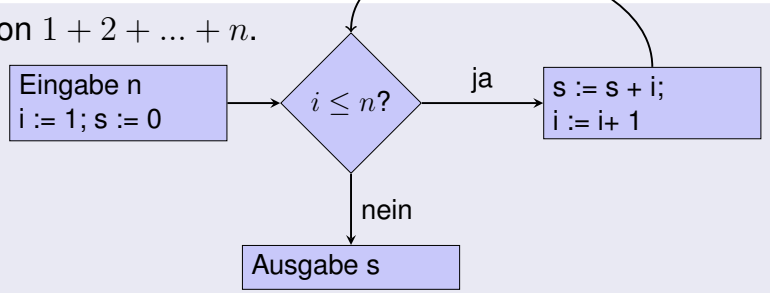
4. Kontrollanweisungen I

Auswahlanweisungen, Iterationsanweisungen, Terminierung, Blöcke

Kontrollfluss

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

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



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

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*)
statement

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

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

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

`for`-Anweisung: Terminierung

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

Hier und meistens:

- *expression* ändert einen Wert, der in *condition* vorkommt.
- 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 ( e; v; e) r;
```


Halteproblem

Unentscheidbarkeit des Halteproblems

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

Das heisst, die Korrektheit von Programmen kann *nicht* automatisch überprüft werden.[5]

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

Beispiel: Primzahltest

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

Eine Schleife, die das testet:

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

- Beobachtung 1:
  Nach der `for`-Anweisung gilt $d \leq n$.
- Beobachtung 2:
  n ist Primzahl genau wenn am Ende $d = n$.

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";
}
```

5. Kontrollanweisungen II

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

Sichtbarkeit

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

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

main block

block

„Blickrichtung”

Kontrollanweisung definiert Block

Kontrollanweisungen verhalten sich in diesem Zusammenhang wie Blöcke.

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

block

Gültigkeitsbereich einer Deklaration

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

Im Block

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

scope

Im Funktionsrumpf

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

scope

In Kontrollanweisung

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

scope

Gültigkeitsbereich 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;
}
```

in main

i_2 in for

Gültigkeit von i

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 [6]

[6] 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

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

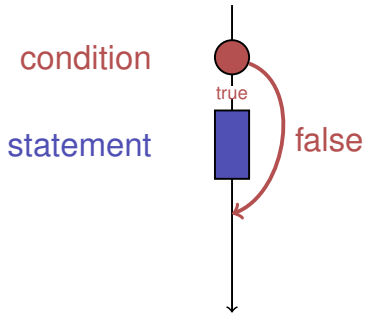
Es gilt aber sogar: Nicht ganz so einfach falls ein continue im Spiel ist!

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

Kontrollfluss

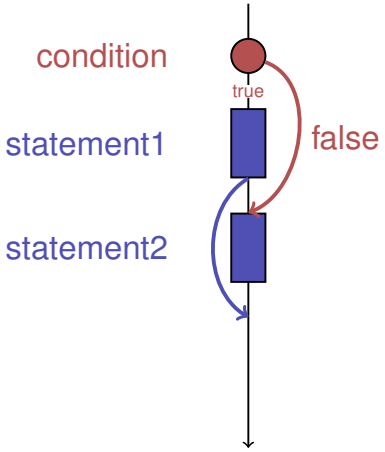
Reihenfolge der (wiederholten) Ausführung von Anweisungen

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



`if` (*condition*)
    statement

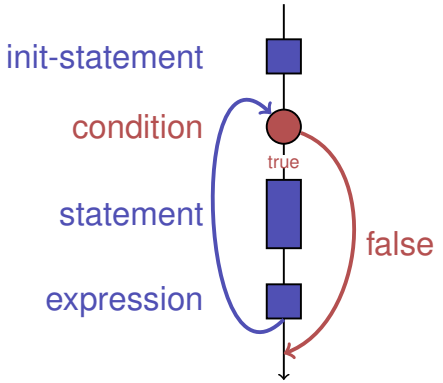
Kontrollfluss `if else`



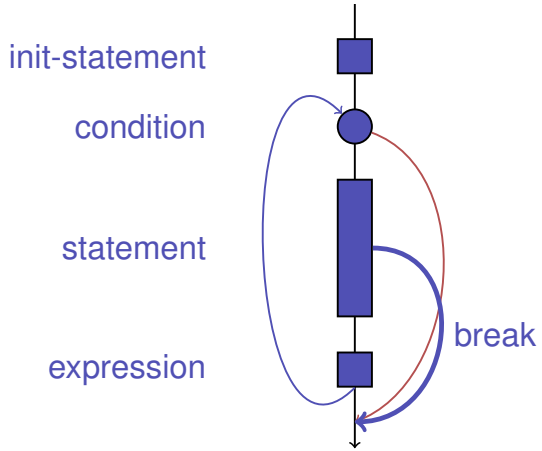
`if` (*condition*)
    statement1
else
    statement2

Kontrollfluss `for`

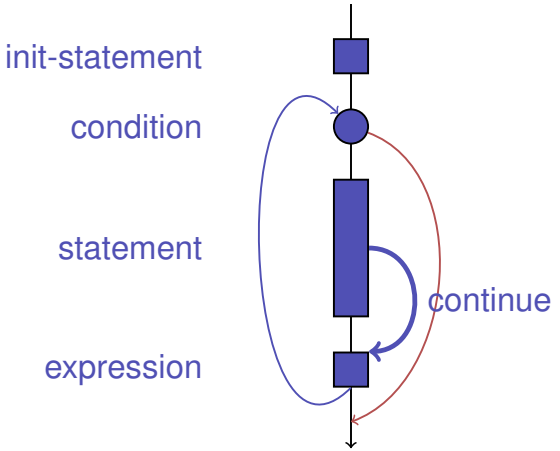
`for` (*init statement* *condition* `;` *expression*)
    statement



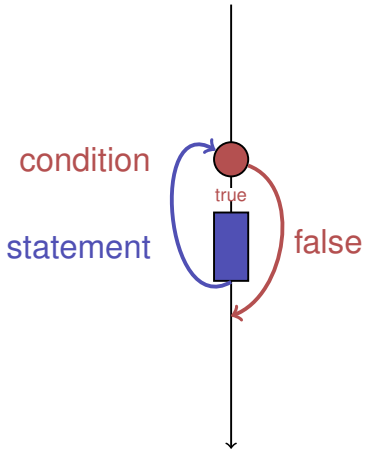
Kontrollfluss `break` in for



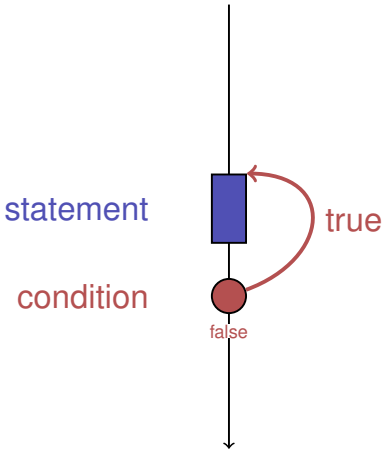
Kontrollfluss `continue` in for



Kontrollfluss `while`



Kontrollfluss `do while`



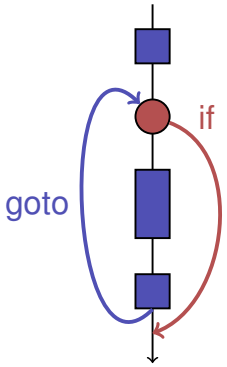
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

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

Modelle:

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



BASIC und die Home-Computer...

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

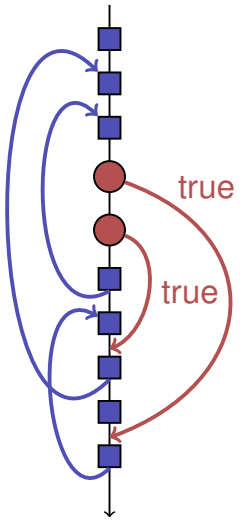
Home-Computer Commodore C64 (1982)

http://de.wikipedia.org/wiki/Commodore_64

Spaghetti-Code mit `goto`

Ausgabe aller Primzahlen mit der Programmiersprache BASIC

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



Die „richtige” Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

- 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

Die `switch`-Anweisung

`switch` (*condition*)
statement

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

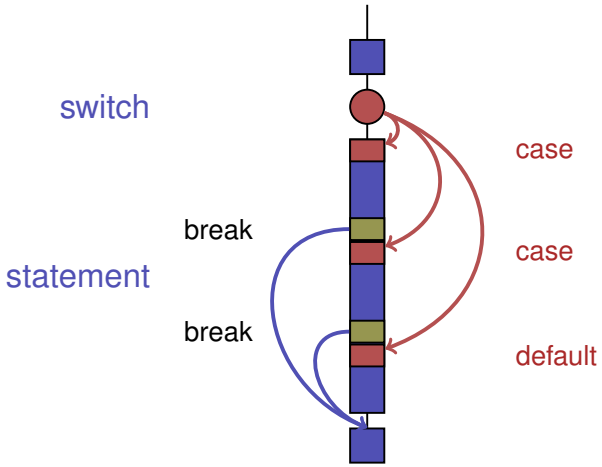
```
int Note;
...
switch (Note) {
    case 6:
        std::cout << "super!";
        break;
    case 5:
        std::cout << "cool!";
        break;
    case 4:
        std::cout << "ok.";
        break;
    default:
        std::cout << "hmm...";
}
```


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`



Kontrollfluss `switch` allgemein

Fehlt `break`, geht es mit dem nächsten Fall weiter.

7: ???
6: ok.
5: ok.
4: ok.
3: oops!
2: ooops!
1: oooops!
0: ???

```
switch (Note) {
    case 6:
    case 5:
    case 4:
        std::cout << "ok.";
        break;
    case 1:
        std::cout << "o";
    case 2:
        std::cout << "o";
    case 3:
        std::cout << "oops!";
        break;
    default:
        std::cout << "???";
}
```

6. 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

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

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

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

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

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

Arithmetische Operatoren

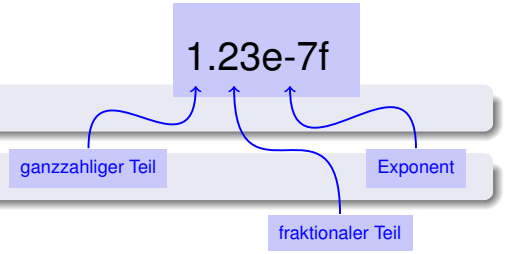
Wie bei `int`, aber ...

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

Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

- Dezimalkomma
- und / oder Exponent.



`1.0` : Typ `double`, Wert 1

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

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

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

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

Rechnen mit `float`: Beispiel

Approximation der Euler-Zahl

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

mittels der ersten 10 Terme.

Rechnen mit `float`: Eulersche Zahl

```
// Program: euler.cpp
// Approximate the Euler number e.

#include <iostream>

int main ()
{
  // values for term i, initialized for i = 0
  float t = 1.0f;   // 1/i!
  float e = 1.0f;   // i-th approximation of e

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

  return 0;
}
```


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
```


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”.

Löcher im Wertebereich

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

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

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

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

Eingabe 1.1

Eingabe 1.0

Eingabe 0.1

Ausgabe 2.23517e-8

Was ist denn hier los?

7. Fliesskommazahlen II

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

Fliesskommazahlensysteme

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert:

- $\beta \geq 2$, die Basis,
- $p \geq 1$, 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

Beispiel

- $\beta = 10$

Darstellungen der Dezimalzahl 0.1

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

Normalisierte Darstellung

Normalisierte Zahl:

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

Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

Bemerkung 2

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

Menge der normalisierten Zahlen

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

Normalisierte Darstellung

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

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

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, $0 < x < 2$.

- 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 $x' = 2 \cdot (x - b_0)$)

Umrechnung dezimal $\rightarrow$ binär

Angenommen, $0 < x < 2$.

Binärdarstellung:

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

Binärdarstellung von 1.1

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

$\Rightarrow 1.0\overline{00011}$, 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`: $1.1 - 1.0 \neq 0.1$

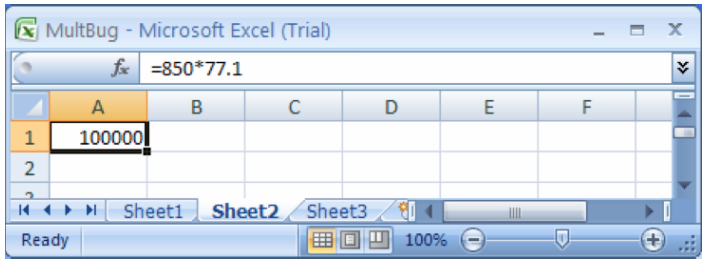
Binärdarstellungen von 1.1 und 0.1

auf meinem Computer:

$$\begin{aligned} \texttt{1.1} &= \underline{1.100000000000000}888178\ldots \\ \texttt{1.1f} &= \underline{1.1000000}238418\ldots \end{aligned}$$

Der Excel-2007-Bug

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



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

Rechnen mit Fliesskommazahlen

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

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

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:

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

- Double precision (`double`) Zahlen:

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

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

Der IEEE Standard 754

Warum

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

- 1 Bit für das Vorzeichen
- 23 Bit für den Signifikanden (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.

Fliesskomma-Richtlinien Regel 1

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.


```
// 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;
}
```


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

Fliesskomma-Richtlinien — Regel 3

Regel 3

Subtrahiere keine zwei Zahlen sehr ähnlicher Grösse!

Auslöschungsproblematik, siehe Skript.

Literatur

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

Randy Glasbergen, 1996

8. Funktionen I

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

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

⇒ 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 << ".\n";
```

"Funktion pow"

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


Funktion zur Potenzberechnung

```
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow(double b, int e)
{
    double result = 1.0;
    if (e < 0) { // b^e = (1/b)^(-e)
        b = 1.0/b;
        e = -e;
    }
    for (int i = 0; i < e; ++i)
        result *= b;
    return result;
}
```


Funktion zur Potenzberechnung

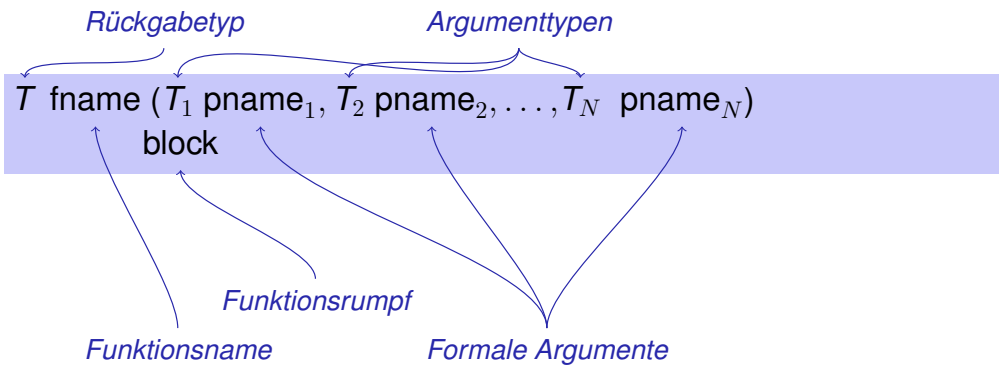
```
// Prog: callpow.cpp
// Define and call a function for computing powers.
#include <iostream>

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 || !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
- Funktionsaufruf selbst ist R-Wert.

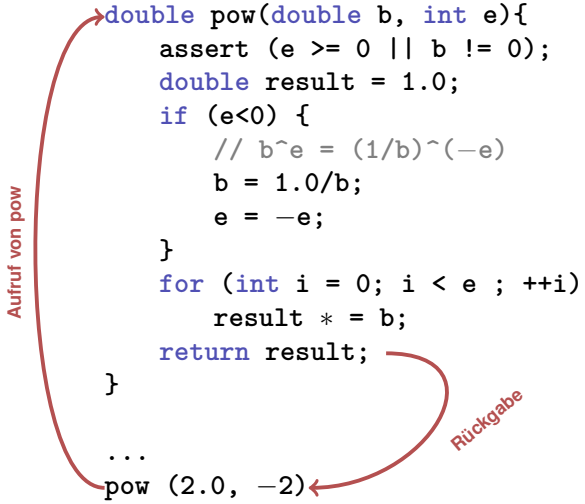
fname: R-Wert $\times$ R-Wert $\times \cdots \times$ R-Wert $\longrightarrow$ R-Wert

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[7]

- 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)

[7] 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`

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

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

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

`void`-Funktionen

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

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

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. ⇒ 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.

9. Funktionen II

Stepwise Refinement, Gültigkeitsbereich, Bibliotheken, Standardfunktionen

Stepwise Refinement

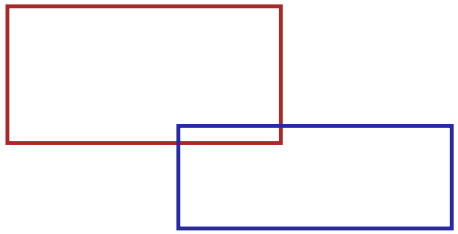
- Einfache *Programmiertechnik* zum Lösen komplexer Probleme

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)

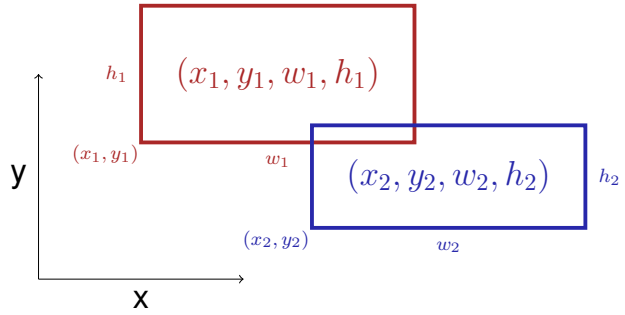
```
int main()
{
    // Eingabe Rechtecke

    // Schnitt?

    // Ausgabe der Loesung

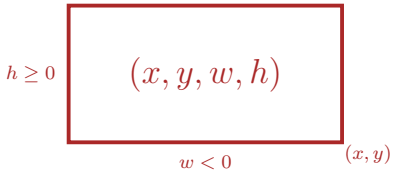
    return 0;
}
```


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
}
```

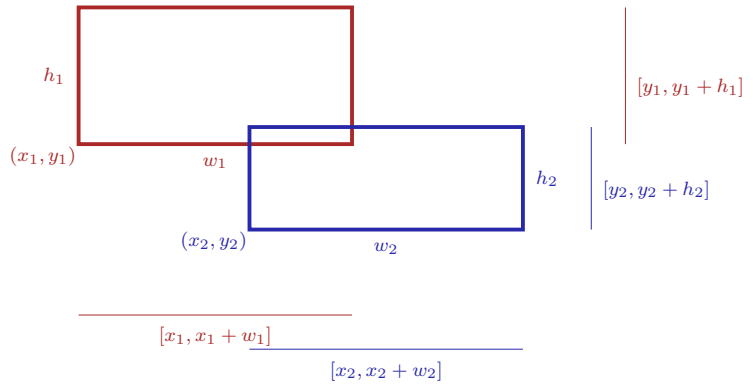
Funktion main ✓

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!

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

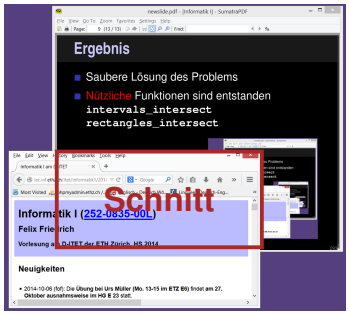
// PRE: [a1, b1], [a2, h2] are (generalized) intervals,
//      with [a,b] := [b,a] if a>b
// POST: returns true if [a1, b1],[a2, b2] intersect
bool intervals_intersect (int a1, int b1, int a2, int b2)
{
  return std::max(a1, b1) >= std::min(a2, b2)
      && std::min(a1, b1) <= std::max(a2, b2);
}

// PRE: (x1, y1, w1, h1), (x2, y2, w2, h2) are rectangles, where
//      w1, h1, w2, h2 may be negative.
// POST: returns true if (x1, y1, w1, h1),(x2, y2, w2, h2) intersect
bool rectangles_intersect (int x1, int y1, int w1, int h1,
                           int x2, int y2, int w2, int h2)
{
    return intervals_intersect (x1, x1 + w1, x2, x2 + w2)
        && intervals_intersect (y1, y1 + h1, y2, y2 + h2);
}
```

```
int main ()
{
  std::cout << "Enter two rectangles [x y w h each]\n";
  int x1, y1, w1, h1;
  std::cin >> x1 >> y1 >> w1 >> h1;
  int x2, y2, w2, h2;
  std::cin >> x2 >> y2 >> w2 >> h2;
  bool clash = rectangles_intersect (x1,y1,w1,h1,x2,y2,w2,h2);
  if (clash)
    std::cout << "intersection!\n";
  else
    std::cout << "no intersection!\n";
  return 0;
}
```

Ergebnis

- Saubere Lösung des Problems
- Nützliche Funktionen sind entstanden
  `intervals_intersect`
  `rectangles_intersect`



Wo darf man eine Funktion benutzen?

```
#include<iostream>

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

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

Gültigkeit f

Gültigkeitsbereich einer Funktion

- ist der Teil des Programmes, in dem die Funktion aufgerufen werden kann
- 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;
}
```

Gültigkeit f

. . . aber so!

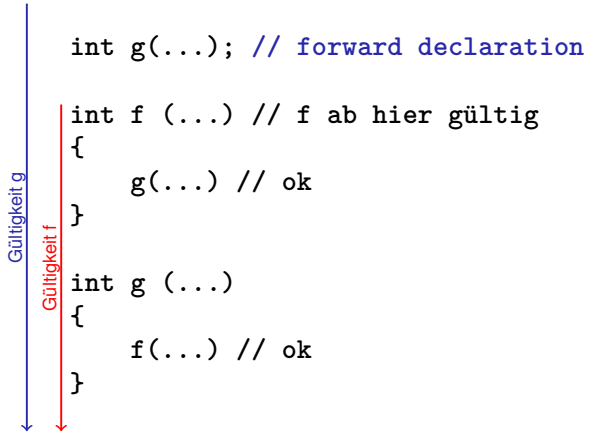
```
#include<iostream>
int f (int i); // Gueltigkeitsbereich von f ab hier

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

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


Forward Declarations, wozu?

Funktionen, die sich gegenseitig aufrufen:




```
int 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 "math.cpp"

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

  return 0;
}
```

Datei im Arbeitsverzeichnis

Nachteil des Inkludierens

- `#include` kopiert die Datei (`math.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 `math.cpp` unabhängig vom Hauptprogramm:

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

math.cpp

g++ -c math.cpp

```
0011101011001010100
0001011101010001111
00010 Funktion pow 1
1111000011010100011
1111111010001110100
0101011010110100011
1001011111001010100
```

math.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);
```

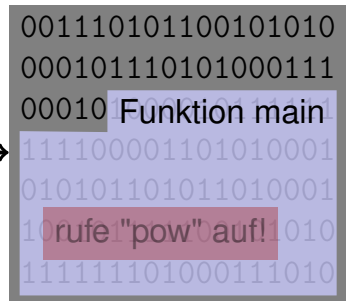
math.h

Level 2: Getrennte Übersetzung

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

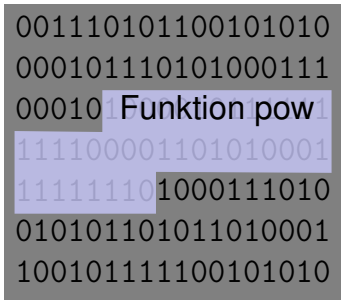
```
#include <iostream>
#include "math.h"
int main()
{
  std::cout << pow(2,-2) << "\n";
  return 0;
}
```

callpow3.cpp



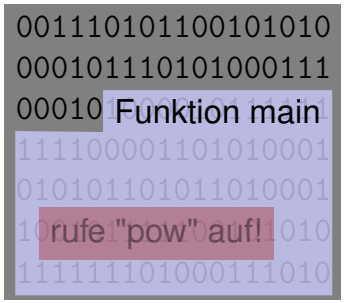
callpow3.o

Der Linker vereint...



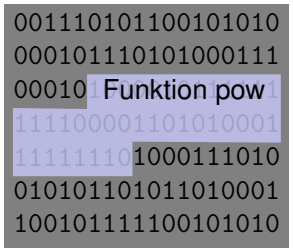
math.o

+



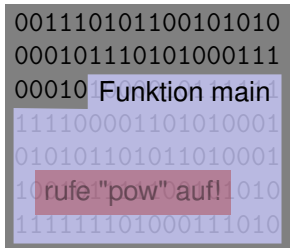
callpow3.o

... was zusammengehört



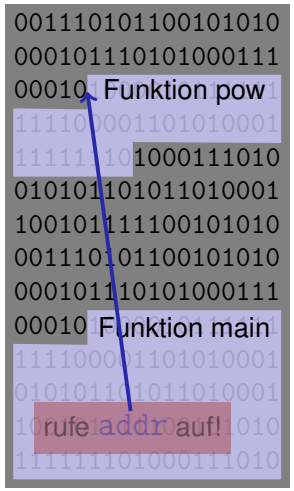
math.o

+



callpow3.o

=



Ausführbare Datei callpow

Verfügbarkeit von Quellcode?

Beobachtung

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

Viele Anbieter von Funktionsbibliotheken liefern dem Benutzer keinen Quellcode.

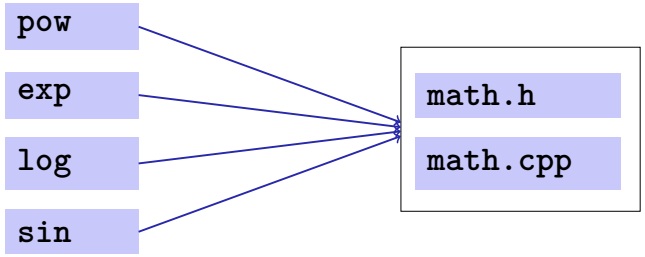
Header-Dateien sind dann die *einzigen* lesbaren Informationen.

„Open Source” Software

- Alle Quellcodes sind verfügbar.
- Nur das erlaubt die Weiterentwicklung durch Benutzer und engagierte “Hacker”.
- Selbst im kommerziellen Bereich ist „open source” 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...

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

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

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


...vermeiden Namenskonflikte

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

int main()
{
    double x = std::pow (2.0, -2); // <cmath>
    double y = ifee::pow (2.0, -2); // ifeemath.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 `ifee::pow`);
- führen auf einfache Weise zu interessanten und effizienten Programmen;
- garantierten einen Qualitäts-Standard, der mit selbstgeschriebenen Funktionen kaum erreicht werden kann.

Primzahltest mit `sqrt`

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \ldots, n-1\}$ ein Teiler von n ist.

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


Primzahltest mit `sqrt`

$n \geq 2$ ist Primzahl genau dann, wenn kein d in $\{2, \ldots, \lfloor\sqrt{n}\rfloor\}$ ein Teiler von n ist.

```
unsigned int bound = std::sqrt(n);
unsigned int d;
for (d = 2; d <= bound && n % d != 0; ++d);
```

- Das funktioniert, weil `std::sqrt` auf die nächste darstellbare `double`-Zahl rundet (IEEE Standard 754).
- Andere mathematische Funktionen (`std::pow`,...) sind in der Praxis fast so genau.

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

Referenztypen (z.B. `int&`)

10. Referenztypen

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

Swap!

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

Referenztypen

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

Referenztypen

Referenztypen: Definition

T& Gelesen als „*T*-Referenz”

Zugrundeliegender Typ

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

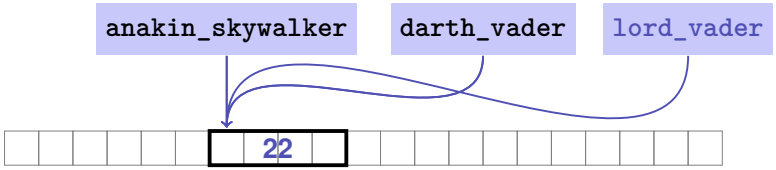
Anakin Skywalker alias Darth Vader

Anakin Skywalker alias Darth Vader

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

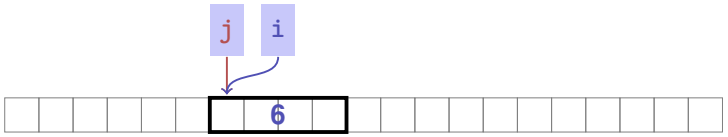
```
int& j;  // Fehler: j muss Alias von irgendetwas sein

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

Call 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
```



Call by Reference

Formales Argument hat Referenztyp:

⇒ **Call by Reference**

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

Call by Value

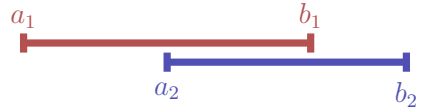
Formales Argument hat keinen Referenztyp:

⇒ **Call by Value**

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

Im Kontext: Zuweisung an Referenzen

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



Im Kontext: Initialisierung von Referenzen

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

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


Return by Value / Reference

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

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

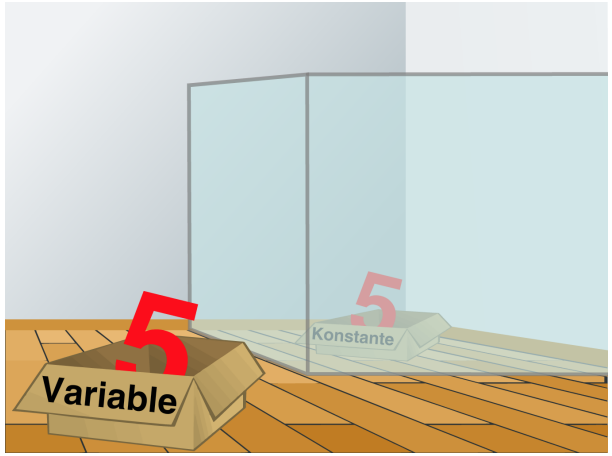
- Compiler kontrolliert Einhaltung des `const`-Versprechens

```
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.

Const-Referenzen

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

`const` *T*`&` `r` = *lvalue*;
r wird mit der Adresse von *lvalue* initialisiert (effizient)

`const` *T*`&` `r` = *rvalue*;
r wird mit der Adresse eines temporären Objektes vom Wert des *rvalue* initialisiert (flexibel)

Was genau ist konstant?

Betrachte L-Wert vom Typ `const` *T*

- Fall 1: *T* ist kein Referenztyp

Dann ist der L-Wert eine Konstante.

```
const int n = 5;
int& i = n; // 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
```


Wann `const` *T*`&` ?

Regel

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

Beispiele folgen später in der Vorlesung

11. Felder (Arrays) I

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

Felder: Motivation

- Wir können jetzt über Zahlen iterieren

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

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

Felder: erste Anwendung

Das Sieb des Erathostenes

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

Am Ende des Streichungsprozesses bleiben nur die Primzahlen übrig.

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

Sieb des Eratosthenes: Initialisierung

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

Konstante!

`crossed_out[i]` gibt an, ob i schon ausgestrichen wurde.

Sieb des Eratosthenes: Berechnung

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

Das Sieb: gehe zur jeweils nächsten nichtgestrichenen Zahl i (diese ist Primzahl), gib sie aus und streiche alle echten Vielfachen von i aus.

Felder: Definition

Deklaration einer Feldvariablen (array):

T a [expr]

Zugrundeliegender Typ

Konstanter ganzzahliger Ausdruck
Wert gibt die Länge des Feldes an

Variable des Feld-Typs

Typ von `a`*: „T[k]"*
Wertebereich von `a`*:* T^k

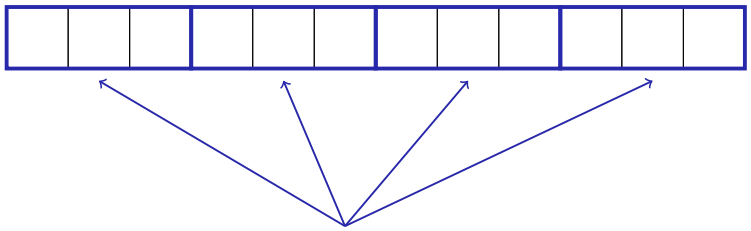
Wert ist bei Kompilierung bekannt.
Z.B. Literal, Konstante

Beispiel: `bool crossed_out[n]`

Speicherlayout eines Feldes

- Ein Feld belegt einen *zusammenhängenden* Speicherbereich

Beispiel: ein Feld mit 4 Elementen



Speicherzellen für jeweils einen Wert vom Typ `T`

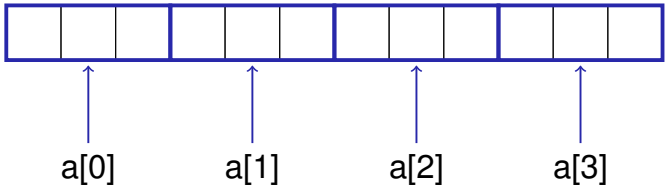
Wahlfreier Zugriff (Random Access)

Der L-Wert

Wert i

a [expr]

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



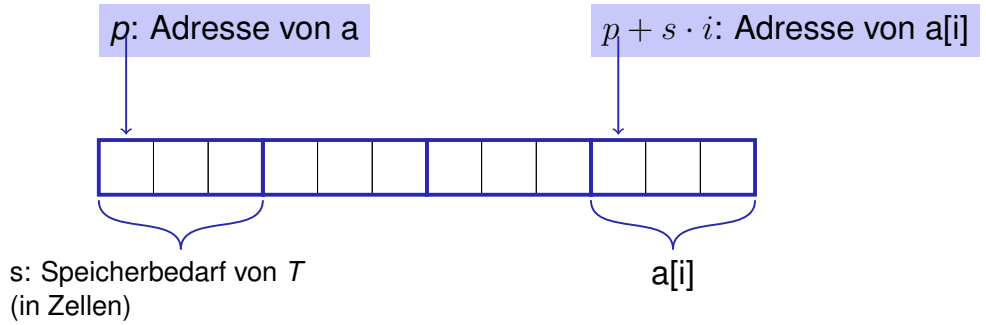
Wahlfreier Zugriff (Random Access)

a [*expr*]

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

Wahlfreier Zugriff (Random Access)

- Wahlfreier Zugriff ist sehr effizient:



Feld-Initialisierung

- `int a[5];`

  Die 5 Elemente von `a` bleiben uninitialisiert (können später Werte zugewiesen bekommen)
- `int a[5] = {4, 3, 5, 2, 1};`

  Die 5 Elemente von `a` werden mit einer *Initialisierungsliste* initialisiert.
- `int a[] = {4, 3, 5, 2, 1};`

Auch ok: Länge wird vom Compiler deduziert

Felder sind primitiv

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

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

Felder sind primitiv

Prüfung der Feldgrenzen

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

Felder sind primitiv (II)

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

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

Warum?

Felder sind primitiv

- Felder sind „Erblast” der Sprache C und aus heutiger Sicht primitiv.
- In C sind Felder sehr maschinennah und effizient, bieten aber keinen „Luxus” wie eingebautes Initialisieren und Kopieren.
- Fehlendes Prüfen der Feldgrenzen hat weitreichende Konsequenzen. Code mit nicht erlaubten aber möglichen Index-Zugriffen wurde von Schadsoftware schon (viel zu) oft ausgenutzt.
- Die Standard-Bibliothek bietet komfortable Alternativen

Vektoren

- Offensichtlicher Nachteil statischer Felder: *konstante Feldlänge*

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

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

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

Initialisierung mit n Elementen Initialwert `false`.

Elementtyp, in spitzen Klammern

Sieb des Eratosthenes mit Vektoren

```
#include <iostream>
#include <vector> // standard containers with array 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;
```


Zeichen und Texte

- Texte haben wir schon gesehen:

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

  String-Literal

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

Zeichen:	Wert des fundamentalen Typs `char`
Text:	Feld mit zugrundeliegendem Typ `char`

Der Typ `char` (“character”)

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

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

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

Literal vom Typ `char`

Der Typ `char` (“character”)

ist formal ein ganzzahliger Typ

- Werte konvertierbar nach `int` / `unsigned int`
- 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

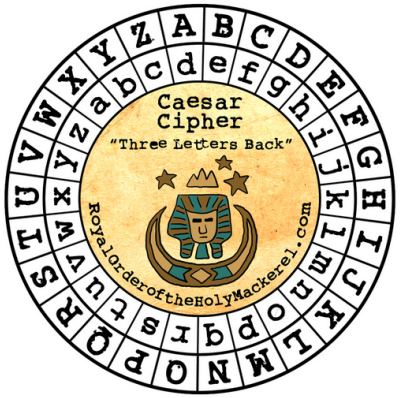
Symbol	Codierung (jeweils 16 Bit)
☠	11100010 10011000 10100000
☃	11100010 10011000 10000011
☔	11100010 10001101 10101000
☹	11100010 10011000 10011001
㋀	11100011 10000000 10100000
ﷺ	11101111 10101111 10111001

http://t-a-w.blogspot.ch/2008/12/funny-characters-in-unicode.html

Caesar-Code

Ersetze jedes druckbare Zeichen in einem Text durch seinen Vor-Vor-Vorgänger.

' '	(32)	$\rightarrow$	'|'	(124)
'!'	(33)	$\rightarrow$	'}'	(125)
		...		
'D'	(68)	$\rightarrow$	'A'	(65)
'E'	(69)	$\rightarrow$	'B'	(66)
		...		
$\sim$	(126)	$\rightarrow$	'{'	(123)


  ```

Caesar-Code: Hauptprogramm

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

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

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

Leerzeichen und Zeilenumbrüche sollen *nicht* ignoriert werden

Caesar-Code: Hauptprogramm

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

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

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

Verschiebt nur druckbare Zeichen.

Caesar-Code: shift-Funktion

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

Call by reference!

Überlauf – 95 zurück!

Unterlauf – 95 vorwärts!

Normale Verschiebung

./caesar_encrypt < power8.cpp

```
„|Moldo^j7!mltbo5+'mm
„|0^fpb|^|krj_bo|ql|qeb|bfdeqe|mltbo+

 fk'irab|9flpqob^j;|

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

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

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

Program = Moldo^j

Caesar-Code: Entschlüsselung

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

Jetzt: Verschiebung um 3 nach *rechts*

Interessante Art, `power8.cpp` auszugeben:

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

12. Felder (Arrays) II

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

Strings als Felder

- sind repräsentierbar mit zugrundeliegendem Typ `char`

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

- können auch durch String-Literale definiert werden

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

- können nur mit konstanter Grösse definiert werden

Texte

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

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

definiert einen String der Länge 4

- Ein String ist im Prinzip ein Feld mit zugrundeliegendem Typ `char`, plus Zusatzfunktionalität
- Benutzung benötigt `#include <string>`

Strings: gepimpte `char`-Felder

Ein `std::string`...

- kennt seine Länge

```
text.length()
```

gibt Länge als `int` zurück (Aufruf einer Mitglieds-Funktion; später in der Vorlesung)

- kann mit variabler Länge initialisiert werden

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

`text` wird mit n `'a'`s gefüllt

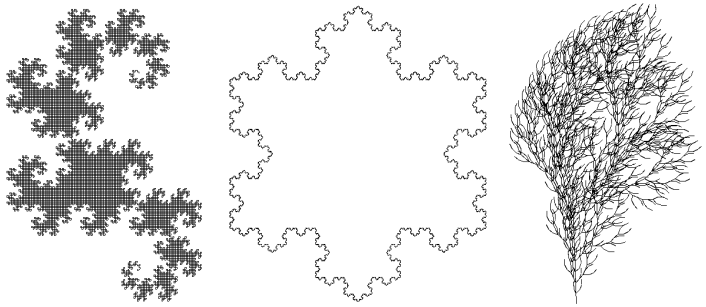
- „versteht" Vergleiche

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

`true` wenn `text1` und `text2` übereinstimmen

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

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

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

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

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

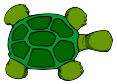
$\vdots$ $\quad \vdots$

Definition

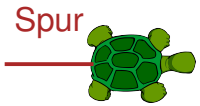
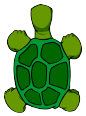
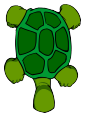
$P(c_1 c_2 \ldots c_n) := P(c_1)P(c_2)\ldots P(c_n)$

Turtle-Grafik

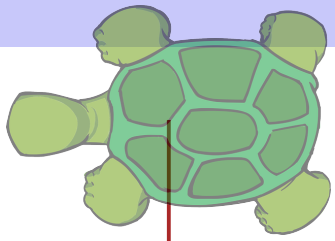
Schildkröte mit Position und Richtung



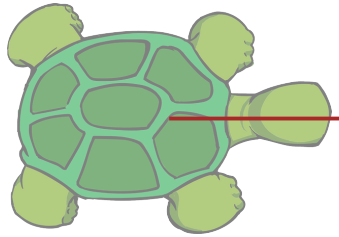
Schildkröte versteht 3 Befehle:

F: Gehe einen Schritt vorwärts ✓ 	+: Drehe dich um 90 Grad ✓ 	−: Drehe dich um −90 Grad ✓ 

Wörter zeichnen!



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



lindenmayer.cpp: Hauptprogramm

Wörter $w_0, w_1, w_2, \ldots w_n \in \Sigma^*$: `std::string`

```
...
#include "turtle.h"
...
std::cout << "Number of iterations =? ";
unsigned int n;
std::cin >> n;

std::string w = "F";                    // w = w_0 = F

for (unsigned int i = 0; i < n; ++i)
  w = next_word (w);                    // w = w_i → w = w_{i+1}

draw_word (w);                          // Zeichne w = w_n!
```


lindenmayer.cpp: next_word

```
// POST: replaces all symbols in word according to their
//       production and returns the result
std::string next_word (std::string word) {
  std::string next;
  for (unsigned int k = 0; k < word.length(); ++k)
    next += production (word[k]);
  return next;
}

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

lindenmayer.cpp: draw_word

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

Springe zum case, der word[k] entspricht.

Vorwärts! (Funktion aus unserer Schildkröten-Bibliothek)

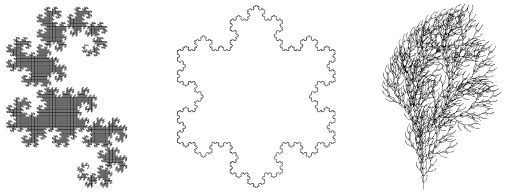
Überspringe die folgenden cases

Drehe dich um 90 Grad! (Funktion aus unserer Schildkröten-Bibliothek)

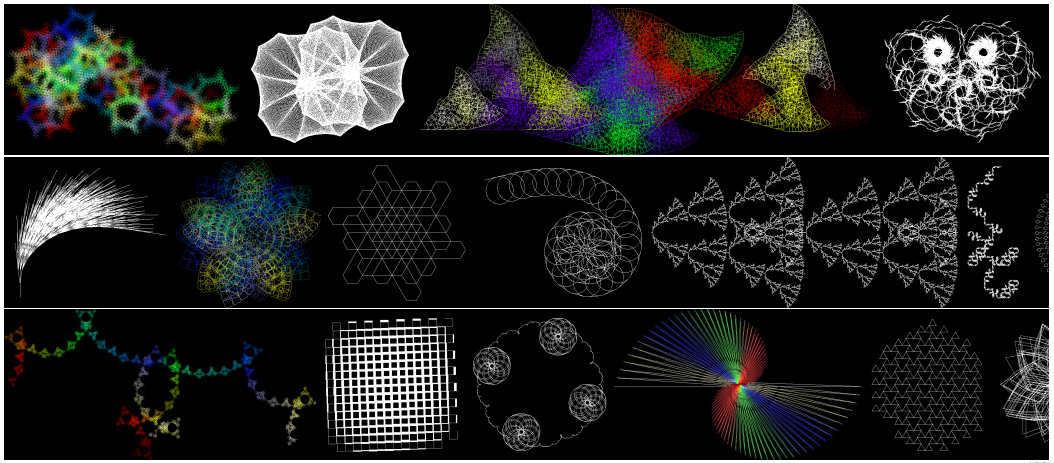
Drehe dich um -90 Grad! (Funktion aus unserer Schildkröten-Bibliothek)

L-Systeme: Erweiterungen

- Beliebige Symbole ohne grafische Interpretation (dragon.cpp)
- Beliebige Drehwinkel (snowflake.cpp)
- Sichern und Wiederherstellen des Schildkröten-Zustandes → Pflanzen (bush.cpp)



L-System-Challenge: amazing.cpp!



Mehrdimensionale Felder

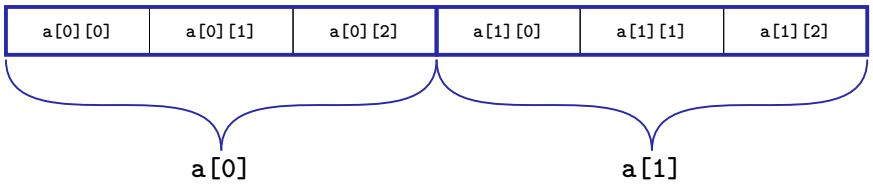
- sind Felder von Feldern
- dienen zum Speichern von *Tabellen, Matrizen,...*

```
int a[2][3]
```

a hat zwei Elemente, und jedes von ihnen ist ein Feld der Länge 3 mit zugrundeliegendem Typ int

Mehrdimensionale Felder

Im Speicher: flach



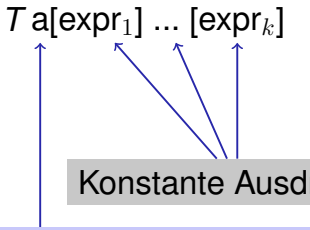
Im Kopf: Matrix

Spalten →

Zeilen ↓	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]

Mehrdimensionale Felder

- sind Felder von Feldern von Feldern ....



a hat $expr_1$ Elemente und jedes von ihnen ist ein Feld mit $expr_2$ Elementen, von denen jedes ein Feld mit $expr_3$ Elementen ist, ...

Mehrdimensionale Felder

Initialisierung:

```
int a[][3] =
    {
        {2,4,6},{1,3,5}
    }
```

Erste Dimension kann weggelassen werden

2	4	6	1	3	5

Vektoren von Vektoren

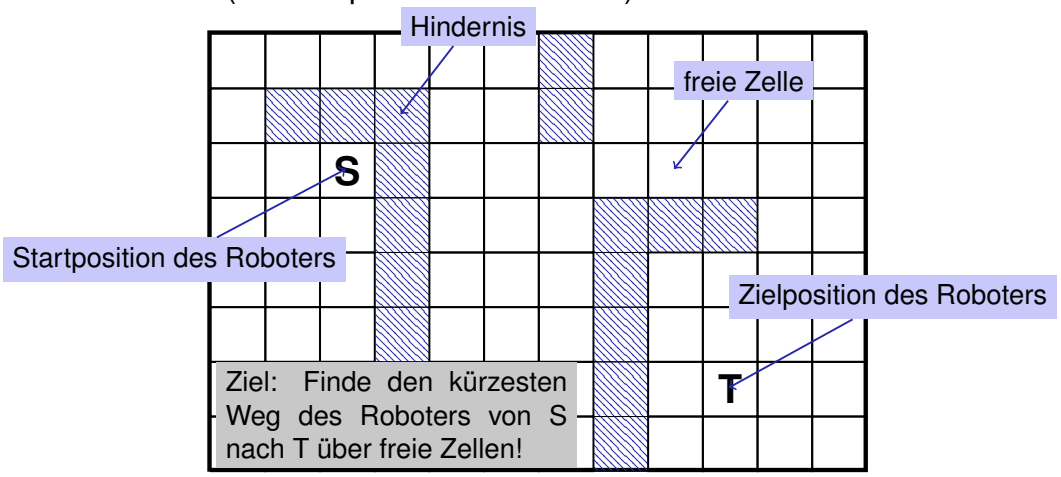
- Wie bekommen wir mehrdimensionale Felder mit variablen Dimensionen?
- Lösung: Vektoren von Vektoren

Beispiel: Vektor der Länge n von Vektoren der Länge m:

```
std::vector<std::vector<int> > a (n,
                    std::vector<int>(m));
```

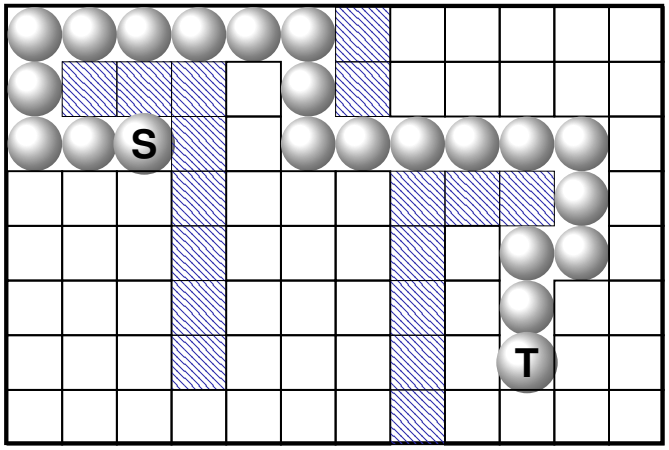
Anwendung: Kürzeste Wege

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



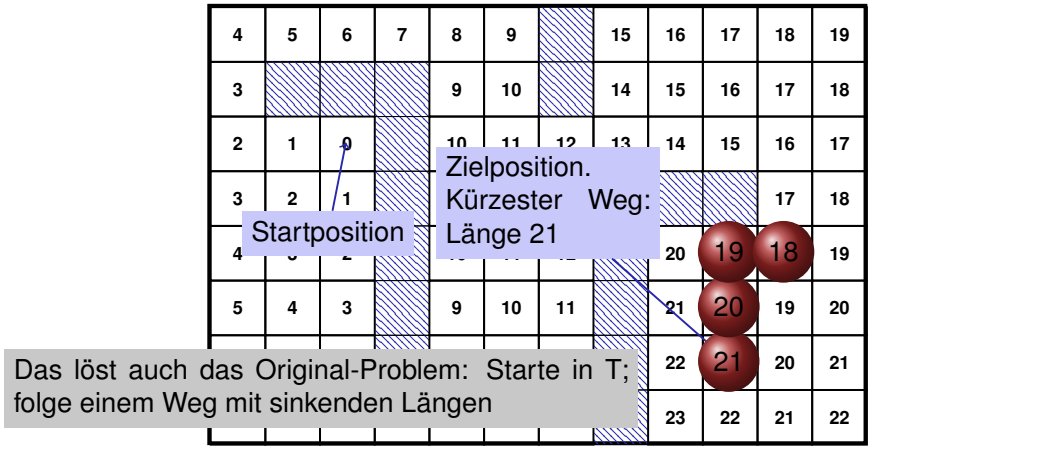
Anwendung: Kürzeste Wege

Lösung



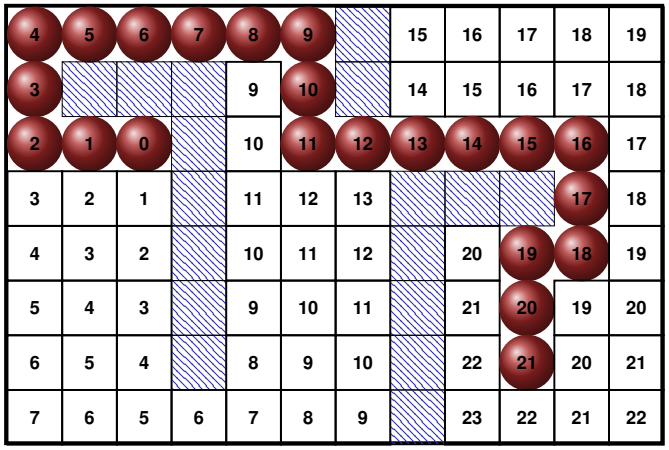
Ein (scheinbar) anderes Problem

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

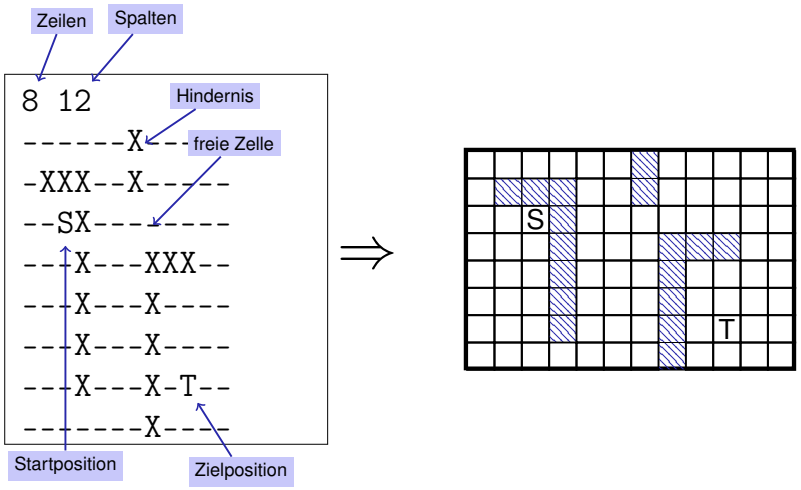


Ein (scheinbar) anderes Problem

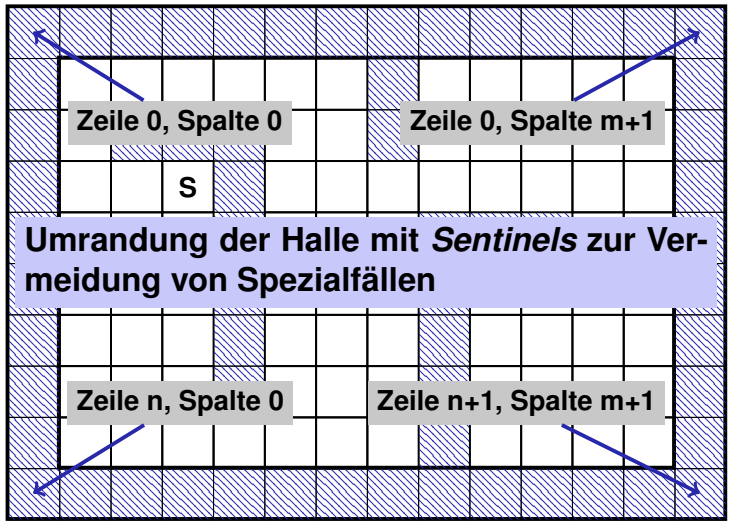
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



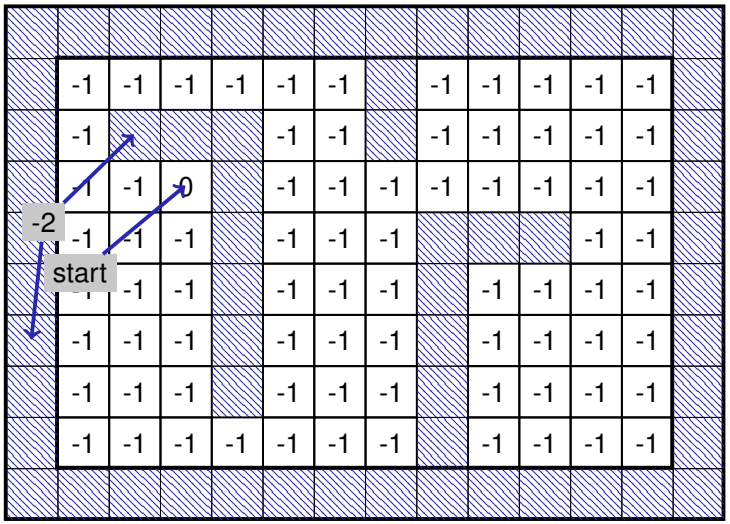
Vorbereitung: Eingabeformat



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));
```

Wächter (Sentinel)

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

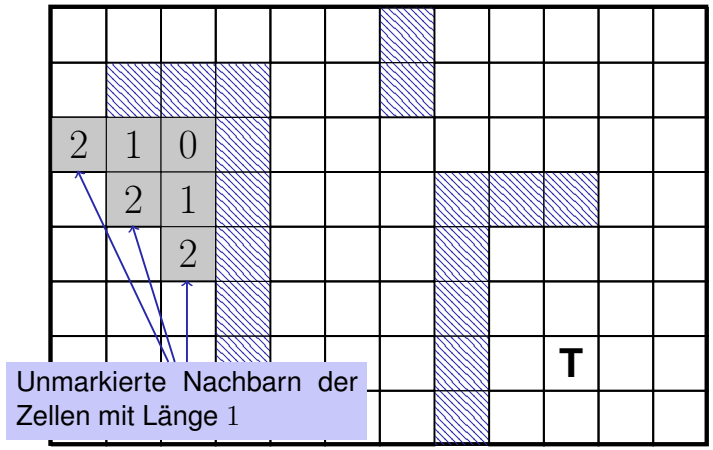
- Hinzufügen der umschliessenden „Wände"

```
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

	-3	-3	-3	-3	-3	-3		15	16	17	18	19	
	-3				9	-3		14	15	16	17	18	
	-3	-3	0		10	-3	-3	-3	-3	-3	-3	17	
	3	2	1		11	12	13				-3	18	
	4	3	2		10	11	12		20	-3	-3	19	
	5	4	3		9	10	11		21	-3	19	20	
	6	5	4		8	9	10		22	-3	20	21	
	7	6	5	6	7	8	9		23	22	21	22	

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";
}
```

$\Rightarrow$

```
oooooX-----
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 $i - 1$

Felder als Funktionsargumente

Felder können auch als *Referenz*-Argumente an eine Funktion übergeben werden. (Hier const, weil nur Lesezugriff nötig).

```
void print_vector(const int (&v)[3]) {
  for (int i = 0; i<3 ; ++i) {
    std::cout << v[i] << " ";
  }
}
```


Felder als Funktionsargumente

Das geht auch für mehrdimensionale Felder.

```
void print_matrix(const int (&m)[3][3]) {
  for (int i = 0; i<3 ; ++i) {
    print_vector (m[i]);
    std::cout << "\n";
  }
}
```


Vektoren als Funktionsargumente

Vektoren können *per value*, aber auch als *Referenz*-Argumente an eine Funktion übergeben werden.

```
void print_vector(const std::vector<int>& v) {
  for (int i = 0; i<v.size() ; ++i) {
    std::cout << v[i] << " ";
  }
}
```

Hier: *Call by Reference* ist effizienter, weil der Vektor sehr lang sein kann.

Vektoren als Funktionsargumente

Das geht auch für mehrdimensionale Vektoren.

```
void print_matrix(const std::vector<std::vector<int> >& m) {
  for (int i = 0; i<m.size() ; ++i) {
    print_vector (m[i]);
    std::cout << "\n";
  }
}
```

13. Zeiger, Algorithmen, Iteratoren und Container I

Zeiger, Address- und Dereferenoperator,
Feld-nach-Zeiger-Konversion

Komische Dinge...

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

int main(){
  int a[] = {3, 2, 1, 5, 4, 6, 7};

  // gib das kleinste Element in a aus
  std::cout << *std::min_element (a, a + 7);

  return 0;
}
```

??? ???

Dafür müssen wir zuerst *Zeiger* verstehen!

Referenzen: Wo ist Anakin?

"Suche nach Vader, und Anakin finden du wirst."

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

// anakin_skywalker = 22
```

Zeiger: Wo ist Anakin?

"Anakins Adresse ist `0x7fff6bdd1b54`."

```
int anakin_skywalker = 9;
int* here = &anakin_skywalker;
std::cout << here; // Adresse
*here =  22;

// anakin_skywalker = 22
```

Swap mit Zeigern

```
void swap(int* x, int* y){
  int t = *x;
  *x = *y;
  *y = t;
}

...
int a = 2;
int b = 1;
swap(&a, &b);
std::cout << "a= " << a << "\n"; // 1
std::cout << "b = " << b << "\n"; // 2
```


Zeiger Typen

T* Zeiger-Typ zum zugrunde liegenden Typ T.

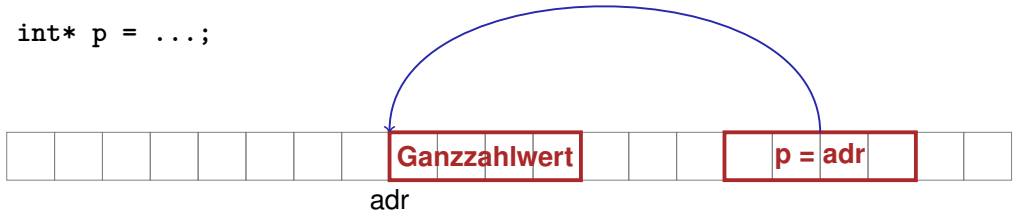
Ein Ausdruck vom Typ T* heisst *Zeiger* (auf T).

Zeiger Typen

Wert eines Zeigers auf T ist die Adresse eines Objektes vom Typ T.

Beispiele

int* p; Variable p ist Zeiger auf ein int.
float* q; Variable q ist Zeiger auf ein float.



Adress-Operator

Der Ausdruck

L-Wert vom Typ T

↓

& lval

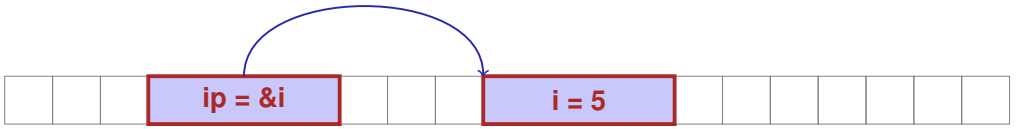
liefert als R-Wert einen *Zeiger* vom Typ *T** auf das Objekt an der Adresse von *lval*

Der Operator *&* heisst Adress-Operator.

Adress-Operator

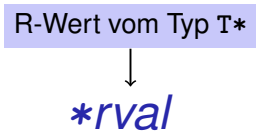
Beispiel

```
int i = 5;
int* ip = &i;  // ip initialisiert
               // mit Adresse von i.
```



Dereferenz-Operator

Der Ausdruck



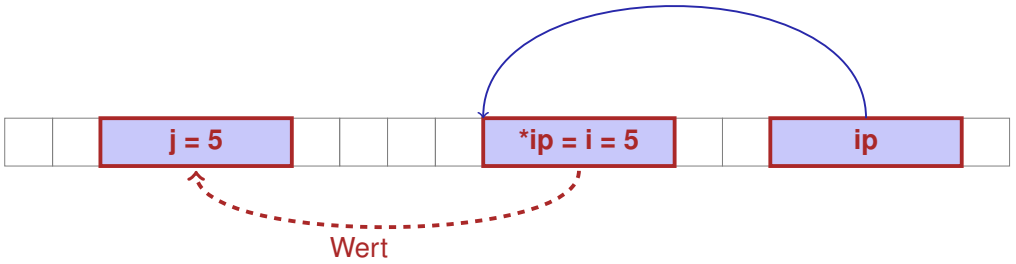
liefert als L-Wert den *Wert* des Objekts an der durch *rval* repräsentierten Adresse

Der Operator $*$ heisst Dereferenz-Operator.

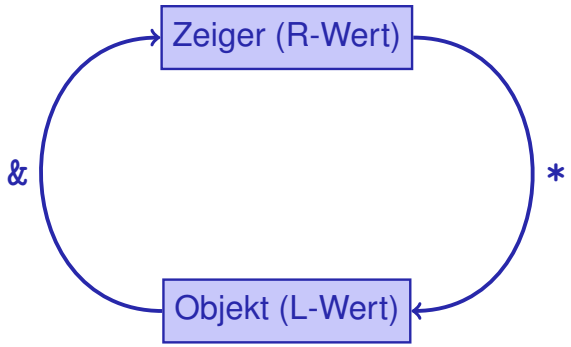
Dereferenz-Operator

Beispiel

```
int i = 5;
int* ip = &i;  // ip initialisiert
               // mit Adresse von i.
int j = *ip;   // j == 5
```



Adress- und Dereferenzoperator



Zeiger-Typen

Man zeigt nicht mit einem `double*` auf einen `int`!

Beispiele

```
int* i = ...;    // an Adresse i "wohnt" ein int...
double* j = i;   //...und an j ein double: Fehler!
```


Eselsbrücke

Die Deklaration

```
T* p;        p ist vom Typ "Zeiger auf T"
```

kann gelesen werden als

```
T *p;        *p ist vom Typ T
```

Obwohl das legal ist, schreiben wir es nicht so!

Zeiger-Arithmetik: Zeiger plus `int`

- *ptr*: Zeiger auf Element $a[k]$ des Arrays a mit Länge n
- Wert von *expr*: ganze Zahl i mit $0 \leq k + i \leq n$

ptr + expr

ist Zeiger auf $a[k + i]$.

Für $k + i = n$ erhalten wir einen *past-the-end*-Zeiger, der nicht dereferenziert werden darf.

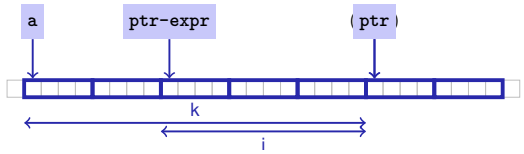
Zeiger-Arithmetik: Zeiger minus `int`

- 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 $k - i$.



Konversion Feld $\Rightarrow$ Zeiger

Wie bekommen wir einen Zeiger auf das erste Element eines Feldes?

- Statisches Feld vom Typ $T[\mathrm{n}]$ ist konvertierbar nach $T*$

Beispiel

```
int a[5];
int* begin = a; // begin zeigt auf a[0]
```

- Längeninformation geht verloren („Felder sind primitiv").

Iteration über ein Feld mit Zeigern

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

- `a+5` ist ein Zeiger direkt hinter das Ende des Feldes (past-the-end), der nicht dereferenziert werden darf.
- Zeigervergleich (`p < a+5`) bezieht sich auf die Reihenfolge der beiden Adressen im Speicher.

14. Zeiger, Algorithmen, Iteratoren und Container II

Iteration mit Zeigern, Felder: Indizes vs. Zeiger, Felder und Funktionen, Zeiger und const, Algorithmen, Container und Traversierung, Vektor-Iteratoren, Typedef, Mengen, das Iterator-Konzept

Zur Erinnerung: Mit Zeigern übers Feld

Beispiel

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

- Ein Feld ist in einen Zeiger auf sein erstes Element konvertierbar.
- Zeiger kennen Arithmetik und Vergleiche.
- Zeiger können dereferenziert werden.

$\Rightarrow$ Mit Zeigern kann man auf Feldern operieren.

Felder: Indizes vs. Zeiger

```
int a[n];

// Aufgabe: setze alle Elemente auf 0

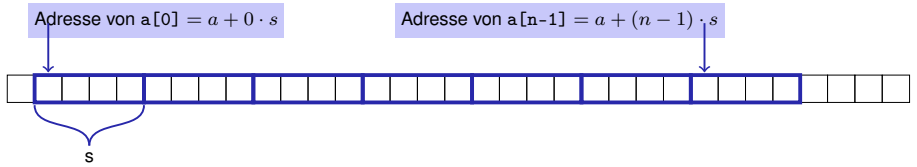
// Lösung mit Indizes ist lesbarer
for (int i = 0; i < n; ++i)
    a[i] = 0;

// Lösung mit Zeigern ist schneller und allgemeiner
int* begin = a;  // Zeiger aufs erste Element
int* end = a+n;  // Zeiger hinter das letzte Element
for (int* p = begin; p != end; ++p)
   *p = 0;
```


Felder und Indizes

```
// Setze alle Elemente auf value
for (int i = 0; i < n; ++i)
   a[i] = value;
```

Berechnungsaufwand



$\Rightarrow$ Eine **Addition** und eine **Multiplikation** pro Element

Die Wahrheit über wahlfreien Zugriff

Der Ausdruck

```
a[i]
```

ist äquivalent zu

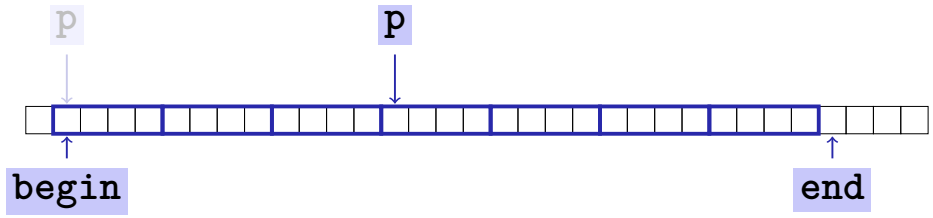
```
*(a + i)
```

$$a + i \cdot s$$

Felder und Zeiger

```
// Setze alle Elemente auf value
for (int* p = begin; p != end; ++p)
   *p = value;
```

Berechnungsaufwand



$\Rightarrow$ eine **Addition** pro Element

Ein Buch lesen . . . mit Indizes . . . mit Zeigern

Wahlfreier Zugriff

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

Sequentieller Zugriff

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

Feldargumente: *Call by (const) reference*

```
void print_vector (const int (&v)[3]) {
  for (int i = 0; i<3 ; ++i) {
    std::cout << v[i] << " ";
  }
}
void make_null_vector (int (&v)[3]) {
  for (int i = 0; i<3 ; ++i) {
    v[i] = 0;
  }
}
```


Feldargumente: *Call by value (nicht wirklich...)*

```
void make_null_vector (int v[3]) {
  for (int i = 0; i<3 ; ++i) {
    v[i] = 0;
  }
}
...
int a[10];
make_null_vector (a); // setzt nur a[0], a[1], a[2]

int* b;
make_null_vector (b); // kein Feld bei b, Crash!
```


Feldargumente: *Call by value* gibt's nicht

- Formale Argumenttypen T`[n]` oder T`[]` (Feld über `T`) sind äquivalent zu $T*$ (Zeiger auf `T`)
- Bei der Übergabe eines Feldes wird ein Zeiger auf das erste Element übergeben
- Längeninformation geht verloren
- Funktion kann keinen Feldausschnitt verarbeiten (Beispiel: Suche eines Elements nur im hinteren Teil des Feldes)

Felder in Funktionen

Konvention der Standard-Bibliothek: Übergabe eines Feldes (oder eines Feldausschnitts) mit zwei Zeigern:

- `begin`: Zeiger auf das erste Element
- `end`: Zeiger *hinter* das letzte Element
- `[begin, end)` bezeichnet die Elemente des Feldausschnitts
- *Gültiger* Bereich heisst: hier "leben" wirklich Feldelemente
- `[begin, end)` ist leer, wenn `begin == end`

Felder in Funktionen: `fill`

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

int a[5];
fill (a, a+5, 1);
for (int i=0; i<5; ++i)
  std::cout << a[i] << " ";
```

Erwartet Zeiger auf das erste Element eines Bereichs

Übergabe der Adresse (des ersten Elements) von `a`

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
```


Null-Zeiger

- spezieller Zeigerwert, der angibt, dass noch auf kein Objekt gezeigt wird
- repräsentiert durch die ganze Zahl 0 (konvertierbar nach `T*`)

```
int* iptr = 0;
```

- kann nicht dereferenziert werden (prüfbar zur Laufzeit)
- zur Vermeidung undefinierten Verhaltens

```
int* iptr; // iptr points into ''nirvana''
int j = *iptr; // illegal address in *
```

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 Feld zeigen.

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

Zeigeroperatoren

Beschreibung	Op	Stellig-keit	Prä-zedenz	Assoziativi-tät	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

Mutierende Funktionen

- Zeiger können (wie auch Referenzen) für Funktionen mit Effekt verwendet werden.

Beispiel

```
int a[5];
fill(a, a+5, 1); // verändert a
```

Übergabe der Adresse des Elements hinter `a`

Übergabe der Adresse (des ersten Elements) von `a`

- Solche Funktionen heissen *mutierend*

Const-Korrektheit

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

```
// PRE: [begin , end) is a valid and nonempty range
// POST: the smallest value in [begin, end) is returned
int min (const int* begin ,const int* end)
{
    assert (begin != end);
    int m = *begin; // current minimum candidate
    for (const int* p = ++begin; p != end; ++p)
        if (*p < m) m = *p;
    return m;
}
```

- Kennzeichnung mit `const`: Objekte können durch solche `const`-Zeiger nicht im Wert verändert werden.

Const und Zeiger

Zur Determinierung der Zugehörigkeit des `const` Modifiers:

const T ist äquivalent zu *T const* und kann auch so geschrieben werden

```
const int a;     ⟺   int const a;
const int* a;    ⟺   int const *a;
```

Lies Deklaration von rechts nach links

```
int const a;              a ist eine konstante Ganzzahl
int const* a;             a ist ein Zeiger auf eine konstante Ganzzahl
int* const a;             a ist ein konstanter Zeiger auf eine Ganzzahl
int const* const a;       a ist ein konstanter Zeiger auf eine konstante Ganz-
                          zahl
```


`const` ist nicht absolut

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

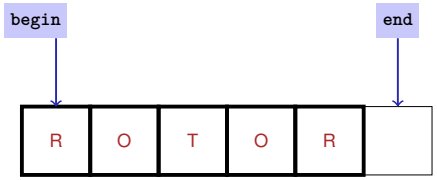
beispiel

```
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;
}
```



Algorithmen

Für viele alltägliche Probleme existieren vorgefertigte Lösungen in der Standardbibliothek.

Beispiel: Füllen eines Feldes

```
#include <algorithm> // needed for std::fill
...

int a[5];
std::fill (a, a+5, 1);

for (int i=0; i<5; ++i)
    std::cout << a[i] << " "; // 1 1 1 1 1
```

Algorithmen

Vorteile der Verwendung der Standardbibliothek

- Einfachere Programme
- Weniger Fehlerquellen
- Guter, schneller Code
- Code unabhängig vom Datentyp (nächste Folie)
- Es existieren natürlich auch Algorithmen für schwierigere Probleme, wie z.B. das (effiziente) Sortieren eines Feldes

Algorithmen

Die gleichen vorgefertigten Algorithmen funktionieren für viele verschiedene Datentypen.

Beispiel: Füllen eines Feldes

```
#include <algorithm> // needed for std::fill
...

char c[3];
std::fill (c, c+3, '!');

for (int i=0; i<3; ++i)
    std::cout << c[i]; // !!!
```


Exkurs: Templates

- Templates erlauben die Angabe eines Typs als Argument
- Der Compiler deduziert den passenden Typ aus den Aufrufargumenten

Beispiel: `fill` mit Templates

```
template <typename T>
void fill (T* begin, T* end, T value) {
    for (T* p = begin; p != end; ++p)
        *p = value;
}
int a[5];
fill (a, a+5, 1);   // 1 1 1 1 1

char c[3];
fill (c, c+3, '!'); // !!!
```

Die eckigen Klammern kennen wir schon von `std::vector<int>`. Vektoren sind auch als Templates realisiert.

Auch `std::fill` ist als Template realisiert!

Container und Traversierung

- **Container:** Behälter (Feld, Vektor,...) für Elemente
- **Traversierung:** Durchlaufen eines Containers
  - Initialisierung der Elemente (`fill`)
  - Suchen des kleinsten Elements (`min`)
  - Prüfen von Eigenschaften (`is_palindrome`)
  - ...
- Es gibt noch viele andere Container (Mengen, Listen,...)

Werkzeuge zur Traversierung

- Felder: Indizes (wahlfrei) oder Zeiger (natürlich)
- Feld-Algorithmen (`std::`) benutzen Zeiger

```
int a[5];
std::fill (a, a+5, 1);   // 1 1 1 1 1
```

- Wie traversiert man Vektoren und andere Container?

```
std::vector<int> v (5, 0); // 0 0 0 0 0
std::fill (?, ?, 1);   // 1 1 1 1 1
```


Vektoren: *too sexy for pointers*

- Unser `fill` mit Templates funktioniert für Vektoren nicht...
- ...und `std::fill` so auch nicht:

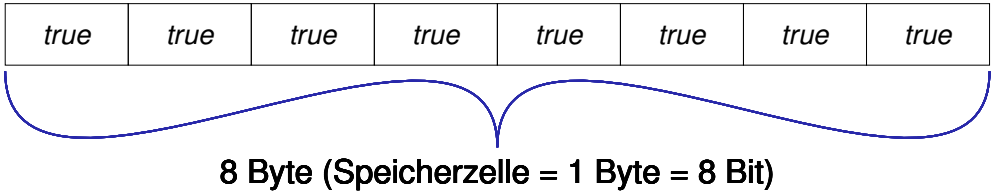
```
std::vector<int> v (5, 0);
std::fill (v, v+5, 1);   // Fehlermeldung des Compilers !
```

Vektoren sind was Besseres...

- Sie lassen sich weder in Zeiger konvertieren,...
- ...noch mit Zeigern traversieren.
- Das ist ihnen viel zu primitiv.☺

Auch im Speicher: Vektor $\neq$ Feld

```
bool a[8] = {true, true, true, true, true, true, true, true};
```




```
std::vector<bool> v (8, true);
```

`0b11111111` 1 Byte

`bool*`-Zeiger passt hier nicht, denn er läuft byteweise, nicht bitweise!

Vektor-Iteratoren

Iterator: ein "Zeiger", der zum Container passt.

Beispiel: Füllen eines Vektors mit `std::fill` – so geht's!

```
#include <vector>
#include <algorithm> // needed for std::fill

...
std::vector<int> v(5, 0);
std::fill (v.begin(), v.end(), 1);
for (int i=0; i<5; ++i)
    std::cout << v[i] << " "; // 1 1 1 1 1
```

Vektor-Iteratoren

Für jeden Vektor sind zwei *Iterator-Typen* definiert.

- `std::vector<int>::const_iterator`
  - für nicht-mutierenden Zugriff
  - analog zu `const int*` für Felder
- `std::vector<int>::iterator`
  - für mutierenden Zugriff
  - analog zu `int*` für Felder
- Ein Vektor-Iterator `it` ist kein Zeiger, verhält sich aber so:
  - zeigt auf ein Vektor-Element und kann dereferenziert werden (`*it`)
  - kennt Arithmetik und Vergleiche (`++it`, `it+2`, `it < end`,...)

Vektor-Iteratoren: `begin()` und `end()`

- `v.begin()` zeigt auf das erste Element von `v`
- `v.end()` zeigt hinter das letzte Element von `v`
- Damit können wir einen Vektor traversieren...

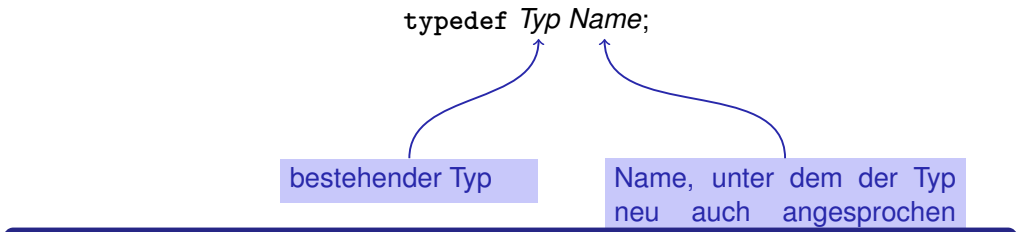
```
for (std::vector<int>::const_iterator it = v.begin();
                                      it != v.end(); ++it)
    std::cout << *it << " ";
```

- ...oder einen Vektor füllen.

```
std::fill (v.begin(), v.end(), 1);
```


Typnamen in C++ können laaaaaaaang werden

- `std::vector<int>::const_iterator`
- Dann hilft die Deklaration eines *Typ-Alias* mit



Beispiele

```
typedef std::vector<int> int_vec;
typedef int_vec::const_iterator Cvit;
```


Vektor-Iteratoren funktionieren wie Zeiger

```
typedef std::vector<int>::const_iterator Cvit;

std::vector<int> v(5, 0); // 0 0 0 0 0

// output all elements of a, using iteration
for (Cvit it = v.begin(); it != v.end(); ++it)
    std::cout << *it << " ";
```

Vektor-Element, auf das `it` zeigt

Vektor-Iteratoren funktionieren wie Zeiger

```
typedef std::vector<int>::iterator Vit;

// manually set all elements to 1
for (Vit it = v.begin(); it != v.end(); ++it)
    *it = 1;

// output all elements again, using random access
for (int i=0; i<5; ++i)
    std::cout << v[i] << " ";
```

Inkrementieren des Iterators

Kurzschreibweise für `*(v.begin()+i)`

Andere Container: Mengen (Sets)

- Eine Menge ist eine ungeordnete Zusammenfassung von Elementen, wobei jedes Element nur einmal vorkommt.

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- C++: `std::set<T>` für eine Menge mit Elementen vom Typ `T`

Mengen: Beispiel einer Anwendung

- Stelle fest, ob ein gegebener Text ein Fragezeichen enthält und gib alle im Text vorkommenden *verschiedenen* Zeichen aus!

Buchstabensalat (1)

Fasse den Text als Menge von Buchstaben auf:

```
#include<set>
...
typedef std::set<char>::const_iterator Csit;
...
std::string text =
"What are the distinct characters in this string?";

std::set<char> s (text.begin(),text.end());
```

Menge wird mit *String-Iterator-Bereich* `[text.begin(), text.end())` initialisiert

Buchstabensalat (2)

Stelle fest, ob der Text ein Fragezeichen enthält und gib alle im Text enthaltenen Buchstaben aus

Suchalgorithmus, aufrufbar mit beliebigem Iterator-Bereich

```
// check whether text contains a question mark
if (std::find (s.begin(), s.end(), '?') != s.end())
    std::cout << "Good question!\n";

// output all distinct characters
for (Csit it = s.begin(); it != s.end(); ++it)
    std::cout << *it;
```

Ausgabe:
```
Good question!
?Wacdeghinrst
```


Mengen und Indizes?

- Kann man Mengen mit wahlfreiem Zugriff traversieren? Nein.

```
for (int i=0; i<s.size(); ++i)
    std::cout << s[i];
```

Fehlermeldung: no subscript operator

- Mengen sind ungeordnet.
  - Es gibt kein "i-tes Element".
  - Iteratorvergleich `it != s.end()` geht, nicht aber `it < s.end()`!

Das Konzept der Iteratoren

C++kennt verschiedene Iterator-Typen

- Jeder Container hat einen zugehörigen Iterator-Typ
- Alle können dereferenzieren (`*it`) und traversieren (`++it`)
- Manche können mehr, z.B. wahlfreien Zugriff (`it[k]`, oder äquivalent `*(it + k)`), rückwärts traversieren (`--it`),...

Das Konzept der Iteratoren

Jeder Container-Algorithmus der Standardbibliothek ist *generisch*.
Das heisst:

- Der Container wird per Iterator-Bereich übergeben
- Der Algorithmus funktioniert für alle Container, deren Iteratoren die Anforderungen des Algorithmus erfüllen
- `std::find` erfordert z.B. nur `*` und `++`
- Implementationsdetails des Containers sind nicht von Bedeutung

Warum Zeiger und Iteratoren?

Würde man nicht diesen Code

```
for (int i=0; i<n; ++i)
    a[i] = 0;
```

gegenüber folgendem Code bevorzugen?

```
for (int* ptr=a; ptr<a+n; ++ptr)
    *ptr = 0;
```

Vielleicht, aber um (hier noch besser!) das generische `std::fill(a, a+n, 0);` benutzen zu können, *müssen* wir mit Zeigern arbeiten.

Warum Zeiger und Iteratoren?

Zur Verwendung der Standardbibliothek muss man also wissen:

- statisches Feld `a` ist zugleich ein Zeiger auf das erste Element von `a`
- `a+i` ist ein Zeiger auf das Element mit Index i

Verwendung der Standardbibliothek mit anderen Containern: Zeiger $\Rightarrow$ Iteratoren

Warum Zeiger und Iteratoren?

Beispiel: Zum Suchen des kleinsten Elementes eines Containers im Bereich $[\text{begin}, \text{end})$ verwende den Funktionsaufruf

```
std::min_element(begin, end)
```

- Gibt einen *Iterator* auf das kleinste Element zurück.
- Zum Auslesen des kleinsten Elementes muss man noch dereferenzieren:

```
*std::min_element(begin, end)
```


Darum Zeiger und Iteratoren

- Selbst für Nichtprogrammierer und “dumme” Nur-Benutzer der Standardbibliothek: Ausdrücke der Art
  `*std::min_element(begin, end)`
  lassen sich ohne die Kenntnis von Zeigern und Iteratoren nicht verstehen.
- Hinter den Kulissen der Standardbibliothek ist das Arbeiten mit dynamischem Speicher auf Basis von Zeigern unvermeidbar. Mehr dazu später in der Vorlesung!

15. Rekursion 1

Mathematische Rekursion, Terminierung, der Aufrufstapel, Beispiele, Rekursion vs. Iteration

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)`

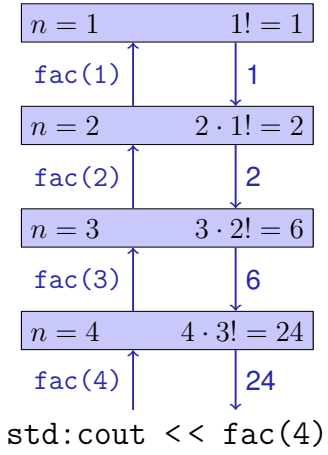
```
// POST: return value is n!
unsigned int fac (unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

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:

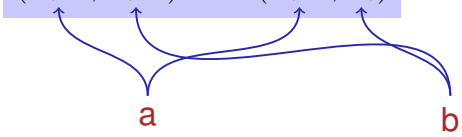
- Berechne jede Fibonacci-Zahl nur einmal, in der Reihenfolge $F_0, F_1, F_2, \ldots, F_n$!
- Merke dir jeweils die zwei letzten berechneten Zahlen (Variablen `a` und `b`)!
- Berechne die nächste Zahl als Summe von `a` und `b`!

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)`

$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$



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.

16. Rekursion 2

Bau eines Taschenrechners, Ströme, 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:

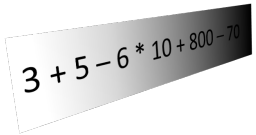
$$\underbrace{13 + 4 * (15 - 7 * 3)}_{\text{Muss gespeichert bleiben, damit jetzt ausgewertet werden kann!}} =$$

Das "Verstehen" eines Ausdrucks erfordert Vorausschau auf kommende Symbole!

Beispiel

Als Vorbereitung: Ströme

Ein Programm verarbeitet Eingaben von einem konzeptuell unbegrenzten Eingabestrom.



Bisher: Eingabestrom der Kommandozeile `std::cin`

```
while (std::cin >> op && op != '=') { ... }
```

Konsumiere op von `std::cin`, Leseposition schreitet fort.

Wir wollen zukünftig aber auch von Dateien lesen können!

Beispiel: BSD 16-bit Checksum

```
#include <iostream>


int main () {

  char c;
  int checksum = 0;
  while (std::cin >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  std::cout << "checksum = " << std::hex << checksum << "\n";
}
```

Eingabe: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Erfordert in der Konsole manuelles Ende der Eingabe [8]

Ausgabe: 67fd

[8] Ctrl-D(Unix) / Ctrl-Z(Windows) am Anfang einer Zeile, welche durch ENTER abgeschlossen wird

Beispiel: BSD 16-bit Checksum mit Datei

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

int main () {
  std::ifstream fileStream ("loremispum.txt");
  char c;
  int checksum = 0;
  while (fileStream >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  std::cout << "checksum = " << std::hex << checksum << "\n";
}
```

Ausgabe: 67fd

Gibt am Dateiende `false` zurück.

Beispiel: BSD 16-bit Checksum

Wiederverwendung gemeinsam genutzter Funktionalität?

Richtig: mit einer Funktion. Aber wie?

Beispiel: BSD 16-bit Checksum generisch!

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

int checksum (std::istream& is)
{
  char c;
  int checksum = 0;
  while (is >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  return checksum;
}
```

Referenz nötig: wir verändern den Strom!

Gleiches Recht für alle!

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

int checksum (std::istream& is) { ... }

int main () {
  std::ifstream fileStream("loremipsum.txt");

  if (checksum (fileStream) == checksum (std::cin))
       std::cout << "checksums match.\n";
  else
       std::cout << "checksums differ.\n";
}
```

Eingabe: Lorem Yps mit Gimmick
Ausgabe: checksums differ

Warum geht das ?

- `std::cin` ist eine Variable vom Typ `std::istream`. Sie repräsentiert einen Eingabestrom.
- Unsere Variable `fileStream` ist vom Typ `std::ifstream`. Sie repräsentiert einen Eingabestrom auf einer Datei.
- Ein `std::ifstream` *ist auch ein* `std::istream`, kann nur etwas mehr.
- Somit kann `fileStream` überall dort verwendet werden, wo ein `std::istream` verlangt ist.

Nochmal gleiches Recht für alle!

Eingabe aus `stringStream`
Ausgabe: checksums differ

```
#include <iostream>
#include <fstream>
#include <sstream>

int checksum (std::istream& is) { ... }

int main () {
  std::ifstream fileStream ("loremipsum.txt");
  std::stringstream stringStream ("Lorem Yps mit Gimmick");

  if (checksum (fileStream) == checksum (stringStream))
        std::cout << "checksums match.\n";
  else
        std::cout << "checksums differ.\n";
}
```


Zurück zu den Ausdrücken

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

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

Wir werden die Symbole elegant mittels Rekursion zwischenspeichern.

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

Formale Grammatiken

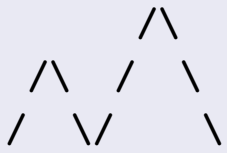
- Alphabet: endliche Menge von Symbolen Σ
- Sätze: endlichen Folgen von Symbolen Σ^*

Eine formale Grammatik definiert, welche Sätze gültig sind.

Berge

- Alphabet: $\{/, \backslash\}$
- Berge: $\mathcal{M} \subset \{/, \backslash\}^*$ (gültige Sätze)

$m' =$ //\\///\\\



Falsche Berge

- Alphabet: $\{/, \backslash\}$
- Berge: $\mathcal{M} \subset \{/, \backslash\}^*$ (gültige Sätze)

$m''' =$ /\\//\ $\notin \mathcal{M}$

Beide Seiten müssen gleiche Starthöhe haben. Ein Berg darf nicht unter seine Starthöhe fallen.

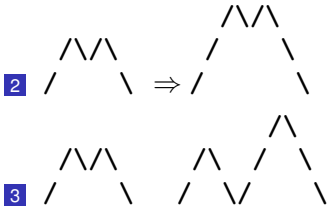
Berge in Backus-Naur-Form (BNF)

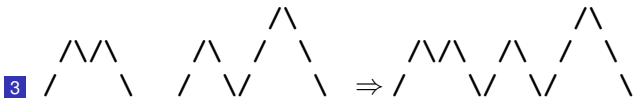
berg = "/\" | "/" berg "\" | berg berg. **Regel**

Alternativen | Nichtterminal | Terminal

Mögliche Berge

1 /\

2 

3 

Man kann beweisen, dass diese BNF “unsere” Berge beschreibt, was a priori nicht ganz klar ist.

Ausdrücke

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

Was benötigen wir in einer BNF?

- Zahl , (Ausdruck)
  -Zahl, -(Ausdruck)
  Faktor
- Faktor * Faktor, Faktor
  Faktor * Faktor / Faktor , ...
  Term
- Term + Term, Term
  Term - Term, ...
  Ausdruck

Die BNF für Ausdrücke

Ein Faktor ist

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

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


Die BNF für Ausdrücke

Ein Term ist

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

Wir brauchen Repetition!

EBNF

Extended Backus Naur Form: Erweiterung der BNF um

- Option `[]` und
- Optionale Repetition `{}`

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

NB: die EBNF ist nicht reichhaltiger als die BNF. Sie erlaubt nur eine kompaktere Schreibweise. Obiges Konstrukt lässt sich mit BNF z.B. so schreiben:

```
term = factor | factor T.
T = "∗" term | "+" term.
```


Die EBNF für Ausdrücke

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

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

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

Parsen

- **Parsen:** Feststellen, ob ein Satz nach der (E)BNF gültig ist.
- **Parser:** Programm zum Parsen
- **Praktisch:** Aus der (E)BNF kann (fast) automatisch ein Parser generiert werden:
  - Regeln werden zu Funktionen
  - Alternativen und Optionen werden zu `if`–Anweisungen
  - Nichtterminale Symbole auf der rechten Seite werden zu Funktionsaufrufen
  - Optionale Repetitionen werden zu `while`–Anweisungen

Funktionen (Parser mit Auswertung)

Ausdruck wird aus einem Eingabestrom gelesen.

```
// POST: extracts a factor from is
//       and returns its value
double factor (std::istream& is);

// POST: extracts a term from is
//       and returns its value
double term (std::istream& is);

// POST: extracts an expression from is
//       and returns its value
double expression (std::istream& is);
```


Vorausschau von einem Zeichen...

...um jeweils die richtige Alternative zu finden.

```
// POST: leading whitespace characters are extracted
//       from is, and the first non-whitespace character
//       is returned (0 if there is no such character)
char lookahead (std::istream& is)
{
  if (is.eof())
    return 0;
  is >> std::ws;          // skip whitespaces
  if (is.eof())
    return 0;             // end of stream
  return is.peek();     // next character in is
}
```


Rosinenpickerei

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

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

Faktoren auswerten

```
double factor (std::istream& is)
{
  double v;
  if (consume(is, '(')){
    v = expression (is);
    consume(is, ')');
  } else if (consume(is, '-'))
    v = -factor (is);
  else
    is >> v;
  return v;
}
```

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


Terme auswerten

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

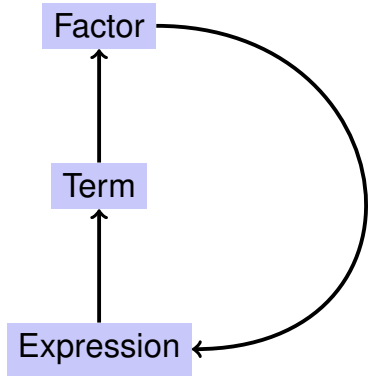
```
term = factor { "*" factor | "/" factor }
```


Ausdrücke auswerten

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

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


Rekursion!



EBNF — Und es funktioniert!

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

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

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

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

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


BNF — Und es funktioniert nicht!

BNF (`calculator_r.cpp`, Auswertung von rechts nach links):

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

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

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

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


Analyse: Repetition vs. Rekursion

Vereinfachung: Summe / Differenz von Zahlen

Beispiele

$$3, \quad 3-5, \quad 3-7-1$$

EBNF:

```
sum = value {"-" value | "+" value}.
```

BNF:

```
sum = value | value "-" sum | value "+" sum.
```

Die beiden Gramatiken erlauben dieselben Ausdrücke.

`value`

```
double value (std::istream& is){
    double val;
    is >> val;
    return val;
}
```

EBNF Variante

```
// sum = value {"-" value | "+" value}.
double sum(std::istream& is) {
    double v = value(is);
    while(true){
        if (consume(is, '-'))
            v -= value(is);
        else if (consume(is, '+'))
            v += value(is);
        else
            return v;
    }
}
```


Wir testen: EBNF Variante

- Eingabe: 1-2
  Ausgabe: -1 ✓
- Eingabe: 1-2-3
  Ausgabe: -4 ✓

BNF Variante

```
// sum = value | value "-" sum | value "+" sum.
double sum(std::istream& is){
    double v = value(is);
    if (consume(is, '-'))
        return v - sum(is);
    else if(consume(is, '+'))
        return v + sum(is);
    return v;
}
```


Wir testen: BNF Variante

- Eingabe: 1-2
  Ausgabe: -1 ✓
- Eingabe: 1-2-3
  Ausgabe: 2 

Wir testen

 Ist die BNF falsch ?

```
sum = value
      | value "−" sum
      | value "+" sum.
```

- Nein, denn sie spricht nur über die Gültigkeit von Ausdrücken, nicht über deren Werte!
- Die Auswertung haben wir naiv “obendrauf” gesetzt.

Dem Problem auf den Grund gehen

```
double sum (std::istream& is){
   double v = value (is);
   if (consume (is,'−'))
      v -= sum (is);
   else if (consume (is,'+'))
    v += sum(is);
   return v;
}
...
std::stringstream input ("1−2−3");
std::cout << sum (input) << "\n"; // 4
```

3	3
2 - "3"	-1
1 - "2 - 3"	2
"1 - 2 - 3"	2

Was ist denn falsch gelaufen?

Die BNF

- spricht offiziell zwar nicht über Werte,
- legt uns aber trotzdem die falsche Klammerung (von rechts nach links) nahe.

```
sum = value | value "−" sum | value "+" sum.
```

führt sehr natürlich zu

$1 - 2 - 3 = 1 - (2 - 3)$

Eine Lösung: Linksrekursion

```
sum  = value | sum "−" value | sum "+" value.
```

Implementationsmuster von vorher funktioniert nicht mehr.
Linksrekursion muss wieder zu Rechtsrekursion aufgelöst werden.

Das sähe dann so aus:

```
sum  = value | value s.
s = "−" sum | "+" sum.
```

Siehe `calculator_1.cpp`

17. Structs und Klassen I

Rationale Zahlen, Struct-Definition, Funktions- und Operatorüberladung, Const-Referenzen, Datenkapselung

Rechnen mit rationalen Zahlen

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

Ziel

Wir bauen uns selbst einen 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;

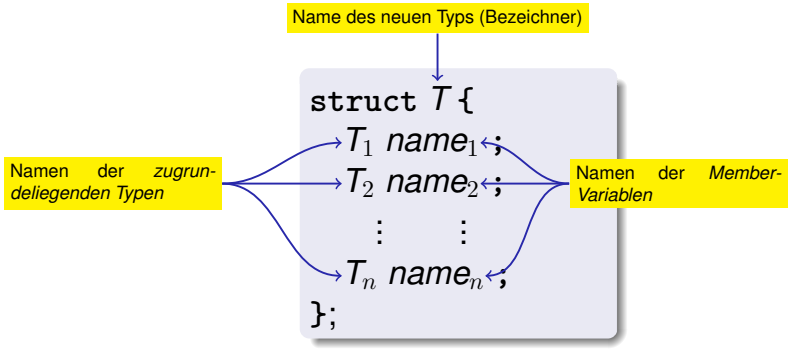
// Input s the same way
rational s;
...
```


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: $T_1 \times T_2 \times \ldots \times T_n$

Struct-Definitionen: Beispiele

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

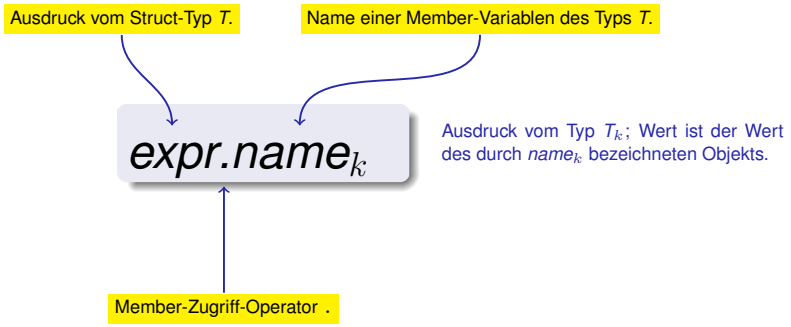
Zugrundeliegende Typen können fundamentale aber auch benutzerdefinierte Typen sein.

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



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

Zur Erinnerung: Grosse Objekte ...

```
struct SimulatedCPU {
    unsigned int pc;
    int stack[16];
    unsigned int stackPosition;
    unsigned int memory[65536];
};

void outputState (SimulatedCPU p) {
    std::cout << "pc=" << p.pc;
    std::cout << ", stack: ";
    for (unsigned int i = p.stackPosition; i != 0; --i)
        std::cout << p.stack[i-1];
}
```

Call by value: mehr als 256k werden kopiert!

... übergibt man als Const-Referenz

```
struct SimulatedCPU {
    unsigned int pc;
    int stack[16];
    unsigned int stackPosition;
    unsigned int memory[65536];
};

void outputState (const SimulatedCPU& p) {
    std::cout << "pc=" << p.pc;
    std::cout << ", stack: ";
    for (int i = p.stackPosition; i != 0; --i)
        std::cout << p.stack[i-1];
}
```

Call by reference: nur eine Adresse wird kopiert.

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

18. Klassen

Klassen, Memberfunktionen, Konstruktoren, Stapel, verkettete Liste, dynamischer Speicher, Copy-Konstruktor, Zuweisungsoperator, Destruktor, Konzept Dynamischer Datentyp

Datenkapselung: `public` / `private`

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

Gute Nachricht: `r.d = 0` aus Versehen geht nicht mehr

Schlechte Nachricht: Der Kunde kann nun gar nichts mehr machen . . .

. . . und wir auch nicht (kein `operator+,...`)

Anwendungscode:

```
rational r;
r.n = 1;     // error: n is private
r.d = 2;     // error: d is private
int i = r.n; // error: n is private
```

Memberfunktionen: Deklaration

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

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
```

Member-Zugriff

Memberfunktionen: Definition

```
// POST: returns  numerator of *this
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 `*this`, verspricht also, dass das implizite Argument nicht im Wert verändert wird.
- `n` ist Abkürzung in der Memberfunktion für `(*this).n`

This rational vs. dieser Bruch

So würde es aussehen...

```
class rational {
    int n;
    ...

    int numerator () const
    {
        return (*this).n;
    }
};

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

... ohne Memberfunktionen

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

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

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

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

```
class rational {
    int n;
    ...

    int numerator () const
    {
        return n;
    }
    ....
};
```

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

```
class rational {
    int n;
    ...

    int numerator () const;
    ...
};

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

- So geht's auch.

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

⇒ 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 (Handout).

Fix: “Unser” Typ `rational::integer`

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

```
public:
   typedef int integer; // 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` → `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

`rational.h`

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

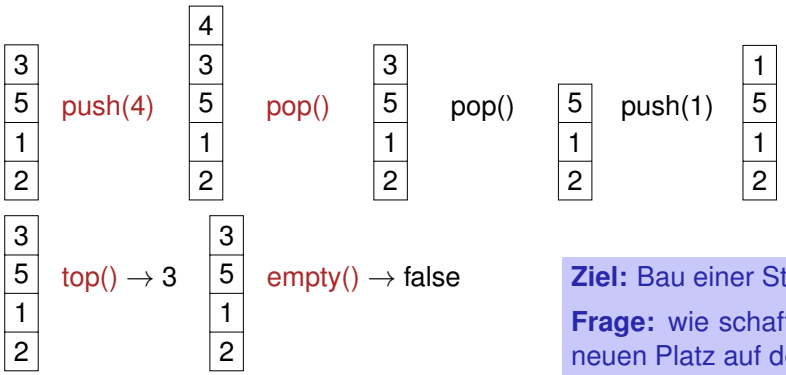
`rational.cpp`

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

Klassenname :: Membername

Motivation: Stapel

Motivation: Stapel (`push, pop, top, empty`)



Ziel: Bau einer Stapel-Klasse!

Frage: wie schaffen wir bei push neuen Platz auf dem Stapel?

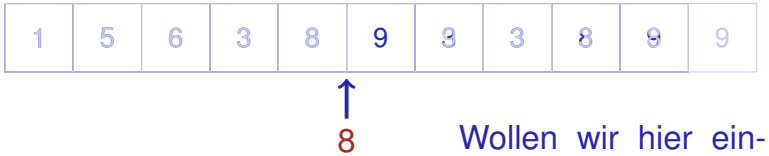
Wir brauchen einen neuen Container!

Unser Haupt-Container bisher: Array (`T[]`)

- Zusammenhängender Speicherbereich, wahlfreier Zugriff (auf i-tes Element)
- Simulation eines Stapels durch ein Array?
- Nein, irgendwann ist das Array “voll.”

Arrays können wirklich nicht alles...

- Einfügen oder Löschen von Elementen „in der Mitte” ist aufwändig.



Wollen wir hier einfügen, müssen wir alles rechts davon verschieben (falls da überhaupt noch Platz ist!)

Arrays können wirklich nicht alles...

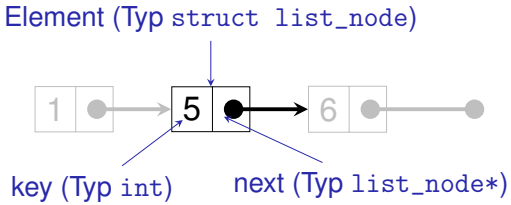
- Einfügen oder Löschen von Elementen „in der Mitte” ist aufwändig.

Wollen wir hier löschen, müssen wir alles rechts davon verschieben

Der neue Container: Verkettete Liste

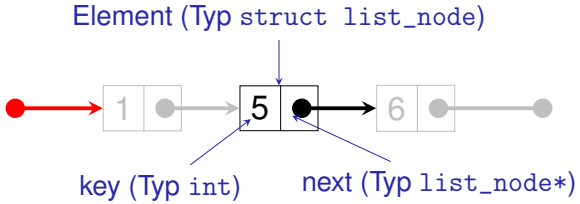
- *Kein* zusammenhängender Speicherbereich und *kein* wahlfreier Zugriff
- Jedes Element “kennt” seinen Nachfolger
- Einfügen und Löschen beliebiger Elemente ist einfach, *auch am Anfang der Liste*
- ⇒ Ein Stapel kann als verkettete Liste realisiert werden

Verkettete Liste: Zoom




```
struct list_node {
  int          key;
  list_node*   next;
  // constructor
  list_node (int k, list_node* n)
    : key (k), next (n) {}
};
```

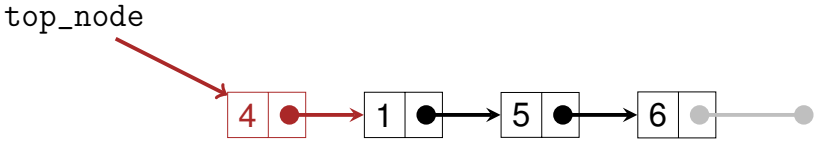

Stapel = Zeiger aufs oberste Element




```
class stack {
public:
   void push (int value) {...}
   ...
private:
   list_node* top_node;
};
```


Sneak Preview: `push(4)`

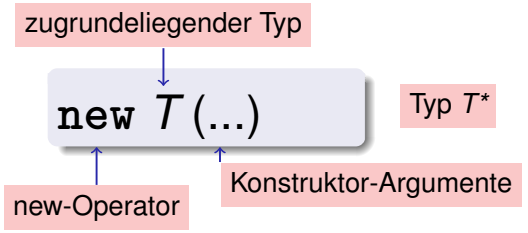
```
void push (int value)
{
  top_node = new list_node (value, top_node);
}
```



Dynamischer Speicher

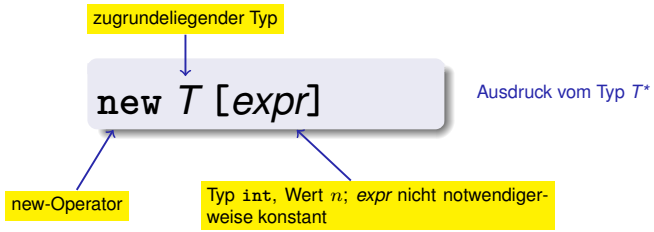
- Für dynamische Datenstrukturen wie Listen benötigt man *dynamischen Speicher*
- Bisher wurde die Grösse des Speicherplatzes für Variablen zur *Compilezeit* festgelegt
- Zeiger erlauben das Anfordern neuen Speichers zur *Laufzeit*
- Dynamische Speicherverwaltung in C++ mit Operatoren `new` und `delete`

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 für Felder

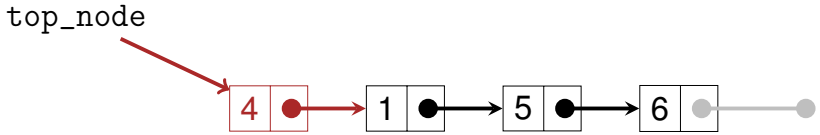


- Neuer Speicher für ein Feld der Länge n mit zugrundeliegendem Typ *T* wird angelegt
- Wert des Ausdrucks ist Adresse des ersten Elements des Feldes

Der `new`-Ausdruck: `push(4)`

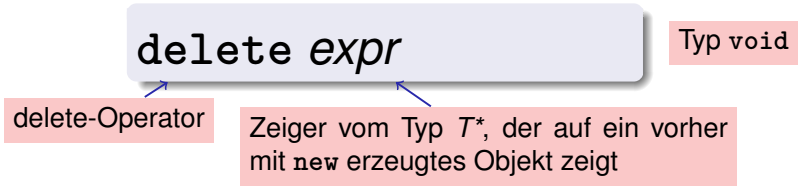
- **Effekt:** Neues Objekt vom Typ *T* wird im Speicher angelegt ...
- ... und mit Hilfe des passenden Konstruktors initialisiert.
- **Wert:** Adresse des neuen Objekts

```
top_node = new list_node (value, top_node);
```



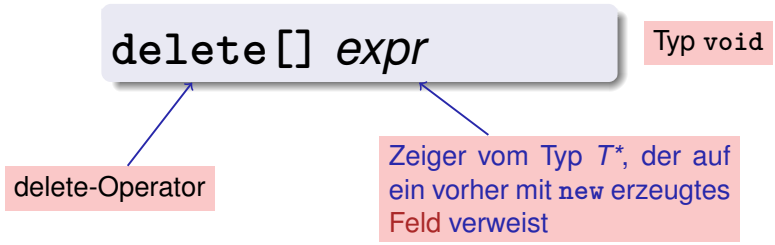
Der `delete`-Ausdruck

Objekte, die mit `new` erzeugt worden sind, haben *dynamische Speicherdauer:* sie “leben”, bis sie explizit *gelöscht* werden.



- **Effekt:** Objekt wird gelöscht, Speicher wird wieder freigegeben

Der `delete`-Ausdruck für Felder



- **Effekt:** Feld wird gelöscht, Speicher wird wieder freigegeben

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.
- `delete` kann leicht vergessen werden: Folge sind Speicherlecks (*memory leaks*). Kann auf Dauer zu Speicherüberlauf führen.

Wer geboren wird, muss sterben...

Richtlinie “Dynamischer Speicher”

Zu jedem `new` gibt es ein passendes `delete`!

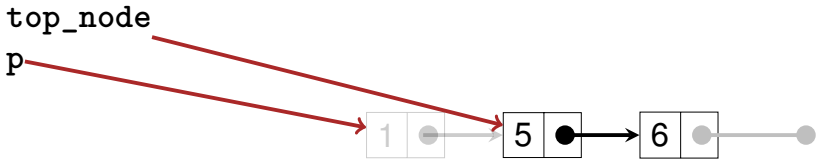
Nichtbeachtung führt zu *Speicherlecks*:

- “Alte” Objekte, die den Speicher blockieren...
- ...bis er irgendwann voll ist (heap overflow)

Weiter mit dem Stapel: pop()

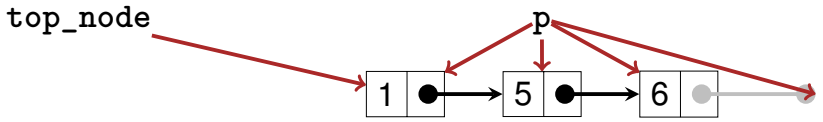
```
void pop()
{
    assert (!empty());
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

Abkürzung für `(*top_node).next`



Stapel traversieren: print()

```
void print (std::ostream& o) const
{
  const list_node* p = top_node;
  while (p != 0) {
    o << p->key << " "; // 1 5 6
    p = p->next;
  }
}
```



Stapel ausgeben: operator<<

```
class stack {
public:
   void push (int value) {...}
   ...
   void print (std::ostream& o) const {...}
private:
   list_node* top_node;
};

// POST: s is written to o
std::ostream& operator<< (std::ostream& o, const stack& s)
{
   s.print (o);
   return o;
}
```


Leerer Stapel , empty(), top()

```
stack()      // default constructor
    : top_node (0)
{}

bool empty () const
{
  return top_node == 0;
}

int top () const
{
  assert (!empty());
  return top_node->key;
}
```

Stapel fertig? Offenbar noch nicht...

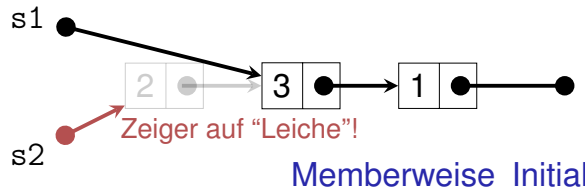
```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n"; // 2 3 1

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?



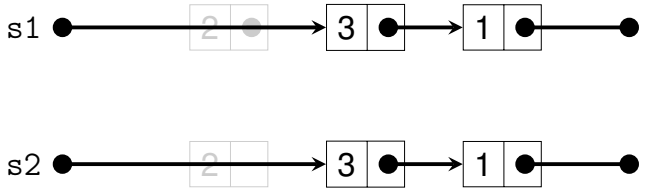
Memberweise Initialisierung: kopiert nur den `top_node`-Zeiger

```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();  // Oops, Programmabsturz!
```


Wir brauchen eine echte Kopie!




```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 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

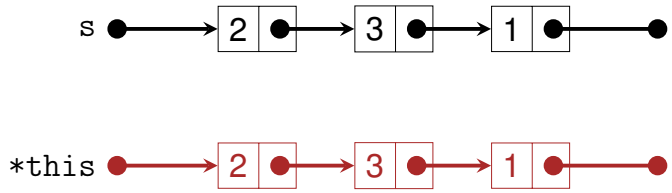
  T `x = t;` (t vom Typ T)
  T `x (t);`

- Falls kein Copy-Konstruktor deklariert ist, so wird er automatisch erzeugt (und initialisiert memberweise – Grund für obiges Problem)

Mit dem Copy-Konstruktor klappt's!

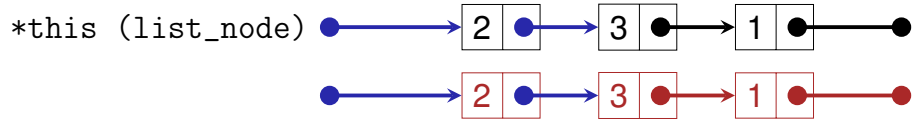
Hier wird eine Kopierfunktion des `list_node` benutzt:

```
// POST: *this is initialized with a copy of s
stack (const stack& s)
  : top_node (0)
{
  if (s.top_node != 0)
    top_node = s.top_node->copy();
}
```



Die (rekursive) Kopierfunktion von `list_node`

```
// POST: pointer to a copy of the list starting
//       at *this is returned
list_node* copy () const
{
  if (next != 0)
    return new list_node (key, next->copy());
  else
    return new list_node (key, 0);
}
```



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!

Hier wird eine Aufräumfunktion des `list_node` benutzt:

```
// POST: *this (left operand) is getting a copy of s (right operand)
stack& operator= (const stack& s)
{
  if (top_node != s.top_node) { // keine Selbtszuweisung!
    if (top_node != 0) {
      top_node->clear(); // loesche Knoten in *this
      top_node = 0;
  }
    if (s.top_node != 0)
       top_node = s.top_node->copy(); // kopiere s nach *this
    }
  return *this; // Rueckgabe als L-Wert (Konvention)
}
```


Die (rekursive) Aufräumfunktion des `list_node`

```
// POST: the list starting at *this is deleted
void clear ()
{
  if (next != 0)
    next->clear();
  delete this;
}
```

Zombie-Elemente

```
{
  stack s1; // local variable
  s1.push (1);
  s1.push (3);
  s1.push (2);
  std::cout << s1 << "\n"; // 2 3 1
}
// s1 has died (become invalid)...
```

- ... aber die drei *Elemente* des Stapels `s1` leben weiter (Speicherleck)!
- Sie sollten zusammen mit `s1` aufgeräumt werden!

Der Destruktor

- Der Destruktor einer Klasse T ist die eindeutige Memberfunktion mit Deklaration

  $$\sim T\,();$$

- Wird automatisch aufgerufen, wenn die Speicherdauer eines Klassenobjekts endet
- Falls kein Destruktor deklariert ist, so wird er automatisch erzeugt und ruft die Destruktoren für die Membervariablen auf (Zeiger `top_node`, kein Effekt – Grund für Zombie-Elemente)

Mit dem Destruktor klappt's!

```
// POST: the dynamic memory of *this is deleted
~stack()
{
  if (top_node != 0)
    top_node->clear();
}
```

- löscht automatisch alle Stapelemente, wenn der Stapel ungültig wird
- Unsere Stapel-Klasse befolgt jetzt die Richtlinie "Dynamischer Speicher"!

Dynamischer Datentyp

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Stapel)
- Andere Anwendungen:
  - Listen (mit Einfügen und Löschen "in der Mitte")
  - Bäume (nächste Woche)
  - Warteschlangen
  - Graphen
- Mindestfunktionalität:
  - Konstruktoren
  - Destruktor
  - Copy-Konstruktor
  - Zuweisungsoperator

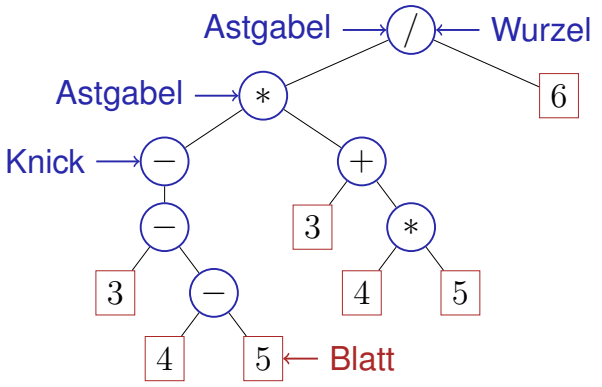
  } Dreierregel: definiert eine Klasse eines davon, so sollte sie auch die anderen zwei definieren!

19. Vererbung und Polymorphie

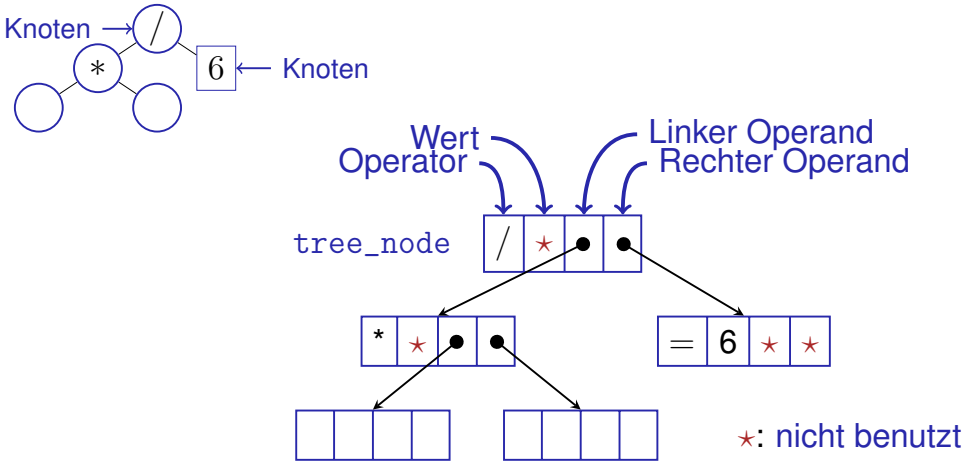
Ausdrucksbäume, Vererbung, Code-Wiederverwendung, virtuelle Funktionen, Polymorphie, Konzepte des objektorientierten Programmierens

(Ausdrucks-)Bäume

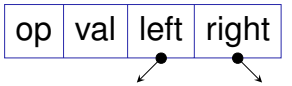
`-(3-(4-5))*(3+4*5)/6`



Astgabeln + Blätter + Knicke = Knoten

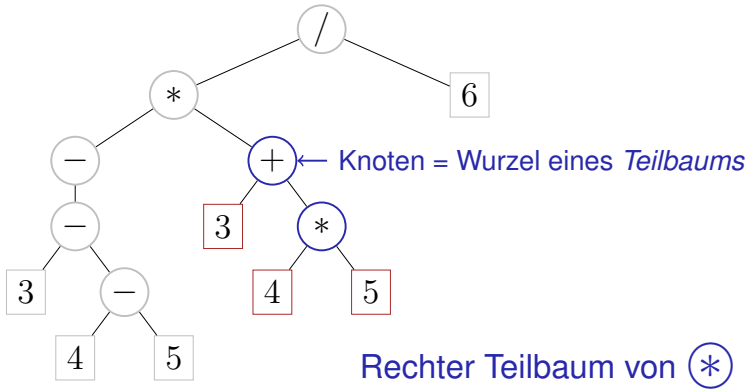


Knoten (`struct tree_node`)




```
struct tree_node {
  char op;
  // leaf node (op: '=')
  double val;
  // internal node ( op: '+', '-', '*', '/')
  tree_node* left; // == 0 für unäres minus
  tree_node* right;
  // constructor
  tree_node (char o, double v, tree_node* l, tree_node* r)
    : op (o), val (v), left (l), right (r)
  {}
};
```


Knoten und Teilbäume



Knoten in Teilbäumen zählen

```
struct tree_node {
  ...
    // POST: returns the size (number of nodes) of
    //       the subtree with root *this
    int size () const
    {
        int s=1;
        if (left) // kurz für left != 0
            s += left->size();
        if (right)
            s += right->size();
        return s;
    }
};
```

Teilbäume auswerten


```
struct tree_node {
  ...
  // POST: evaluates the subtree with root *this
  double eval () const {
    if (op == '=') return val;
    double l = 0;
    if (left) l = left->eval();
    double r = right->eval();
    if (op == '+') return l + r;
    if (op == '-') return l - r;
    if (op == '*') return l * r;
    if (op == '/') return l / r;
    return 0;
  }
};
```

Blatt...
...oder Astgabel:
op unär, oder linker Ast
rechter Ast

Teilbäume klonen


```
struct tree_node {
  ...
  // POST: a copy of the subtree with root *this is
  //       made, and a pointer to its root node is
  //       returned
  tree_node* copy () const {
    tree_node* to = new tree_node (op, val, 0, 0);
    if (left)
        to->left = left->copy();
    if (right)
        to->right = right->copy();
    return to;
  }
};
```

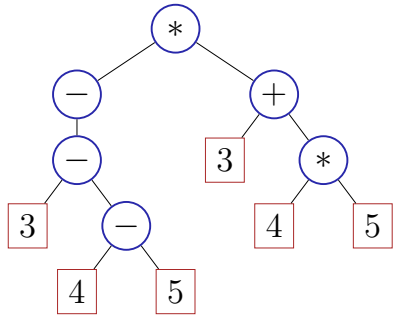

Teilbäume klonen - Kompaktere Schreibweise


```
struct tree_node {
  ...
  // POST: a copy of the subtree with root *this is
  //       made, and a pointer to its root node is
  //       returned
  tree_node* copy () const {
     return new tree_node (op, val,
            left ? left->copy() : 0,
            right ? right->copy() : 0);
  }
};
```

cond ? expr1 : expr 2 hat Wert *expr1*, falls *cond* gilt, *expr2* sonst

Teilbäume fällen


```
struct tree_node {
  ...
  // POST: all nodes in the subtree with root
  // *this are deleted
  void clear() {
    if (left) {
      left->clear();
    }
    if (right) {
      right->clear();
    }
    delete this;
  }
};
```



Bäumige Teilbäume

```
struct tree_node {
  ...
  // constructor
  tree_node (char o, tree_node* l,
             tree_node* r, double v)

  // functionality
  double eval () const;
  void print (std::ostream& o) const;
  int size () const;
  tree_node* copy () const;
  void clear ();
};
```

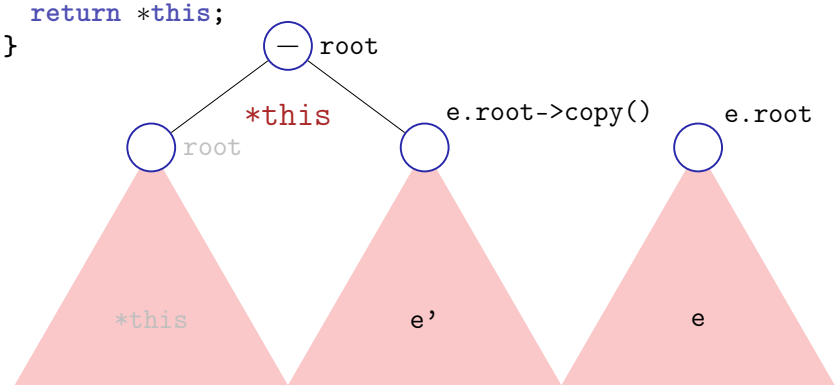

Bäume pflanzen

```
class texpression {
private:
    tree_node* root;
public:
    ...
    texpression (double d)
        : root (new tree_node ('=', d, 0, 0)) {}
    ...
};
```

erzeugt Baum mit einem Blatt

Bäume wachsen lassen

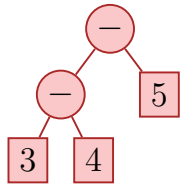
```
texpression& operator-= (const texpression& e)
{
  assert (e.root);
  root = new tree_node ('-', 0, root, e.root->copy());
  return *this;
}
```



Bäume züchten

```
texpression operator- (const texpression& l,
                       const texpression& r)
{
    texpression result = l;
    return result -= r;
}

texpression a = 3;
texpression b = 4;
texpression c = 5;
texpression d = a-b-c;
```



Bäume züchten

Es gibt für `texpression` auch noch

- Default-Konstruktor, Copy-Konstruktor, Assignment-Operator, Destruktor
- die arithemtischen Zuweisungen `+=`, `*=`, `/=`
- die binären Operatoren `+`, `*`, `/`
- das unäre `-`

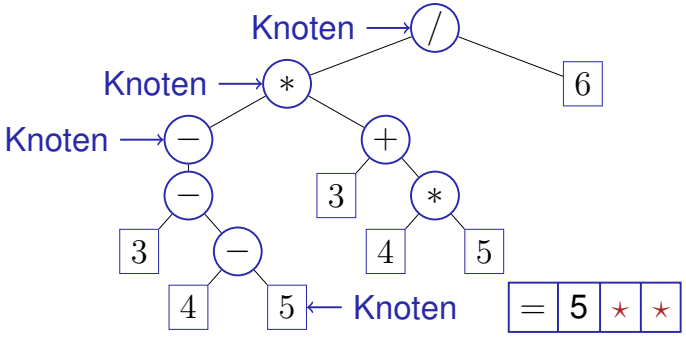
Von Werten zu Bäumen!

```
typedef texpression result_type;   // Typ-Alias

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

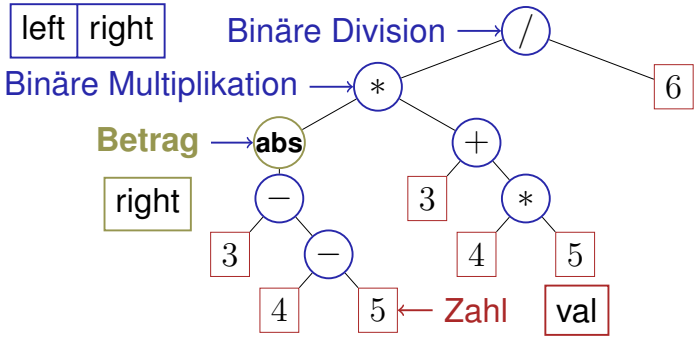
`double_calculator.cpp`
(Ausdruckswert)
→
`texpression_calculator_1.cpp`
(Ausdrucksbaum)

Motivation Vererbung: Bisher



- Astgabeln + Blätter + Knicke = Knoten
- ⇒ Unbenutzte Membervariablen ⋆

Motivation Vererbung: Die Idee



- Überall nur die benötigten Membervariablen!
- Zoo-Erweiterung mit neuen Arten!

Vererbung – Der Hack, zum ersten...

Szenario: Erweiterung des Ausdrucksbaumes um mathematische Funktionen, z.B. `abs`, `sin`, `cos`:

- Erweiterung der Klasse `tree_node` um noch mehr Membervariablen

```
struct tree_node{
    char op; // neu: op = 'f' -> Funktion
    ...
    std::string name; // function name;
}
```

Nachteile:

- Veränderung des Originalcodes (unerwünscht)
- Noch mehr unbenutzte Membervariablen...

Vererbung – Der Hack, zum zweiten...

Szenario: Erweiterung des Ausdrucksbaumes um mathematische Funktionen, z.B. `abs`, `sin`, `cos`:

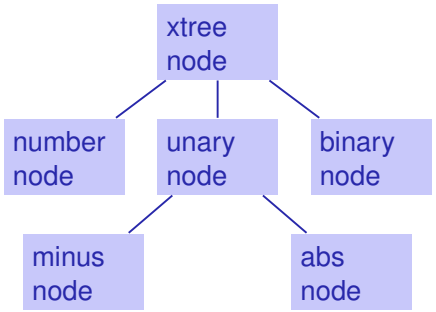
- Anpassung jeder einzelnen Memberfunktion member function

```
double eval () const
{   ...
    else if (op == 'f')
        if (name == "abs")
            return std::abs(right->eval());
    ...
}
```

Nachteile:

- Verlust der Übersichtlichkeit
- Zusammenarbeit mehrerer Entwickler schwierig

Vererbung – die saubere Lösung



- „Aufspaltung" von `tree_node`
- Gemeinsame Eigenschaften verbleiben in der *Basisklasse* `xtree_node` (Erklärung folgt)

Vererbung

Klassen können Eigenschaften *(ver)erben*:

```
struct xtree_node{
  virtual int size() const;
  virtual double eval () const;
};
```

erbt von — Vererbung sichtbar

```
struct number_node : public xtree_node {
  double val;
```

nur für `number_node`

```
  int size () const;
  double eval () const;
};
```

Mitglieder von `xtree_node` werden *überschrieben*

Vererbung – Nomenklatur

```
class A {
    ...
}
```
Basisklasse (Superklasse)

```
class B: public A{
    ...
}
```
Abgeleitete Klasse (Subklasse)

```
class C: public B{
    ...
}
```
„B und C *erben von* A”
„C erbt von B”

Aufgabenteilung: Der Zahlknoten

```
struct number_node: public xtree_node{
  double val;

  number_node (double v) : val (v) {}

  double eval () const {
    return val;
  }

  int size () const {
     return 1;
  }
};
```


Ein Zahlknoten ist ein Baumknoten…

- Ein (Zeiger auf ein) erbendes Objekt kann überall dort verwendet werden, wo ein (Zeiger auf ein) Basisobjekt gefordert ist, aber nicht umgekehrt.

```
number_node* num = new number_node (5);

xtree_node* tn = num; // ok, number_node is
                      // just a special xtree_node

xtree_node* bn = new add_node (tn, num); // ok

number_node* nn = tn; //error:invalid conversion
```


Anwendung

```
class xexpression {
private:
   xtree_node* root;
public:
    xexpression (double d)
      : root (new number_node (d)) {}

    xexpression& operator-= (const xexpression& t)
    {
      assert(t.root);
      root = new sub_node (root, t.root->copy());
      return *this;
    }
     ...
}
```
statischer Typ

dynamischer Typ

Polymorphie

- Virtuelle Mitgliedsfunktion: der *dynamische* Typ bestimmt bei Zeigern auf erbende Objekte die auszuführenden Memberfunktionen

```
struct xtree_node {
  virtual double eval();
...
};
```

- Ohne `virtual` wird der *statische Typ* zur Bestimmung der auszuführenden Funktion herangezogen.

Wir vertiefen das nicht weiter.

Aufgabenteilung: Binäre Knoten

```
struct binary_node : public xtree_node {
  xtree_node* left; // INV != 0
  xtree_node* right; // INV != 0

  binary_node (xtree_node* l, xtree_node* r) :
    left (l), right (r)
  {
    assert (left);
    assert (right);
  }

  int size () const {
    return 1 + left->size() + right->size();
  }
};
```

`size` funktioniert für *alle* binären Knoten. Abgeleitete Klassen (`add_node`, `sub_node`...) *erben* diese Funktion!

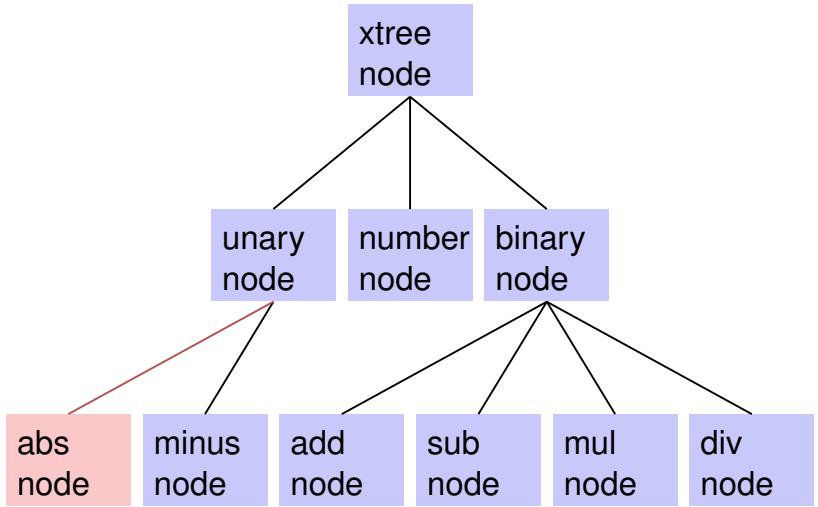
Aufgabenteilung: +, −, * ...

```
struct sub_node : public binary_node {
  sub_node (xtree_node* l, xtree_node* r)
    : binary_node (l, r) {}

  double eval () const {
    return left->eval() - right->eval();
  }
};
```

`eval` spezifisch für +, −, *, /

Erweiterung um `abs` Funktion



Erweiterung um `abs` Funktion

```
struct unary_node: public xtree_node
{
  xtree_node* right; // INV != 0
  unary_node (xtree_node* r);
  int size () const;
};

struct abs_node: public unary_node
{
  abs_node (xtree_node* arg) : unary_node (arg) {}

  double eval () const {
    return std::abs (right->eval());
  }

};
```


Da ist noch was... Speicherbehandlung

```
struct xtree_node {
  ...
  // POST: a copy of the subtree with root
  //       *this is made, and a pointer to
  //       its root node is returned
  virtual xtree_node* copy () const;

  // POST: all nodes in the subtree with
  //        root *this are deleted
  virtual void clear () {};
};
```


Da ist noch was... Speicherbehandlung

```
struct unary_node: public xtree_node {
  ...
  virtual void clear () {
    right->clear();
    delete this;
  }
};

struct minus_node: public unary_node {
  ...
  xtree_node* copy () const
  {
    return new minus_node (right->copy());
  }
};
```


`xtree_node` ist kein dynamischer Datentyp ??

- Wir haben keine Variablen vom Typ `xtree_node` mit automatischer Speicherdauer
- Copy-Konstruktor, Zuweisungsoperator und Destruktor sind überflüssig
- Speicherverwaltung in der *Container-Klasse*

```
class xexpression {
  // Copy-Konstruktor
  xexpression (const xexpression& v);
  // Zuweisungsoperator
  xexpression& operator=(const xexpression& v);
  // Destruktor
  ~xexpression ();
};
```

`xtree_node::copy`

`xtree_node::clear`

Mission: Monolithisch $\rightarrow$ modular ✓

```
struct tree_node {
  char op;
  double val;
  tree_node* left;
  tree_node* right;
  ...

  double eval () const
  {
    if (op == '=') return val;
    else {
    double l = 0;
    if (left != 0) l = left->eval();
    double r = right->eval();
    if (op == '+') return l + r;
    if (op == '-') return l - r;
    if (op == '*') return l * r;
    if (op == '/') return l / r;
    assert (false); // unknown operator
    return 0;
  }

  int size () const { ... }

  void clear() { ... }

  tree_node* copy () const { ... }

};
```

```
struct number_node: public xtree_node {
  double val;
  ...
  double eval () const {
  return val;
  }
};

struct minus_node: public unary_node {
  ...
  double eval () const {
    return -right->eval();
  }
};

struct minus_node : public binary_node {
  ...
  double eval () const {
    return left->eval() - right->eval();
  }
}

struct abs_node : public unary_node {
  ...
  double eval () const {
    return left->eval() - right->eval();
  }
}
```


Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Kapselung

- Verbergen der Implementierungsdetails von Typen (privater Bereich)
- Definition einer Schnittstelle zum Zugriff auf Werte und Funktionalität (öffentlicher Bereich)
- Ermöglicht das Sicherstellen von Invarianten und den Austausch der Implementierung

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Vererbung

- Typen können Eigenschaften von Typen erben.
- Abgeleitete Typen können neue Eigenschaften besitzen oder vorhandene überschreiben.
- Macht Code- und Datenwiederverwendung möglich.

Zusammenfassung der Konzepte

.. der objektorientierten Programmierung

Polymorphie

- Ein Zeiger kann abhängig von seiner Verwendung unterschiedliche zugrundeliegende Typen haben.
- Die unterschiedlichen Typen können bei gleichem Zugriff auf ihre gemeinsame Schnittstelle verschieden reagieren.
- Macht „nicht invasive” Erweiterung von Bibliotheken möglich.

20. Zusammenfassung

Zweck und Format

Nennung der wichtigsten Stichwörter zu den Kapiteln. Checkliste: "kann ich mit jedem Begriff etwas anfangen?"

- Ⓜ Motivation: Motivierendes Beispiel zum Kapitel
- Ⓚ Konzepte: Konzepte, die nicht von der Implementation (Sprache) C++abhängen
- Ⓢ Sprachlich (C++): alles was mit der gewählten Sprache zusammenhängt
- Ⓑ Beispiele: genannte Beispiele der Vorlesung

1. Einführung

Ⓜ Ⓚ
- Euklidischer Algorithmus
- Algorithmus, Turingmaschine, Programmiersprachen, Kompilation, Syntax und Semantik
- Werte und Effekte, (Fundamental)typen, Literale, Variablen, Bezeichner, Objekte, Ausdrücke, Operatoren, Anweisungen

Ⓢ
- Include-Direktiven `#include <iostream>`
- Hauptfunktion `int main(){...}`
- Kommentare, Layout `// Kommentar`
- Typen, Variablen, L-Wert `a` , R-Wert `a+b`
- Ausdrucksanweisung `b=b*b;` , Deklarationsanweisung `int a;`, Rückgabeanweisung `return 0;`

2. Ganze Zahlen

Ⓜ
- Celsius to Fahrenheit

Ⓚ
- Assoziativität und Präzedenz, Stelligkeit
- Ausdrucksbäume, Auswertungsreihenfolge
- Arithmetische Operatoren
- Binärzahldarstellung, Hexadezimale Zahlen, Wertebereich
- Zahlendarstellung mit Vorzeichen, Zweierkomplement

Ⓢ
- Arithmetische Operatoren `9 * celsius / 5 + 32`
- Inkrement / Dekrement `expr++`
- Arithmetische Zuweisungen `expr1 += expr2`
- Konversion `int` ↔ `unsigned int`

Ⓑ
- Celsius to Fahrenheit, Ersatzwiderstand

3. Wahrheitswerte

Ⓚ
- Boole'sche Funktionen, Vollständigkeit
- DeMorgan'sche Regeln

Ⓢ
- Der Typ `bool`
- Logische Operationen `a && !b`
- Relationale Operationen `x < y`
- Präzedenzen `7 + x < y && y != 3 * z`
- Kurzschlussauswertung `x != 0 && z / x > y`
- Die `assert`-Anweisung, `#include <cassert>`

Ⓑ
- Div-Mod Identität.

4./5. Kontrollanweisungen

Ⓜ
- Linearer Kontrollfluss vs. interessante Programme, Spaghetti-Code

Ⓚ
- Auswahlanweisungen, Iterationsanweisungen
- (Vermeidung von) Endlosschleifen, Halteproblem
- Sichtbarkeits- und Gültigkeitsbereich, Automatische Speicherdauer
- Äquivalenz von Iterationsanweisungen

Ⓢ
- if Anweisungen `if (a % 2 == 0) {..}`
- for Anweisungen `for (unsigned int i = 1; i <= n; ++i) ...`
- while und do-Anweisungen `while (n > 1) {...}`
- Blöcke, Sprunganweisungen `if (a < 0) continue;`

Ⓑ
- Summenberechnung (Gauss), Primzahltest, Collatz-Folge, Fibonacci Zahlen, Taschenrechner

6./7. Fliesskommazahlen

Ⓜ
- Richtig Rechnen: Celsius / Fahrenheit

Ⓚ
- Fixkomma- vs. Fliesskommazahldarstellung
- (Löcher im) Wertebereich
- Rechnen mit Fliesskommazahlen, Umrechnung
- Fliesskommazahlensysteme, Normalisierung, IEEE Standard 754
- *Richtlinien für das Rechnen mit Fliesskommazahlen*

Ⓢ
- Typen `float`, `double`
- Fliesskommaliterale `1.23e-7f`

Ⓑ
- Celsius/Fahrenheit, Euler, Harmonische Zahlen

8./9. Funktionen

Ⓜ
- Potenzberechnung

Ⓚ
- Kapselung von Funktionalität
- Funktionen, formale Argumente, Aufrufargumente
- Gültigkeitsbereich, Vorwärts-Deklaration
- Prozedurales Programmieren, Modularierung, Getrennte Übersetzung
- *Stepwise Refinement*

Ⓢ
- Funktionsdeklaration, -definition `double pow(double b, int e){ ... }`
- Funktionsaufruf `pow (2.0, -2)`
- Der typ `void`

Ⓑ
- Potenzberechnung, perfekte Zahlen, Minimum, Kalender

10. Referenztypen

Ⓜ
- Funktion Swap

Ⓚ
- Werte-/ Referenzsemantik, Call by Value / Call by Reference
- Lebensdauer von Objekten / Temporäre Objekte
- Konstanten

Ⓢ
- Referenztyp `int& a`
- Call by Reference und Return by Reference `int& increment (int& i)`
- Const-Richtlinie, Const-Referenzen, Referenzrichtlinie

Ⓑ
- Swap, Inkrement

11./12. Felder (Arrays)

Ⓜ
- Iteration über Daten: Feld des Eratosthenes

Ⓚ
- Felder, Speicherlayout, Wahlfreier Zugriff
- (Fehlende) Grenzenprüfung
- Vektoren
- Zeichen: ASCII, UTF8, Texte, Strings

Ⓢ
- Feldtypen `int a[5] = {4,3,5,2,1};`
- Zeichen und Texte, der Typ char `char c = 'a';`, Konversion nach `int`
- Mehrdimensionale Felder, Vektoren von Vektoren

Ⓑ
- Sieb des Eratosthenes, Caesar-Code, Kürzeste Wege, Lindenmayer-Systeme

13./14. Zeiger, Iteratoren, Container

Ⓜ
- Arrays als Funktionsargumente

Ⓚ
- Zeiger, Möglichkeiten und Gefahren der Indirektion
- Wahlfreier Zugriff vs. Iteration, Zeiger-Arithmetik
- Container und Iteratoren

Ⓢ
- Zeiger `int* x;`, Konversion Feld $\rightarrow$ Zeiger, Nullzeiger
- Adress-, Dereferenzoperator `int *ip = &i; int j = *ip;`
- Zeiger und Const `const int *a;`
- Algorithmen und Iteratoren `std::fill (a, a+5, 1);`
- Typdefinitionen `typedef std::set<char>::const_iterator Sit;`

Ⓑ
- Füllen eines Feldes, Buchstabensalat

15./16. Rekursion

Ⓜ
- Rekursive math. Funktionen, Taschenrechner

Ⓚ
- Rekursion
- Aufrufstapel, Gedächtnis der Rekursion
- Korrektheit, Terminierung,
- Rekursion vs. Iteration
- EBNF, Formale Grammatiken, Ströme, Parsen
- Auswertung, Assoziativität

Ⓑ
- Fakultät, GGT, Fibonacci, Berge, Taschenrechner

17. Structs und Klassen I

Ⓜ
- Datentyp Rationale Zahlen selber bauen

Ⓚ
- Heterogene Datenstruktur
- Funktions- und Operator-Overloading
- Datenkapselung

Ⓢ
- Struct Definition `struct rational {int n; int d;};`
- Mitgliedszugriff `result.n = a.n * b.d + a.d * b.n;`
- Initialisierung und Zuweisung,
- Überladen von Funktionen `pow(2)` vs. `pow(3,3);`, Überladen von Operatoren

Ⓑ
- rationale Zahlen, komplexe Zahlen

18. Klassen, Dynamische Datentypen

Ⓜ
- Rationale Zahlen mit Kapselung, Stack

Ⓚ
- Verkettete Liste, Allokation, Deallokation, Dynamischer Datentyp

Ⓢ
- Klassen `class rational { ... };`
- Zugriffssteuerung `public:` / `private:`
- Mitgliedsfunktionen `int rational::denominator () const`
- Copy-Konstruktor, Destruktor, Dreierregel
- Konstruktoren `rational (int den, int nm): d(den), n(no) {}`
- `new` und `delete`
- Copy-Konstruktor, Zuweisungs-Operator, Destruktor

Ⓑ
- Verkettete Liste, Stack

19. Baumstrukturen, Vererbung und Polymorphie

Ⓜ
- Ausdrucksbäume,
- Erweiterung von Ausdrucksbäumen
- Vererbung

Ⓚ
- Baumstrukturen
- Vererbung
- Polymorphie

Ⓢ
- Vererbung `class tree_node: public number_node`
- Virtuelle Funktionen `virtual void size() const;`

Ⓑ
- Ausdrucksbaum, Parser auf Ausdrücken, Erweiterung um abs-Knoten

Ende

Ende der Vorlesung.