



Felix Friedrich, Malte Schwerhoff

Informatik

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

Herbst 2019

Willkommen

zur Vorlesung Informatik

am MATH/PHYS Department der ETH Zürich.

Ort und Zeit:

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

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

Vorlesungs-Webseite

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

Team

Chefassistent
Assistenten

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

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

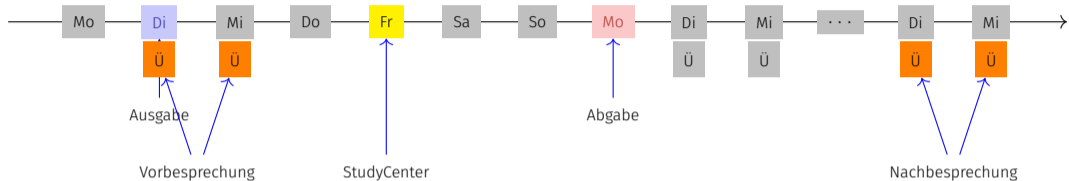
Dozenten

Dr. Malte Schwerhoff / Dr. Felix Friedrich

Einschreibung in Übungsgruppen

- Gruppeneinteilung selbstständig via Webseite
- Einschreibung bereits offen

Ablauf



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

Zu den Übungen

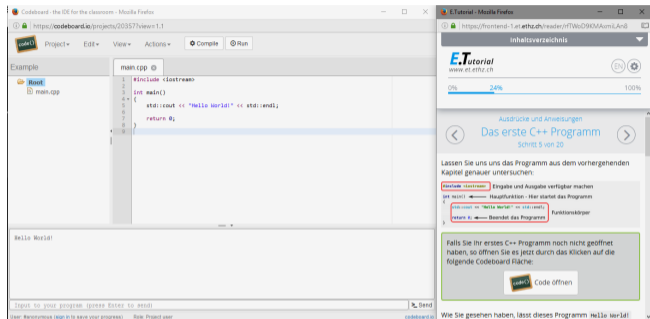
- Bearbeitung der wöchentlichen Uebungsserien ist also freiwillig, wird aber **dringend** empfohlen!

Fehlende Ressourcen 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 zweiten 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 wird sowohl praktisches Wissen (Programmierfähigkeit) als auch theoretisches Wissen (Hintergründe, Systematik) geprüft.

Unser Angebot (VVZ)

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

Unser Angebot (Konkret)

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

Akademische Lauterkeit

Regel

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

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

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

Credits

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

1. Einführung

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

Was ist Informatik?

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

(Wikipedia, nach dem „Duden Informatik“)

Informatik vs. Computer

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

Mike Fellows, US-Informatiker (1991)

Informatik vs. Computer

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

Informatik \neq EDV-Kenntnisse

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

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

Informatik: *Grundlagenwissen*

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

Zurück in die Gegenwart: Inhalt dieser Vorlesung

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

Algorithmus: Kernbegriff der Informatik

Algorithmus:

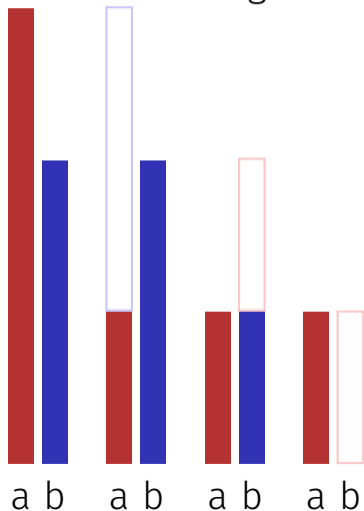
- Handlungsanweisung zur schrittweisen Lösung eines Problems
- Ausführung erfordert keine Intelligenz, nur Genauigkeit (sogar Computer können es)
- nach *Muhammed al-Chwarizmi*, Autor eines arabischen Rechen-Lehrbuchs (um 825)



“Dixit algorizmi...” (lateinische Übersetzung)

Der älteste nichttriviale Algorithmus

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



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

Solange $b \neq 0$

Wenn $a > b$ dann

$$a \leftarrow a - b$$

Sonst:

$$b \leftarrow b - a$$

Ergebnis: a .

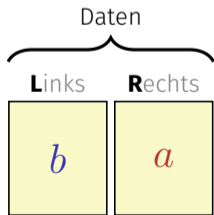
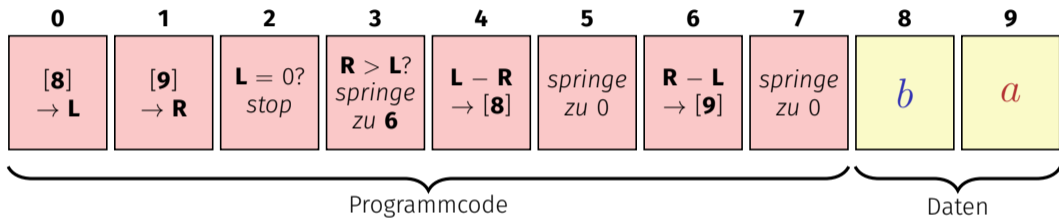
Algorithmen: 3 Abstraktionsstufen

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

Euklid: Kernidee und Pseudocode gesehen, Implementierung noch nicht

Euklid in der Box

Speicher



Solange $b \neq 0$

Wenn $a > b$ dann

$$a \leftarrow a - b$$

Sonst:

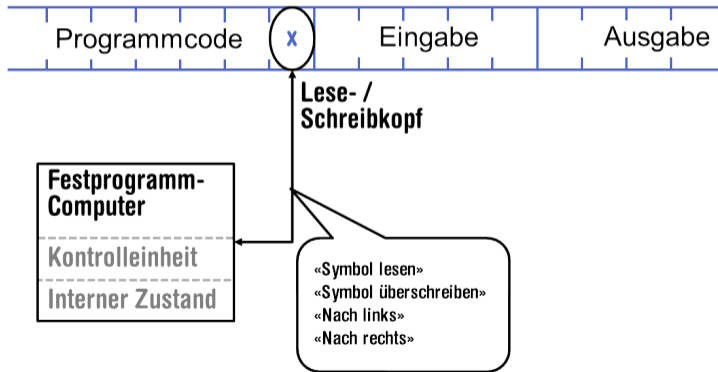
$$b \leftarrow b - a$$

Ergebnis: a .

Register

Computer – Konzept

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

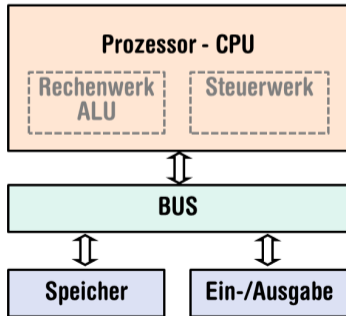


Alan Turing

Computer – Umsetzung

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

Von Neumann Architektur



Konrad Zuse



John von Neumann

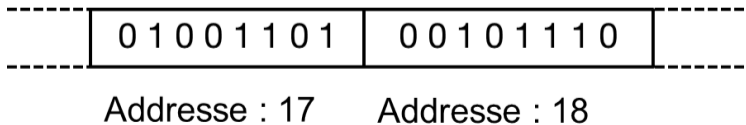
Computer

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)

Programmieren

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



The Harvard Computers, Menschliche Berufsrechner, ca.1890

Rechengeschwindigkeit

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



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

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

¹Uniprozessor Computer bei 1GHz

Warum Programmieren?

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

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

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

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

Darum Programmieren!

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

Programmiersprachen

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

Höhere Programmiersprachen

darstellbar als Programmtext, der

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

Programmiersprachen – Einordnung

Unterscheidung in

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

Warum C++?

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

Allgemeiner Konsens

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

Warum C++?

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

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

Syntax und Semantik

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

Deutsch vs. C++

Deutsch

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

C++

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

C++: Fehlerarten illustriert an deutscher Sprache

- Das Auto fuhr zu schnell.
- DasAuto fuh r zu sxhnell.
- Rot das Auto ist.
- Man empfiehlt dem Dozenten nicht zu widersprechen
- Sie ist nicht gross und rothaarig.
- Die Auto ist rot.
- Das Fahrrad galoppiert schnell.
- Manche Tiere riechen gut.

Syntaktisch und semantisch korrekt.

Syntaxfehler: Wortbildung.

Syntaxfehler: Satzstellung.

Syntaxfehler: Satzzeichen fehlen .

Syntaktisch korrekt aber mehrdeutig. [kein Analogon]

Syntaktisch korrekt, doch semantisch fehlerhaft: Falscher Artikel. [Typfehler]

Syntaktisch und grammatikalisch korrekt! Semantisch fehlerhaft. [Laufzeitfehler]

Syntaktisch und semantisch korrekt. Semantisch mehrdeutig. [kein Analogon]

Syntax und Semantik von C++

Syntax:

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

Semantik:

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

Syntax und Semantik von C++

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

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

Was braucht es zum Programmieren?

- **Editor:** Programm zum Ändern, Erfassen und Speichern von C++-Programmtext
- **Compiler:** Programm zum Übersetzen des Programmtexts in Maschinsprache
- **Computer:** Gerät zum Ausführen von Programmen in Maschinsprache
- **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
- **Ausdrücke**
- L- und R-Werte
- Operatoren
- Anweisungen

Das erste C++ Programm

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

Verhalten eines Programmes

Zur Compilationszeit:

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

Zur Laufzeit:

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

„Beiwerk“: Kommentare

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

Kommentare

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

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)
 - Rückgabewert (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.

```
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

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

Effekt: Ausgabe des Strings Compute

Effekt: Eingabe einer Zahl und Speichern in a

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

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

Effekt: Rückgabe des Wertes 0

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

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

Typ-
namen

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**)
- ...

Variablen

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

`int a;` definiert Variable mit

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

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!

Ausdrücke

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

Analogie: Baukasten

Ausdrücke

Baukasten

```
// input
```

Zusammengesetzter Ausdruck

```
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";
```

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).

$a * a$

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

$b = b * b$

- 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.

Literale

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

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

L-Werte und R-Werte

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

R-Wert

L-Wert (Ausdruck + Adresse)

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

L-Wert (Ausdruck + Adresse)

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

Beispiel: Literal 0

Operatoren und Operanden

Baukasten

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

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

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

Linker Operand (Ausgabestrom)

Ausgabe-Operator

Rechter Operand (String)

Rechter Operand (Variablenname)

Eingabe-Operator

Linker Operand (Eingabetrom)

Zuweisungsoperator

Multiplikationsoperator

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: $\mathbf{a * a}$ und $\mathbf{b * b}$

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: $\mathbf{b = b * b}$ und $\mathbf{a = b}$

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**

Beispiel: power8.cpp

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

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

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

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

Terminologie: L-Werte und R-Werte

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

- Ausdruck der einen **Speicherplatz** identifiziert
- Zum Beispiele eine Variable **a**
(später im Kurs lernen wir weitere L-Werte kennen)
- **Wert** ist der Inhalt an der Speicheradresse entsprechend dem Typ des Ausdrucks (für **a** z.B. Wert **2**).
- L-Wert kann seinen Wert *ändern* (z.B. per Zuweisung).

Terminologie: L-Werte und R-Werte

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

- Ausdruck der kein L-Wert ist
- Beispiel: Integer-Literal `0`
- Jeder L-Wert kann als R-Wert benutzt werden (aber nicht umgekehrt) ...
- ... indem der *Wert* des L-Werts genommen wird (z.B. könnte der L-Wert `a` den Wert `2` haben, der dann als R-Wert verwendet wird)
- Ein R-Wert kann seinen Wert *nicht ändern*

L-Werte und R-Werte

```
std::cout << "Compute a^8 for a = ? ";  
int a;  
std::cin >> a;  
int r = a * a; // r = a^2  
r = r * r; // r = a^4  
std::cout << a << "^8 = " << r * r << ".\n";  
return 0;
```

R-Wert

L-Wert (Ausdruck + Adresse)

L-Wert (Ausdruck + Adresse)

R-Wert

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

Celsius to Fahrenheit

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

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

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

9 * celsius / 5 + 32

9 * celsius / 5 + 32

- Arithmetischer Ausdruck,
 - 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

Vorzeichen

$-3 - 4$

bedeutet

$(-3) - 4$

Regel 3: Stelligkeit

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

Klammerung

Jeder Ausdruck kann mit Hilfe der

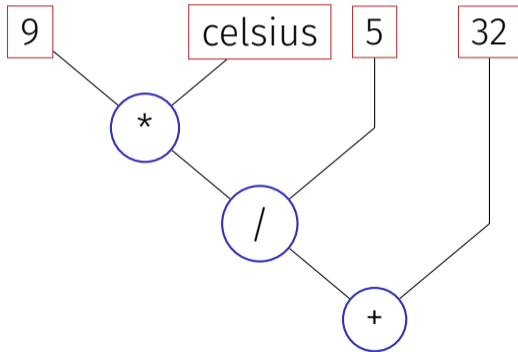
- Assoziativitäten
- Präzedenzen
- Stelligkeiten (Anzahl Operanden)

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

Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

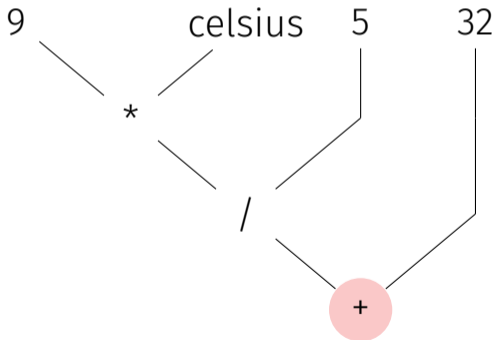
`((9 * celsius) / 5) + 32)`



Auswertungsreihenfolge

„Von oben nach unten“ im Ausdrucksbaum

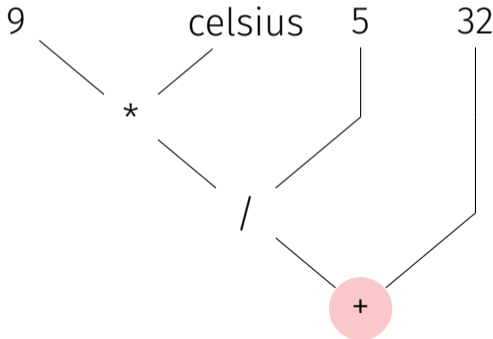
9 * celsius / 5 + 32



Auswertungsreihenfolge

Reihenfolge nicht eindeutig bestimmt:

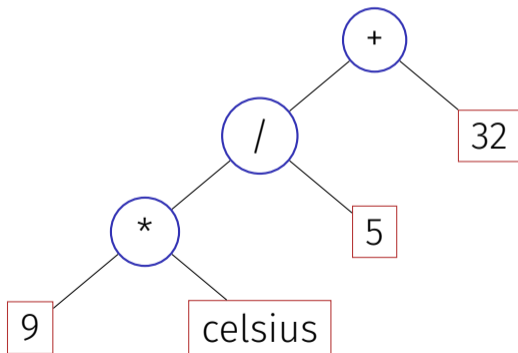
$$9 * \text{celsius} / 5 + 32$$



Ausdrucksbäume – Notation

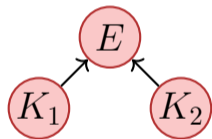
Üblichere Notation: Wurzel oben

9 * celsius / 5 + 32



Auswertungsreihenfolge – formaler

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



C++: anzuwendende gültige Reihenfolge nicht spezifiziert.

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

Auswertungsreihenfolge

Richtlinie

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

Arithmetische Operatoren

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

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

Einschub: Zuweisungsausdruck – nun genauer

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

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

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

Division

- Operator / realisiert ganzzahlige Division

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

- In `fahrenheit.cpp`

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

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

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

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

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

Präzisionsverlust

Richtlinie

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

Division und Modulo

- Modulo-Operator berechnet Rest der ganzzahligen Division

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

- Es gilt

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

- Es gilt auch:

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

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

Inkrement und Dekrement

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

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

Nachteile

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

In-/Dekrement Operatoren

Post-Inkrement

```
expr++
```

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

Prä-Inkrement

```
++expr
```

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

Post-Dekrement

```
expr--
```

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

Prä-Dekrement

```
--expr
```

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

In-/Dekrement Operatoren

	Gebrauch	Stelligkeit	Prüz	Assoz.	L/R-Werte
Post-Inkrement	<code>expr++</code>	1	17	links	L-Wert → R-Wert
Prä-Inkrement	<code>++expr</code>	1	16	rechts	L-Wert → L-Wert
Post-Dekrement	<code>expr--</code>	1	17	links	L-Wert → R-Wert
Prä-Dekrement	<code>--expr</code>	1	16	rechts	L-Wert → L-Wert

In-/Dekrement Operatoren

```
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

$$\begin{array}{c} a \ += \ b \\ \Leftrightarrow \\ a \ = \ a \ + \ b \end{array}$$

Analog für $-$, $*$, $/$ und $\%$

Arithmetische Zuweisungen

	Gebrauch	Bedeutung
<code>+=</code>	<code>expr1 += expr2</code>	<code>expr1 = expr1 + expr2</code>
<code>-=</code>	<code>expr1 -= expr2</code>	<code>expr1 = expr1 - expr2</code>
<code>*=</code>	<code>expr1 *= expr2</code>	<code>expr1 = expr1 * expr2</code>
<code>/=</code>	<code>expr1 /= expr2</code>	<code>expr1 = expr1 / expr2</code>
<code>%=</code>	<code>expr1 %= expr2</code>	<code>expr1 = expr1 % expr2</code>

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

101011 entspricht **43**.

Niedrigstes Bit, Least Significant Bit (LSB)

Höchstes Bit, Most Significant Bit (MSB)

Rechentricks

- Abschätzung der Grössenordnung von Zweierpotenzen²:

$$2^{10} = 1024 = 1\text{Ki} \approx 10^3.$$

$$2^{20} = 1\text{Mi} \approx 10^6,$$

$$2^{30} = 1\text{Gi} \approx 10^9,$$

$$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi}.$$

$$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}.$$

²Dezimal 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**

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“.

Wozu Hexadezimalzahlen?

„Für Programmierer und Techniker“ (Bedienungsanleitung des Schachcomputers *Mephisto II*, 1981)



Beispiele:

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



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:

$$ABCD = (A \times 16^3) + (B \times 16^2) + (C \times 16^1) + (D \times 16^0)$$

Für A gilt: 7 = -1; 8 = 0; 9 = +1 usw.

Eine Bauereinheit (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:



c) Anzeige **805E**
(E=-14) Umrechnung nach folgendem Verfahren:
 $(14 \times 16^0) + (5 \times 16^1) + (0 \times 16^2) + (0 \times 16^3) = 14 + 80 + 0 + 0 =$
 $= +94 \text{ Punkte.}$



d) Anzeige **7F80**
(7=-1; F=15) Umrechnung wie folgt:
 $(0 \times 16^0) + (8 \times 16^1) + (15 \times 16^2) - (1 \times 16^3) = 0 + 128 + 3840 - 4096 =$

Beispiel: Hex-Farben

#00FF00

r g b

Wozu Hexadezimalzahlen?

Die NZZ hätte viel Platz sparen können...

4e 5a 5a

01001110 01011010 01011010

01001010 01010110 01001101

www.nzz.ch · Fr. 4.00 · €3.50

Freitag, 8. Juni 2012 · Nr. 131 · 233. Jhg.



01000110 01101100 11111100
01000011 01101000 01110100 01101100 01101001 01011110 01100011 01110001 01100001 01101100 01001011 01101110 01001010 0010-

01000010 01100101
01110010 01101001

01100011 01101000 01110100 01100101

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

01110011 01100001

01101011 01100101 01100101 00100000 011-
01001 01101110 00100000 01010001 0111-
001 01110010 01100101 01100101 01101110
00001101 00001010 00001101 00001010
01010011 01101110 01101111 00101101 010-
0010 01000001 01011111 01100010 01100001
01100011 01101000 01110010 01010111 0101101
00100000 01010001 01100011 01101000 011-
00001 01110101 01110000 01101100 01100-
001 01101000 01111010 00100000

01100110 01100101

01110010 01101110 01100111 01100101 011-
01000 01100001 01101100 01101000 0110-
0101 01101100 00000101 00001010 00001101

01100101 01110001 00100000 01001101 011-
00001 01110001 01110001 01100001 01101-
011 01100101 01110001 00100000 01100011
01110100 01100001 01110000 01110100 011-
0010 01100001 01100110 01101011 0110110
01100000 01100001 01101110 00101110 001-
00100 01100100 01110001 01100101 001-
00000 00100100 01100101 01100111 01101001
01100101 01110010 01110101 0110110
01100111 00100000 01101101 01100001 011-
00010 01101000 01110100 01100101 00100-
000 01110101 01101110 01100101 01100101
0110110 01100001 01101110 01101110 011-
0100 01100101 00100000 10101011 01100100
01100101 01110010 01110000 01110111 011-
10010 01101001 01110001 01110000 01100101
0110110 1011011 00100000 01100100 011-
00001 01100110 11111100 01110010

00100000 01110110

01100101 01110010 01110001 01101110 011-
0100 01101111 01101111 01110010 01101100
01101100 01101001 01100011 01110000 001-
0110 00000101 00001010 00001011 000-
0010 00001010 11111100 01110010 011001-

Wertebereich des Typs int

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

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

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

Wertebereich des Typs `int`

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

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

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

Überlauf und Unterlauf

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

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

- Es gibt **keine Fehlermeldung!**

Der Typ `unsigned int`

- Wertebereich

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

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

Gemischte Ausdrücke

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

```
17 + 17u
```

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

Konversion

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

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

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.

Zahlen mit Vorzeichen

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

Vorzeichenbehaftete Zahlendarstellung

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

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

- Suche möglichst konsistente Lösung

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

Rechnen mit Binärzahlen (4 Stellen)

Einfache Addition

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

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

Einfache Subtraktion

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

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

Rechnen mit Binärzahlen (4 Stellen)

Addition mit Überlauf


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

Subtraktion mit Unterlauf

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

Warum das funktioniert

Modulo-Arithmetik: Rechnen im Kreis³



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

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

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

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

Negative Zahlen (3 Stellen)

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

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

Zweierkomplement

- Negation durch bitweise Negation und Addition von 1.

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

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

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

- Intuitive „Wrap-Around“ Konversion negativer Zahlen.

$$-n \rightarrow 2^B - n$$

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

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

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

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

AND(x, y)

 $x \wedge y$

- „Logisches Und“

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

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

x	y	AND(x, y)
0	0	0
0	1	0
1	0	0
1	1	1

Logischer Operator &&

`a && b` (logisches Und)

`bool` × `bool` → `bool`

R-Wert × R-Wert → R-Wert

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

OR(x, y)

 $x \vee y$

- „Logisches Oder“

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

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

x	y	OR(x, y)
0	0	0
0	1	1
1	0	1
1	1	1

Logischer Operator ||

`a || b` (logisches Oder)

`bool × bool → bool`

R-Wert × R-Wert → R-Wert

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

NOT(x)

 $\neg x$

- „Logisches Nicht“

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

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

x	NOT(x)
0	1
1	0

Logischer Operator !

!b (logisches Nicht)

bool → **bool**

R-Wert → R-Wert

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

Präzedenzen

$$\begin{array}{c} !b \ \&\& \ a \\ \Updownarrow \\ (!b) \ \&\& \ a \end{array}$$
$$\begin{array}{c} a \ \&\& \ b \ || \ c \ \&\& \ d \\ \Updownarrow \\ (a \ \&\& \ b) \ || \ (c \ \&\& \ d) \end{array}$$
$$\begin{array}{c} a \ || \ b \ \&\& \ c \ || \ d \\ \Updownarrow \\ a \ || \ (b \ \&\& \ c) \ || \ d \end{array}$$

Logische Operatoren: Tabelle

	Symbol	Stelligkeit	Präzedenz	Assoziativität
Logisches Und (AND)	<code>&&</code>	2	6	links
Logisches Oder (OR)	<code> </code>	2	5	links
Logisches Nicht (NOT)	<code>!</code>	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	$\text{XOR}(x, y)$
0	0	0
0	1	1
1	0	1
1	1	0

Vollständigkeit: XOR(x, y)

$x \oplus y$

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

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

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

Vollständigkeit Beweis

- Identifiziere binäre Boolesche Funktionen mit ihrem charakteristischen Vektor.

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

Charakteristischer Vektor: 0110

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

Vollständigkeit Beweis

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

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

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

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

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

Vollständigkeit Beweis

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

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

- 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
<i>true</i>	→	1
<i>false</i>	→	0

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

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

DeMorgansche Regeln

■ $!(a \ \&\& \ b) == (!a \ || \ !b)$

■ $!(a \ || \ b) == (!a \ \&\& \ !b)$

! (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
```

⇒ Keine Division durch 0

4. Defensives Programmieren

Konstanten und Assertions

Fehlerquellen

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

Der Compiler als Freund: Konstanten

Konstanten

- sind Variablen mit unveränderbarem Wert

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

- Verwendung: **const** vor der Definition

Der Compiler als Freund: Konstanten

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

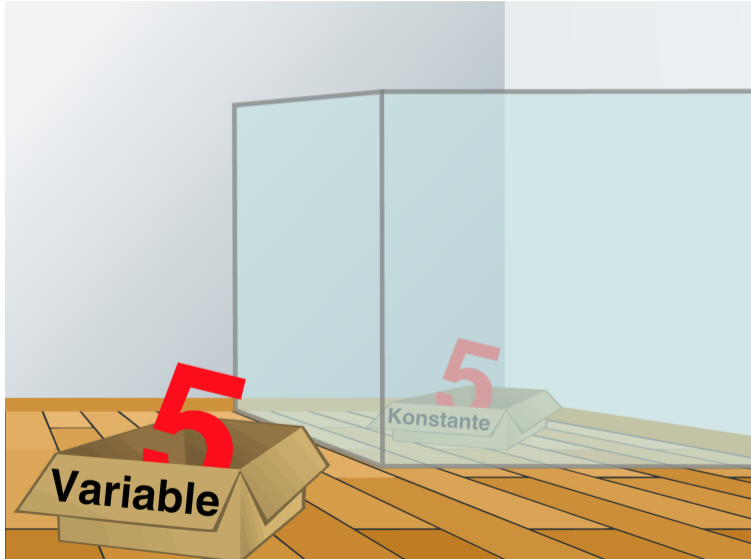
```
const int speed_of_light = 299792458;  
...  
speed_of_light = 300000000;
```

Compilerfehler!



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

Konstanten: Variablen hinter Glas



Die `const`-Richtlinie

Const-Richtlinie

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

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

Fehlerquellen vermeiden

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

Gegen Laufzeitfehler: *Assertions*

`assert (expr)`

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

Assertions für den $ggT(x, y)$

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

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

Eingabe der Argumente für
die Berechnung

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

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

← Vorbedingung für die weitere Berechnung

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

Assertions für den $ggT(x, y)$

... und hinterfrage das Offensichtliche! ...

...

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

← Vorbedingung für die weitere Berechnung

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

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

Verschiedene
Eigenschaften
des ggT
überprüfen

Assertions abschalten

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

```
...
```

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

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

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

```
...
```


Fail-Fast mit Assertions

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



5. Kontrollanweisungen I

Auswahanweisungen, Iterationsanweisungen, Terminierung,
Blöcke

Kontrollfluss

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

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

Auswahlanweisungen

realisieren Verzweigungen

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

if-Anweisung

```
if ( condition )  
    statement
```

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

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

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

if-else-Anweisung

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


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

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

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

Layout!

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



Einrückung

Einrückung

Iterationsanweisungen

realisieren Schleifen:

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

Berechne $1 + 2 + \dots + n$

```
// 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
```

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

Der kleine Gauß (1777 - 1855)

- Wie sie vermutlich wissen, gibt es einen effizienteren Weg, um die Summe der ersten n natürlichen Zahlen zu berechnen. Dazu folgende Anekdote:
- 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

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

- Das ist die Hälfte von

$$\begin{array}{r} 1 + 2 + \dots + 99 + 100 \\ + 100 + 99 + \dots + 2 + 1 \\ \hline = 101 + 101 + \dots + 101 + 101 \end{array}$$

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


for-Anweisung: Syntax

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

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

for-Anweisung: Semantik

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

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

for-Anweisung: Terminierung

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

Hier und meistens:

- *expression* ändert einen Wert, der in *condition* vorkommt.
- Nach endlich vielen Iterationen wird *condition* falsch:
Terminierung.

Endlosschleifen

- Endlosschleifen sind leicht zu produzieren:

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

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

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


Halteproblem

Unentscheidbarkeit des Halteproblems

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

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

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

Beispiel: Primzahltest

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

Eine Schleife, die das testet:

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

Primzahltest: Terminierung

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

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

Primzahltest: Korrektheit

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

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

Blöcke

- Blöcke gruppieren mehrere Anweisungen zu einer neuen Anweisung

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

- Beispiel: Rumpf der main Funktion

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

- Beispiel: Schleifenrumpf

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

6. Kontrollanweisungen II

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

Sichtbarkeit

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

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

Kontrollanweisung definiert Block

Kontrollanweisungen verhalten sich in diesem Zusammenhang wie Blöcke.

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


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;
}
```

Gültigkeit von i_1 in main
Gültigkeit von i_2 in for

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 (expression)  
  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. Computes 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 (condition);
```

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

do Anweisung

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

ist äquivalent zu

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

do-Anweisung: Semantik

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

- Iteration beginnt ←
 - *statement* wird ausgeführt.
 - *condition* wird ausgewertet
 - **true**: Iteration beginnt
 - **false**: do-Anweisung wird beendet.
-

do-Anweisung: Beispiel Taschenrechner

Summiere ganze Zahlen (bei 0 ist Schluss):

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


Zusammenfassung

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

Sprunganweisungen

- `break;`
- `continue;`

break-Anweisung

```
break;
```

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

⁵und 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;
    s += a; /* irrelevant in last iteration */
    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 *ohne* break

Version ohne `break` wertet `a != 0` zweimal aus (und benötigt zusätzlichen Block).

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

continue-Anweisung

```
continue;
```

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

break und continue in der Praxis

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

Taschenrechner mit `continue`

Ignoriere alle negativen Eingaben:

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

Äquivalenz von Iterationsanweisungen

Wir haben gesehen:

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

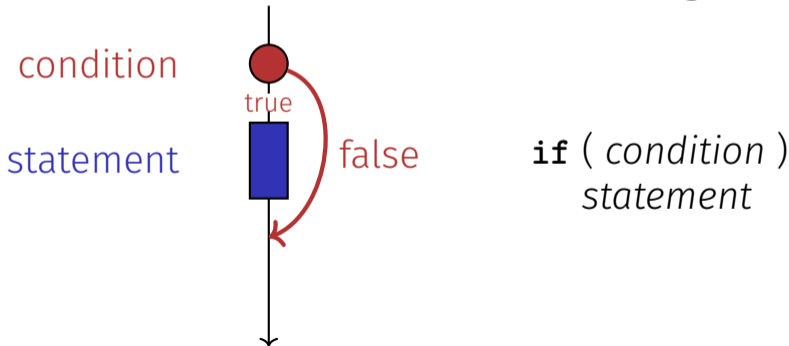
Es gilt aber sogar:

- Alle drei Iterationsanweisungen haben die gleiche „Ausdruckskraft“ (Skript)
- Nicht ganz so einfach falls ein `continue` im Spiel ist

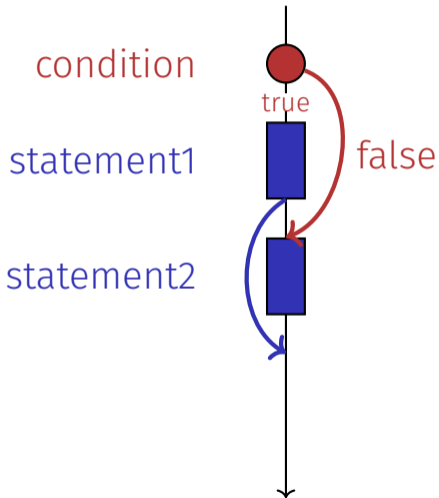
Kontrollfluss

Reihenfolge der (wiederholten) Ausführung von Anweisungen

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



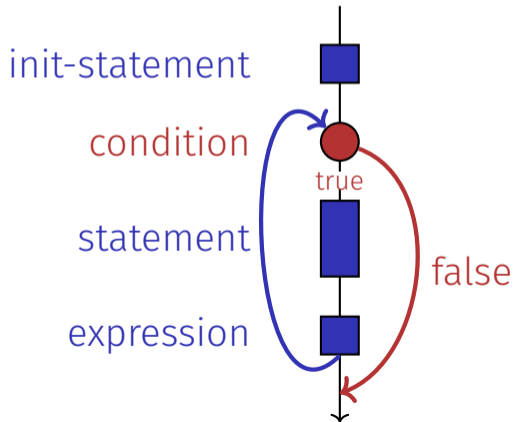
Kontrollfluss if else



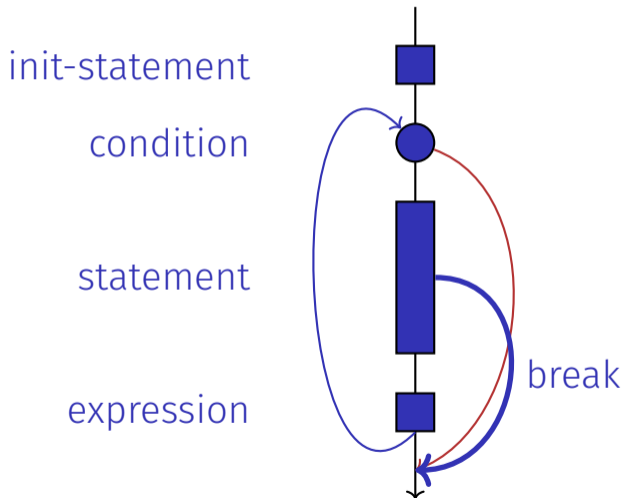
```
if ( condition )  
    statement1  
else  
    statement2
```

Kontrollfluss `for`

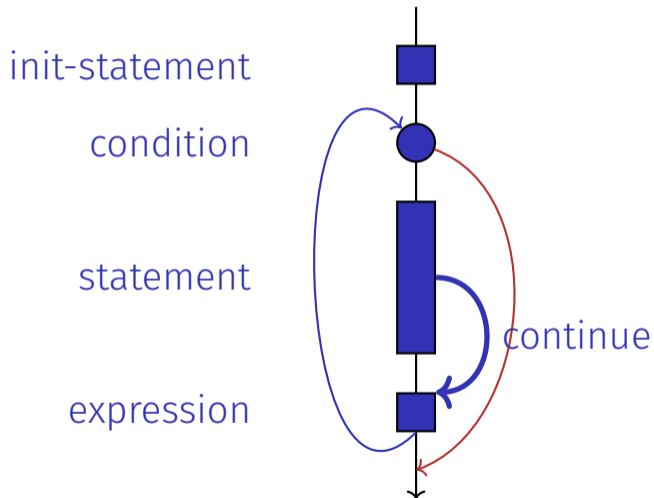
`for` (*init statement* *condition* ; *expression*)
statement



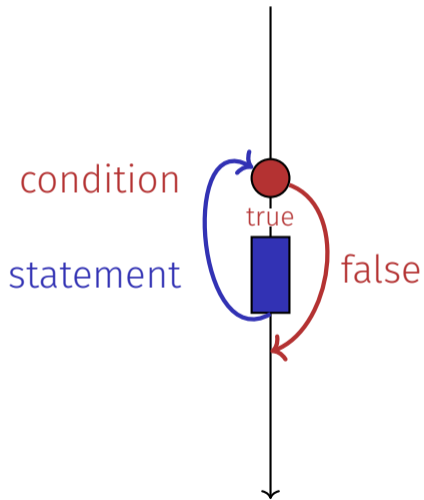
Kontrollfluss **break** in for



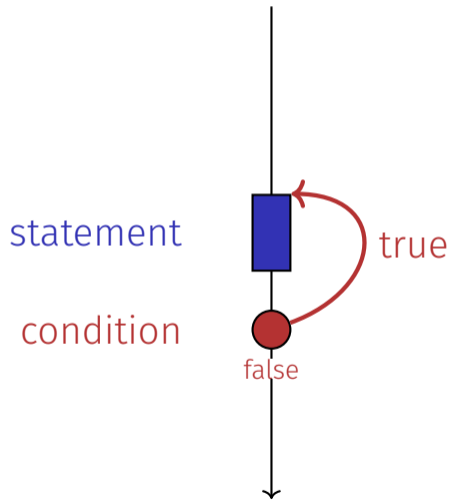
Kontrollfluss `continue` in `for`



Kontrollfluss while



Kontrollfluss do while



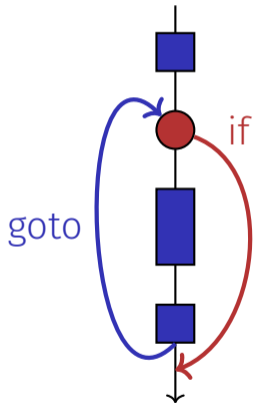
Kontrollfluss: Die guten alten Zeiten?

Beobachtung

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

Sprachen, die darauf basieren:

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



BASIC und die Home-Computer...

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

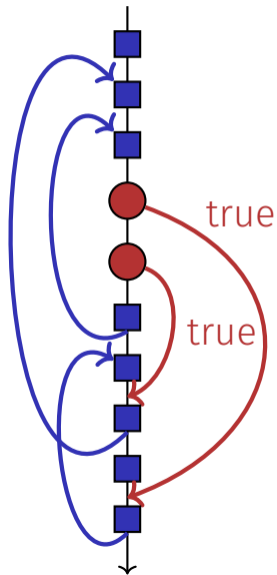


Home-Computer Commodore C64 (1982)

Spaghetti-Code mit goto

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

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



Die „richtige“ Iterationsanweisung

Ziele: Lesbarkeit, Prägnanz. Insbesondere

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

Ziele sind oft nicht gleichzeitig erreichbar.

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

Erster (korrekter) Versuch:

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

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

Weniger Anweisungen, **weniger** Zeilen:

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


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

Weniger Anweisungen, **einfacherer** Kontrollfluss:

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

Das ist hier die „richtige“ Iterationsanweisung

Sprunganweisungen

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

Notenausgabe

1. Funktionale Anforderung:

6 → "Excellent ... You passed!"

5,4 → "You passed!"

3 → "Close, but ... You failed!"

2,1 → "You failed!"

sonst → "Error!"

2. Ausserdem: Text- und Codeduplikation vermeiden

Notenausgabe mit `if`-Anweisungen

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

Nachteil: Kontrollfluss – und somit Programmverhalten – nicht gerade offensichtlich

Notenausgabe mit `switch`-Anweisung

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

← Springe zu passendem `case`

↓ Durchfallen

← Verlasse `switch`

↓ Durchfallen

← Verlasse `switch`

← In allen anderen Fällen

Vorteil: Kontrollfluss klar erkennbar

Die `switch`-Anweisung

```
switch (expression)  
statement
```

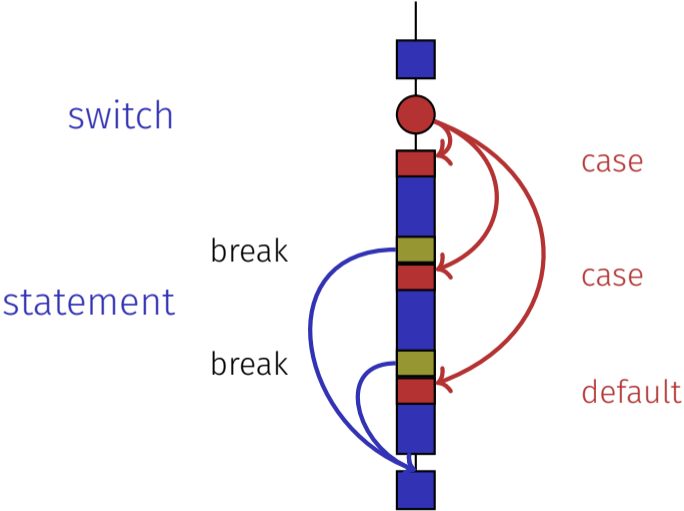
- *expression*: Ausdruck, konvertierbar in einen integralen Typ
- *statement* : beliebige Anweisung, in welcher **case** und **default**-Marken erlaubt sind, **break** hat eine spezielle Bedeutung.
- Benutzung des Durchfallens in der Praxis umstritten, Einsatz gut abwägen (entsprechende Compilerwarnung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

Semantik der `switch`-Anweisung

```
switch (expression)  
    statement
```

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

Kontrollfluss switch



7. Fließkommazahlen I

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

„Richtig“ Rechnen

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

#include <iostream>

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

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

Fixkommazahlen

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

0.0824 = 0000000.082 ← dritte Stelle abgeschnitten

Nachteile

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

Fliesskommazahlen

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

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

Anzahl *signifikanter Stellen* bleibt konstant

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

Typen `float` und `double`

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

Arithmetische Operatoren

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

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

Literale

unterscheiden sich von Ganzzahlliteralen durch Angabe von

■ Dezimalkomma

`1.0` : Typ `double`, Wert 1

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

■ und / oder Exponent.

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

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

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

1.23e-7f



ganzzahliger Teil

Exponent

fraktionaler Teil

Rechnen mit `float`: Beispiel

Approximation der Euler-Zahl

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

mittels der ersten 10 Terme.

Rechnen mit float: Eulersche Zahl

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

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

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

Rechnen mit `float`: Eulersche Zahl

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

Gemischte Ausdrücke, Konversion

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

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

Löcher im Wertebereich

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

Eingabe 1.1

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

Eingabe 1.0

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

Eingabe 0.1

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

Ausgabe 2.23517e-8

Was ist denn hier los?

Wertebereich

Ganzzahlige Typen:

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

Fliesskommatypen:

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

8. Fließkommazahlen II

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

Fliesskommazahlensysteme

Ein Fließkommazahlensystem ist durch vier natürliche Zahlen definiert:

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

Bezeichnung:

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

Fliesskommazahlensysteme

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

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

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

In Basis- β -Darstellung:

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

Fliesskommazahlensysteme

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

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

Unterschiedliche Darstellungsmöglichkeiten durch Wahl des Exponenten

Normalisierte Darstellung

Normalisierte Zahl:

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

Bemerkung 1

Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen.

Bemerkung 2

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

Menge der normalisierten Zahlen

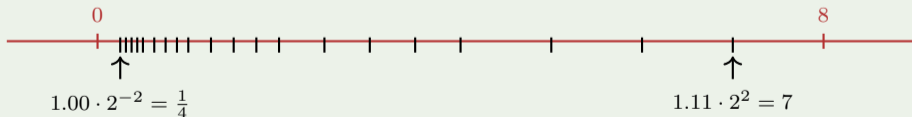
$$F^*(\beta, p, e_{\min}, e_{\max})$$

Normalisierte Darstellung

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

(nur positive Zahlen)

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



Binäre und dezimale Systeme

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

Umrechnung dezimal \rightarrow binär

Angenommen, $0 < x < 2$.

Binärdarstellung:

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

Umrechnung dezimal \rightarrow binär

Angenommen, $0 < x < 2$.

■ Also: $x' = b_{-1}b_{-2}b_{-3}b_{-4}\dots = 2 \cdot (x - b_0)$

■ 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}, \dots :

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

Binärdarstellung von 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = 1$	0.1	0.2
0.2	$b_1 = 0$	0.2	0.4
0.4	$b_2 = 0$	0.4	0.8
0.8	$b_3 = 0$	0.8	1.6
1.6	$b_4 = 1$	0.6	1.2
1.2	$b_5 = 1$	0.2	0.4

⇒ $1.000\overline{11}$, 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:

$$\begin{aligned} \mathbf{1.1} &= \underline{1.10000000000000000000}888178\dots \\ \mathbf{1.1f} &= \underline{1.1000000}238418\dots \end{aligned}$$

Rechnen mit Fließkommazahlen

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

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 1.011 \cdot 2^{-1} \\ \hline = 1.001 \cdot 2^0 \end{array}$$

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 und wird fast überall benutzt

- Single precision (**float**) Zahlen:

$$F^*(2, 24, -126, 127) \text{ (32 bit)} \quad \text{plus } 0, \infty, \dots$$

- Double precision (**double**) Zahlen:

$$F^*(2, 53, -1022, 1023) \text{ (64 bit)} \quad \text{plus } 0, \infty, \dots$$

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

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, ∞ ,...)
- ⇒ 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, ∞ ,...)

⇒ insgesamt 64 Bit.

Beispiel: 32-bit Darstellung einer Fließkommazahl



± Exponent

Mantisse

± $2^{-126}, \dots, 2^{127}$
 $0, \infty, \dots$

1.000000000000000000000000000000
...
1.111111111111111111111111111111

Regel 1

Teste keine gerundeten Fließkommazahlen auf Gleichheit!

```
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!

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

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
```

Beobachtung:

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

Erklärung:

- Bei $1 + 1/2 + 1/3 + \dots$ sind späte Terme zu klein, um noch beizutragen.
- Problematik wie bei $2^5 + 1 \text{ „=“ } 2^5$

Regel 3

Subtrahiere keine zwei Zahlen sehr ähnlicher Grösse!

Auslöschungsproblematik, siehe Skript.

Literatur

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



Randy Glasbergen, 1996

9. Funktionen I

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

Funktionen

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

 "Funktion pow"

```
std::cout << a << "^" << n << " = " << result << ".\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

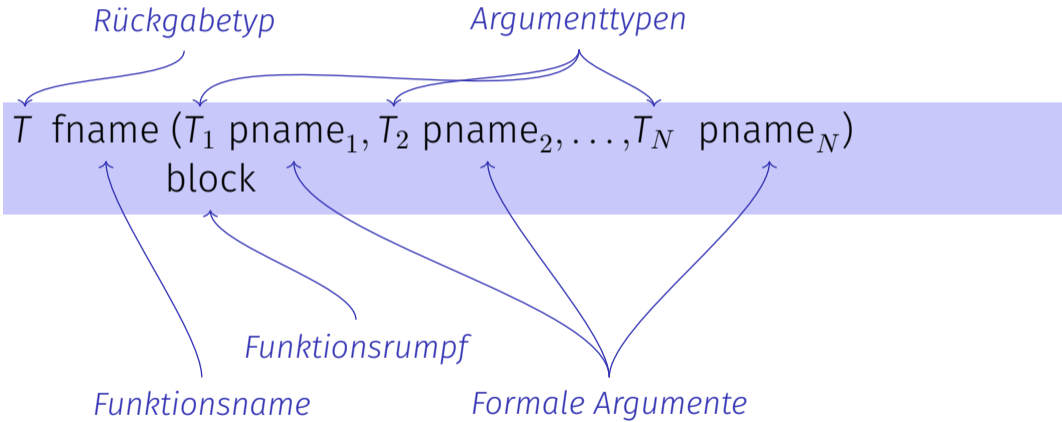
```
// Prog: callpow.cpp
// Define and call a function for computing powers.
#include <iostream>
```

```
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

$fname (expression_1, expression_2, \dots, expression_N)$

- Alle Aufrufargumente müssen konvertierbar sein in die entsprechenden Argumenttypen.
- Der Funktionsaufruf selbst ist ein Ausdruck vom Rückgabebetyp. Wert und Effekt wie in der Nachbedingung der Funktion *fname* angegeben.

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

Funktionsaufrufe

Für die Typen, die wir bisher kennen, gilt:

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

$fname: \text{R-Wert} \times \text{R-Wert} \times \dots \times \text{R-Wert} \longrightarrow \text{R-Wert}$

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

Aufruf von pow

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

Rückgabe

...

pow (2.0, -2)

Formale Funktionsargumente^a

^amanchmal *formale Parameter*

- 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)

Gültigkeit formaler Argumente

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

```
int main(){  
    double b = 2.0;  
    int e = -2;  
    double z = pow(b, e);  
  
    std::cout << z; // 0.25  
    std::cout << b; // 2  
    std::cout << e; // -2  
    return 0;  
}
```

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

Der Typ void

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

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


Der Typ `void`

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

void-Funktionen

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

Ausdruck vom Typ **void** (z.B. Aufruf einer Funktion mit Rückgabetyt **void**)

Funktionen und return

Das Verhalten einer Funktion mit Rückgabetyt ungleich **void** ist **undefiniert** wenn das Ende des Funktionsrumpfes ohne **return** Anweisung erreicht wird.

Falsch:

```
bool compare(float x, float y) {  
    float delta = x - y;  
    if (delta*delta < 0.001f) return true;  
}
```

Der Wert von `compare(10,20)` ist hier undefiniert

Funktionen und return

Das Verhalten einer Funktion mit Rückgabebetyp ungleich **void** ist **undefiniert** wenn das Ende des Funktionsrumpfes ohne **return** Anweisung erreicht wird.

Besser:

```
bool compare(float x, float y) {  
    float delta = x - y;  
    if (delta*delta < 0.001f)  
        return true;  
    else  
        return false;  
}
```

Alle Ausführungspfade erreichen ein **return**

Funktionen und return

Das Verhalten einer Funktion mit Rückgabetyt ungleich **void** ist **undefiniert** wenn das Ende des Funktionsrumpfes ohne **return** Anweisung erreicht wird.

Noch besser (weil einfacher)

```
bool compare(float x, float y) {  
    float delta = x - y;  
    return delta*delta < 0.001f;  
}
```

10. Funktionen II

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

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 *Definitionsereich* der Funktion.

0^e ist für $e < 0$ undefiniert

```
// PRE: e >= 0 || b != 0.0
```


Nachbedingungen

Nachbedingung (postcondition):

- Was gilt nach Funktionsaufruf?
- Spezifiziert *Wert* und *Effekt* des Funktionsaufrufes.

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 be
```

ist formal inkorrekt:

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

Fromme Lügen...sind erlaubt.

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

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

Prüfen von Vorbedingungen...

- Vorbedingungen sind nur Kommentare.
- Wie können wir *sicherstellen*, dass sie beim Funktionsaufruf gelten?

...mit Assertions

```
#include <cassert>
...
// PRE: e >= 0 || b != 0.0
// POST: return value is b^e
double pow(double b, int e) {
    assert (e >= 0 || b != 0);
    double result = 1.0;
    ...
}
```


Nachbedingungen mit Assertions

- Das Ergebnis „komplizierter“ Berechnungen ist oft einfach zu prüfen.
- Dann lohnt sich der Einsatz von `assert` für die Nachbedingung

```
// PRE: the discriminant p*p/4 - q is nonnegative
// POST: returns larger root of the polynomial x^2 + p x + q
double root(double p, double q)
{
    assert(p*p/4 >= q); // precondition
    double x1 = - p/2 + sqrt(p*p/4 - q);
    assert(equals(x1*x1+p*x1+q,0)); // postcondition
    return x1;
}
```

Ausnahmen (Exception Handling)

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

Stepwise Refinement

Einfache Programmier- technik zum Lösen komplexer Probleme

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

Education P. Wegner
Editor

Program Development by Stepwise Refinement

Niklaus Wirth
Eidgenössische Technische Hochschule
Zürich, Switzerland

The creative activity of programming—to be distinguished from coding—is usually taught by examples serving to exhibit certain techniques. It is here considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures. The process of successive refinement of specifications is illustrated by a short but nontrivial example, from which a number of conclusions are drawn regarding the art and the instruction of programming.

Key Words and Phrases: education in programming, programming techniques, stepwise program construction
CR Categories: 1.50, 4.0

1. Introduction

Programming is usually taught by examples. Experience shows that the success of a programming course critically depends on the choice of these examples. Unfortunately, they are too often selected with the prime intent to demonstrate what a computer can do. Instead, a main criterion for selection should be their suitability to exhibit certain widely applicable techniques. Furthermore, examples of programs are commonly presented as finished "products" followed by explanations of their purpose and their linguistic details. But active programming consists of the design of new programs, rather than contemplation of old programs. As a consequence of these teaching methods, the student obtains the impression that programming consists mainly of mastering a language (with all the peculiarities and intricacies so abundant in modern PL's) and relying on one's intuition to somehow transform ideas into finished programs. Clearly, programming courses should teach methods of design and construction, and the selected examples should be such that a gradual development can be nicely demonstrated.

This paper deals with a single example chosen with

these two purposes in mind. Some well-known techniques are briefly demonstrated and motivated (strategy of preselection, stepwise construction of trial solutions, introduction of auxiliary data, recursion), and the program is gradually developed in a sequence of refinement steps.

In each step, one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of an underlying computer or programming language, and must therefore be guided by the facilities available on that computer or language. The result of the execution of a program is expressed in terms of data, and it may be necessary to introduce further data for communication between the obtained subtasks or instructions. As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine program and data specifications in parallel.

Every refinement step implies some design decisions. It is important that these decisions be made explicit, and that the programmer be aware of the underlying criteria and of the existence of alternative solutions. The possible solutions to a given problem emerge as the leaves of a tree, each node representing a point of deliberation and decision. Subtrees may be considered as families of solutions with certain common characteristics and structures. The notion of such a tree may be particularly helpful in the situation of changing purpose and environment to which a program may sometime have to be adapted.

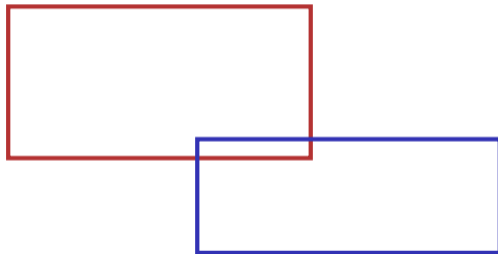
A guideline in the process of stepwise refinement should be the principle to decompose decisions as much as possible, to untangle aspects which are only seemingly interdependent, and to defer those decisions which concern details of representation as long as possible. This

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)

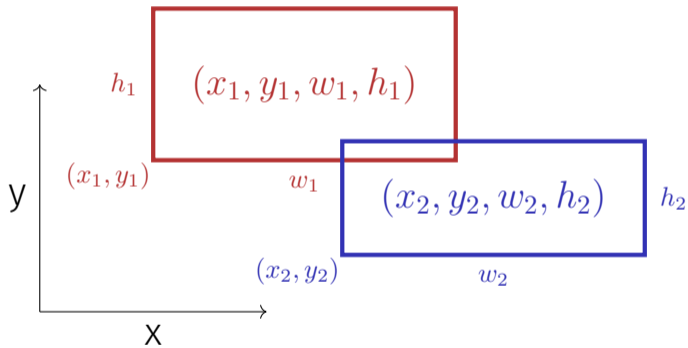
```
int main()
{
    // Eingabe Rechtecke

    // Schnitt?

    // Ausgabe der Loesung

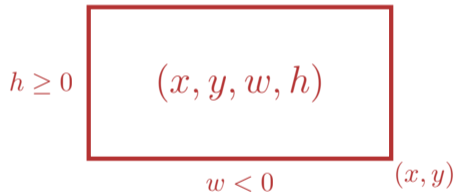
    return 0;
}
```

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 ✓

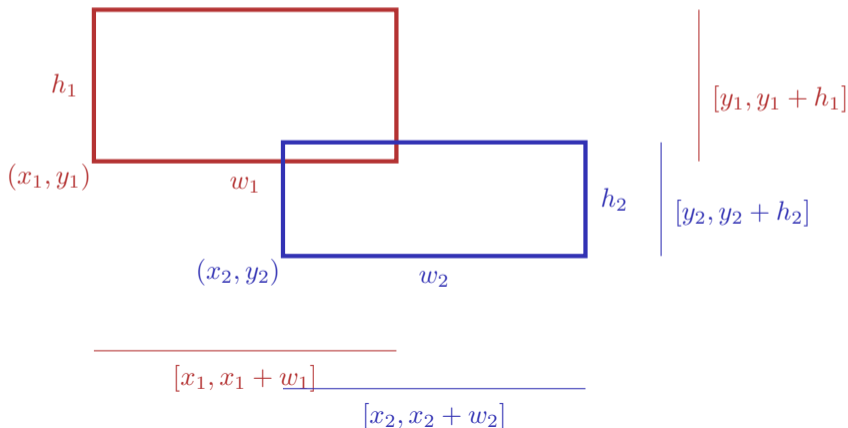
Verfeinerung 3:

...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;
}
```

gibt es schon in der Standardbibliothek

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

Funktion intervals_intersect ✓

Funktion rectangles_intersect ✓

Funktion main ✓

Nochmal zurück zu Intervallen

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

Das haben wir schrittweise erreicht!

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

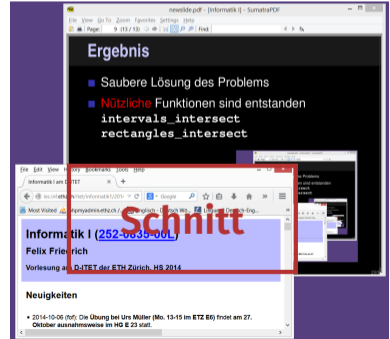
// PRE: [a1, b1], [a2, h2] are (generalized) intervals,
//      with [a,b] := [b,a] if a>b
// POST: returns true if [a1, b1],[a2, b2] intersect
bool intervals_intersect(int a1, int b1, int a2, int b2)
{
    return std::max(a1, b1) >= std::min(a2, b2)
        && std::min(a1, b1) <= std::max(a2, b2);
}

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

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

Ergebnis

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



Wo darf man eine Funktion benutzen?

```
#include <iostream>
```

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

```
Gültigkeit f  
↓  
int f(int i) // Gültigkeitsbereich von f ab hier  
{  
    return i;  
}
```

Gültigkeitsbereich einer Funktion

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

```
Gültigkeit f
↓
int f(int i) // Gültigkeitsbereich von f ab hier
{
    return i;
}
```


...aber so!

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

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

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

Forward Declarations, wozu?

Funktionen, die sich gegenseitig aufrufen:

The diagram illustrates the scope of forward declarations for two mutually recursive functions, `g` and `f`. A blue vertical line on the left, labeled "Gültigkeit g", spans the entire code block, indicating that the forward declaration of `g` is valid from the top to the bottom. A red vertical line, labeled "Gültigkeit f", starts at the beginning of the `int f(...)` function definition and ends at its closing brace, indicating that the forward declaration of `f` is only valid within the scope of `f`.

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

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

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

Wiederverwendbarkeit

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

Level 1: Auslagern der Funktion

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

Level 1: Inkludieren der Funktion

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

```
#include <iostream>
```

```
#include "mymath.cpp" ← Datei im Arbeitsverzeichnis
```

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

    return 0;
}
```

Nachteil des Inkludierens

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

Level 2: Getrennte Übersetzung

von `mymath.cpp` unabhängig vom Hauptprogramm:

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

`mymath.cpp`

`g++ -c mymath.cpp`

```
001110101100101010  
000101110101000111  
000101110101000111  
111100001101010001  
111111101000111010  
010101101011010001  
100101111100101010
```

`mymath.o`

Level 2: Getrennte Übersetzung

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

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

mymath.h

Level 2: Getrennte Übersetzung

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

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

callpow3.cpp



```
001110101100101010
000101110101000111
000101110101000111
Funktion main
111100001101010001
010101101011010001
101010110101101010
rufe pow auf!
111111101000111010
```

callpow3.o

Der Linker vereint...

```
001110101100101010
000101110101000111
000101 Funktion pow
111100001101010001
111111101000111010
010101101011010001
100101111100101010
```

mymath.o

+

```
001110101100101010
000101110101000111
000101 Funktion main
111100001101010001
010101101011010001
100101111100101010
111111101000111010
```

callpow3.o

... was zusammengehört

```
001110101100101010
000101110101000111
000101 Funktion pow
111100001101010001
111111101000111010
010101101011010001
100101111100101010
```

mymath.o

+

```
001110101100101010
000101110101000111
000101 Funktion main
111100001101010001
010101101011010001
100101 rufe pow auf!
111111101000111010
```

callpow3.o

=

```
001110101100101010
000101110101000111
000101 Funktion pow
111100001101010001
111111101000111010
010101101011010001
100101111100101010
001110101100101010
000101110101000111
000101 Funktion main
111100001101010001
010101101011010001
100101 rufe addr auf!
111111101000111010
```

Ausführbare Datei callpow3

Verfügbarkeit von Quellcode?

Beobachtung

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

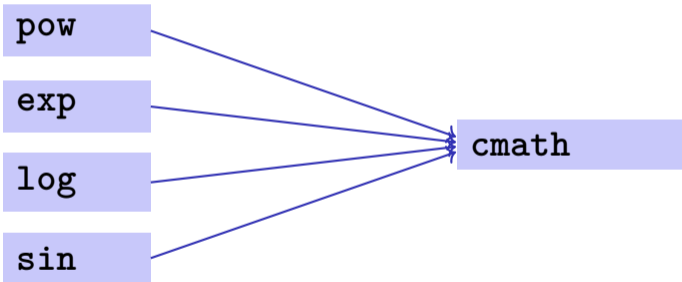
Viele Anbieter von Funktionsbibliotheken liefern dem Benutzer keinen Quellcode.
Header-Dateien sind dann die *einzigsten* lesbaren Informationen.

Open-Source-Software

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

Bibliotheken

- Logische Gruppierung ähnlicher Funktionen



Namensräume...

```
// cmath
namespace std {

    double pow(double b, int e);

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

...vermeiden Namenskonflikte

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

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


Namensräume / Kompilationseinheiten

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

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

Funktionen aus der Standardbibliothek

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

Beispiel: Primzahltest mit sqrt

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

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

Primzahltest mit `sqrt`

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

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

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

Primzahltest mit sqrt

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

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

    // Computation: test possible divisors d up to sqrt(n)
    unsigned int bound = std::sqrt(n);
```

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&`)

11. Referenztypen

Referenztypen: Definition und Initialisierung, Pass by Value, Pass by Reference, temporäre Objekte, Const-Referenzen

Swap!

```
// POST: values of x and y have been exchanged
```

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

```
int main() {  
    int a = 2;  
    int b = 1;  
    swap(a, b);  
    assert(a == 1 && b == 2); // ok! 😊  
}
```

Referenztypen

- Wir können Funktionen in die Lage versetzen, die Werte ihrer Aufrufargumente zu ändern
- 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 ...
- ...aber Initialisierung und Zuweisung funktionieren anders

Anakin Skywalker alias Darth Vader

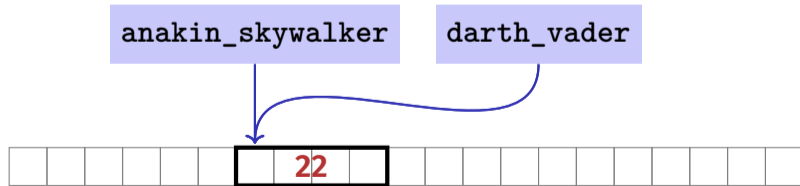


Anakin Skywalker alias Darth Vader

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

Zuweisung an den L-Wert hinter dem Alias

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



Referenztypen: Initialisierung & Zuweisung

```
int& darth_vader = anakin_skywalker;  
darth_vader = 22; // Effekt: anakin_skywalker = 22
```

- Eine Variable mit **Referenztyp** (eine *Referenz*) muss mit einem **L-Wert** initialisiert werden
- Die Variable wird dabei ein *Alias* des **L-Werts** (ein anderer Name für das referenzierte Objekt)
- Zuweisung an die Referenz erfolgt an das Objekt *hinter* dem Alias

Referenztypen: Realisierung

Intern wird ein Wert vom Typ $T\&$ durch die Adresse eines Objekts vom Typ T repräsentiert.

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

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

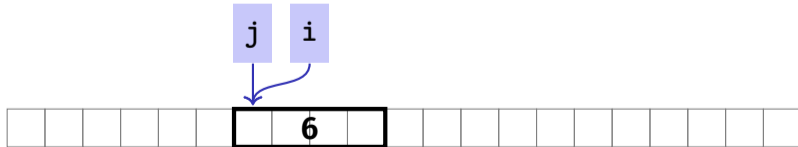

Pass by Reference

Referenztypen erlauben Funktionen, die Werte ihrer Aufrufargumente zu ändern

```
void increment (int& i) ← {  
    ++i;  
}
```

Initialisierung der formalen Argumente: **i** wird Alias des Aufrufarguments **j**

```
...  
int j = 5;  
increment (j);  
std::cout << j; // 6
```



Pass by Reference

Formales Argument *hat* Referenztyp:

⇒ *Pass by Reference*

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

Pass by Value

Formales Argument *hat keinen* Referenztyp:

⇒ *Pass by Value*

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

Referenzen im Kontext von intervals_intersect

```
// PRE: [a1, b1], [a2, b2] are (generalized) intervals,  
// POST: returns true if [a1, b1], [a2, b2] intersect, in which case  
// [l, h] contains the intersection of [a1, b1], [a2, b2]
```

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



```
...
```

```
int lo = 0; int hi = 0;
```

```
if (intervals_intersect(lo, hi, 0, 2, 1, 3)) // Initialisierung  
    std::cout << "[" << lo << "," << hi << "]" << "\n"; // [1,2]
```

Referenzen im Kontext von intervals_intersect

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

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

Return by Reference

- Auch der Rückgabebetyp einer Funktion kann ein Referenztyp sein: *Return by Reference*

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

- Aufruf **inc(x)**, für eine **int**-Variable **x**, hat exakt die Semantik des Prä-Inkrement **++x**
- Funktionsaufruf *selbst* ist nun ein L-Wert
- Daher möglich: **inc(inc(x))** oder **++(inc(x))**

Temporäre Objekte

Was ist hier falsch?

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

Rückgabewert vom Typ `int&` wird Alias des formalen Arguments (lokale Variable `i`), dessen Speicherdauer aber nach Auswertung des Funktionsaufrufes endet

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

Die Referenz-Richtlinie

Referenz-Richtlinie

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

Const-Referenzen

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

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

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

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

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

Was genau ist konstant?

Betrachte L-Wert vom Typ `const T`. **Fall: 1** *T ist kein Referenztyp.*

⇒ Dann ist der *L-Wert eine Konstante*

```
const int n = 5;  
int& a = n; // Compilerfehler: const-qualification discarded  
a = 6;
```

Unser *Schummelversuch* wird vom Compiler erkannt

Was genau ist konstant?

Betrachte L-Wert vom Typ `const T`. **Fall 2:** *T* ist Referenztyp.

⇒ Dann ist der *L-Wert ein Lese-Alias*, durch den der L-Wert *dahinter* nicht verändert werden darf.

```
int n = 5;

const int& r = n; // r ist Lese-Alias von n
r = 6;           // Compilerfehler: read-only reference

int& rw = n;     // rw ist Lese-Schreib-Alias
rw = 6;         // OK
```

Wann `const T&` verwenden?

```
void f_1(T& arg);
```

```
void f_2(const T& arg);
```

- Argumenttypen sind Referenzen; Aufrufargumente werden daher nicht kopiert, was effizient ist
- Aber nur `f_2` „verspricht“ Argument nicht zu verändern

Regel

Funktionsargumenttypen, falls möglich, als `const T&` (*pass by read-only reference*) deklarieren: effizient *und* sicher.

Lohnt sich i.d.R. nicht für fundamentale Typen (`int`, `double`, ...). Wir lernen später in der Vorlesung Typen kennen, die mehr Speicherplatz benötigen.

12. Vektoren I

Vektoren, Sieb des Eratosthenes, Speicherlayout, Iteration

Vektoren: Motivation

- Wir können jetzt über Zahlen iterieren

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

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

Vektoren: erste Anwendung

Das Sieb des Erathostenes

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

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

Am Ende des Streichungsprozesses bleiben nur die Primzahlen übrig.

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

Sieb des Eratosthenes mit Vektoren

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

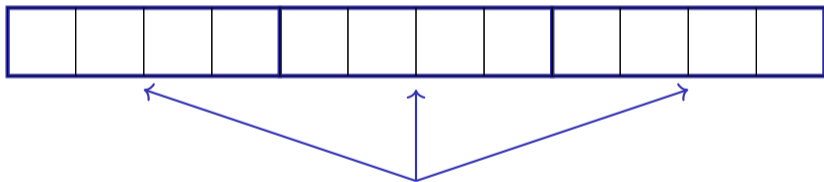
    // definition and initialization: provides us with Booleans
    // crossed_out[0],..., crossed_out[n-1], initialized to false
    std::vector<bool> crossed_out (n, false);

    // computation and output
    std::cout << "Prime numbers in {2,...," << n-1 << "}: \n";
    for (unsigned int i = 2; i < n; ++i)
        if (!crossed_out[i]) { // i is prime
            std::cout << i << " ";
            // cross out all proper multiples of i
            for (unsigned int m = 2*i; m < n; m += i) crossed_out[m] = true;
        }
    std::cout << "\n";
    return 0;
}
```


Speicherlayout eines Vektors

Ein Vektor belegt einen *zusammenhängenden* Speicherbereich

Beispiel: Ein Vektor mit 3 Elementen vom Type **T**



Speichersegmente für jeweils einen Wert vom Typ **T**
(**T** belegt z.B. 4 Bytes)

Wahlfreier Zugriff (Random Access)

Gegeben

- Vektor **vec** mit **T**-Elementen
- **int**-Ausdruck **exp** mit Wert $i \geq 0$

Dann ist der Ausdruck

vec [exp]

- ein *L*-Wert vom Type **T**
- der sich auf das *i*-te Element von **vec** bezieht (Zählung ab 0!)



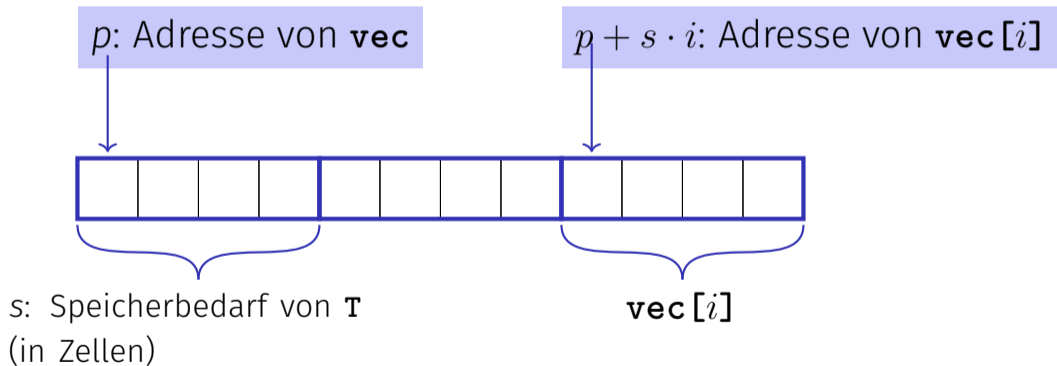
Wahlfreier Zugriff (Random Access)

`vec [exp]`

- Der Wert i von `exp` heisst *Index*
- `[]` ist der *Index-Operator* (auch *Subskript-Operator*)

Wahlfreier Zugriff (Random Access)

Wahlfreier Zugriff ist sehr effizient:



Vektorinitialisierung

- `std::vector<int> vec(5);`

Die 5 Elemente von `vec` werden mit Nullen initialisiert

- `std::vector<int> vec(5, 2);`

Die 5 Elemente von `vec` werden mit 2 initialisiert

- `std::vector<int> vec{4, 3, 5, 2, 1};`

Der Vektor wird mit einer *Initialisierungsliste* initialisiert

- `std::vector<int> vec;`

Ein leerer Vektor wird initialisiert

Achtung

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

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

Achtung

Prüfung der Indexgrenzen

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

Vektoren bieten viel Funktionalität

Hier ein paar Beispielfunktionen, weitere folgen später in der Vorlesung.

```
std::vector<int> v(10);  
std::cout << v.at(10);  
    // Zugriff mit Index-Check → Laufzeitfehler  
    // Ideal für Hausaufgaben  
  
v.push_back(-1); // -1 is appended (added at end)  
std::cout << v.size(); // outputs 11  
std::cout << v.at(10); // outputs -1
```


13. Zeichen und Texte I

Zeichen und Texte, ASCII, UTF-8, Caesar-Code

Zeichen und Texte

- Texte haben wir schon gesehen:

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

String-Literal

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

Zeichen: Wert des fundamentalen Typs **char**

Text: **std::string** \approx Vektor von **char** Elementen

Der Typ `char` („character“)

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

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

Deklariert und initialisiert Variable `c` vom Typ `char` mit Wert `'a'`

Literal vom Typ `char`

Der Typ `char` („character“)

Ist formal ein ganzzahliger Typ

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

Wertebereich:

$\{-128, \dots, 127\}$ oder $\{0, \dots, 255\}$

Der ASCII-Code

- Definiert konkrete Konversionsregeln **char** \longrightarrow **(unsigned) int**

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

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

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

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

- Wird von allen gängigen Computersystemen unterstützt
- Erlaubt Arithmetik über Zeichen

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

Erweiterung von ASCII: Unicode, UTF-8

- Internationalisierung von Software \Rightarrow grosse Zeichensätze nötig. Heute daher üblich:
 - Zeichensatz *Unicode*: 150 Schriftsysteme, ca. 137'000 Zeichen
 - Codierungsstandard *UTF-8*: Abbildung Zeichen \leftrightarrow Zahlen
- UTF-8 ist eine *Multibyte-Codierung*: Häufig genutzte Zeichen (z.B. lat. Alphabet) belegen nur ein Byte, andere Zeichen bis zu vier
- Länge einer Zeichen-Bytefolge wird dabei durch Bitmuster codiert

Nutzbare Bits	Bitmuster
7	0xxxxxxx
11	110xxxxx 10xxxxxx

Einige Unicode-Zeichen in UTF-8

Symbol	Codierung (jeweils 16 Bit)
س	11101111 10101111 10111001
☠	11100010 10011000 10100000
☎	11100010 10011000 10000011
☪	11100010 10011000 10011001
A	01000001

P.S.: Suchen Sie mal nach **apple "unicode of death"** P.S.: Unicode & UTF-8 sind nicht klausurrelevant

Caesar-Code

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

' ' (32) → '|' (124)

'!' (33) → '}' (125)

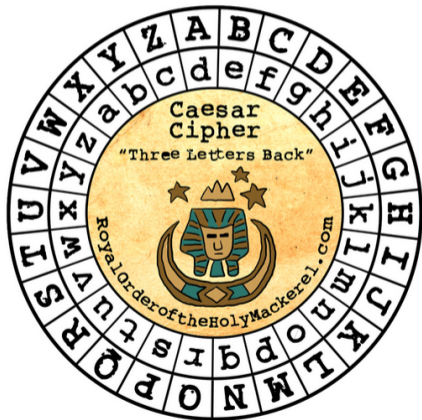
...

'D' (68) → 'A' (65)

'E' (69) → 'B' (66)

...

~ (126) → '{' (123)



Caesar-Code:

shift-Funktion

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

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

    return c;
}
```

"- 32" transforms interval [32, 126] to [0, 94]
"mod(x, 95)" computes $x \bmod 95$ in [0, 94]

Caesar-Code:

caesar-Funktion

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

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

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

Verschiebung druckbarer Zeichen um **s**

Caesar-Code:

Hauptprogramm

```
int main() {  
    int s;  
    std::cin >> s;  
  
    // Shift input by s  
    caesar(s);  
  
    return 0;  
}
```

Verschlüsseln: Verschiebung um n (hier: 3)

```
3.  
Hello World, my password is 1234.  
Koor#Zruog/#p|#sdvvzrug#lv#45671
```

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

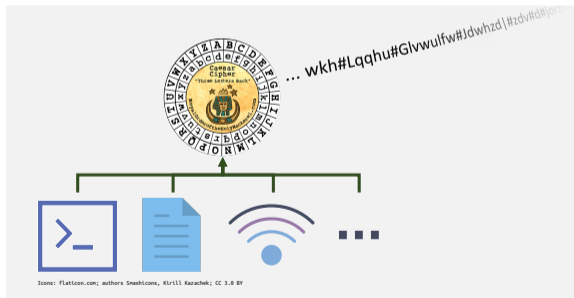
```
-3.  
Koor#Zruog/#p|#sdvvzrug#lv#45671  
Hello World, my password is 1234.
```

Caesar-Code: Generalisierung

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

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

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



14. Zeichen und Texte II

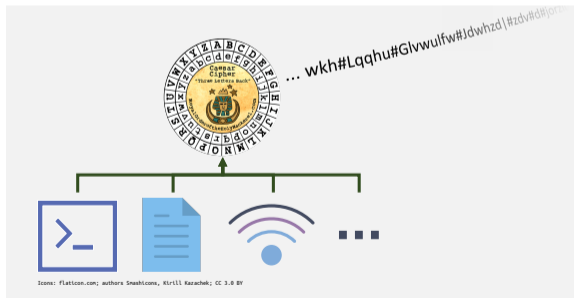
Caesar-Code mit Streams, Texte als Strings,
String-Operationen

Caesar-Code: Generalisierung

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

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

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



Einschub: Abstrakte vs. konkrete Typen

DestroyBox



(abstrakt,
generisch)

(ist eine)



ShredBox

FireBox

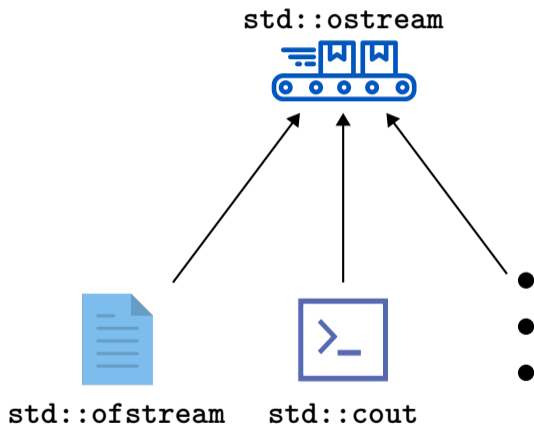
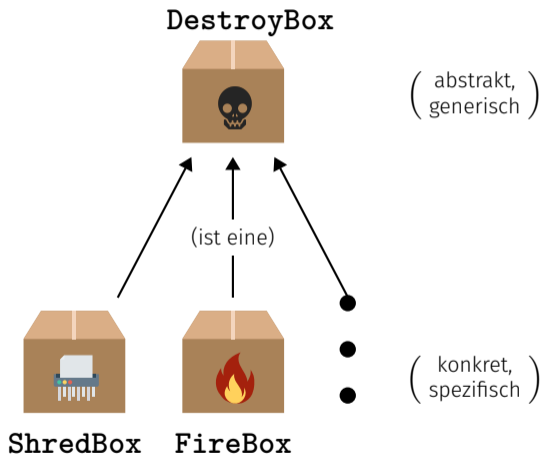
(konkret,
spezifisch)

```
void move_house(DestroyBox& db) {  
    // any destroy box will do  
    db.dispose(old_ikea_couch);  
    db.dispose(cheap_wine);  
    ...  
}
```

```
FireBox fb(5000°C);  
move_house(fb);
```

```
ShredBox sb;  
move_house(sb);
```

Abstrakte und konkrete Zeichenströme



Caesar-Code: Generalisierung

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

    in >> std::noskipws;

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

- `std::istream/std::ostream` ist ein *abstrakter Eingabe-/Ausgabestrom* an `chars`
- Aufruf der Funktion erfolgt mit *konkreten* Strömen, z.B.:
 - Konsole: `std::cin/cout`
 - Dateien: `std::ifstream/ofstream`

Caesar-Code: Generalisierung, Beispiel 1

```
#include <iostream>
```

```
...
```

```
// in void main():
```

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

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

Caesar-Code: Generalisierung, Beispiel 2

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

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

caesar(std::cin, to, s);
```

Aufruf der generischen **caesar**-Funktion: Von **std::cin** zu Datei

Caesar-Code: Generalisierung, Beispiel 3

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

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

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

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

Caesar-Code: Generalisierung, Beispiel 4

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

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

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

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

Ströme: Abschluss

Hinweis: Sie müssen Ströme nur *anwenden* können

- *Anwenderwissen*, auf dem Niveau der vorherigen Folien, reicht für Hausaufgaben und Prüfung aus
- D.h. Sie müssen nicht wissen, wie Ströme intern funktionieren
- Wie sie selbst *abstrakte* und dazu passende *konkrete Typen* erstellen können, erfahren Sie ganz am Ende dieses Kurses

Texte

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

Benutzung von `std::string`

- Deklaration und Initialisierung mittels Literal:

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

- Mit variabler Länge initialisieren:

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

`text` wird mit n 'a's gefüllt

- Texte vergleichen:

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

`true` wenn zeichenweise gleich

Benutzung von `std::string`

- Grösse auslesen:

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

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

- Einzelne Zeichen lesen:

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

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

- Einzelne Zeichen schreiben:

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

Benutzung von `std::string`

- Strings konkatenieren (zusammensetzen):

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

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

15. Vektoren II

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

Mehrdimensionale Vektoren

- Zum Speichern von mehrdimensionalen Strukturen wie Tabellen, Matrizen, ...
- ...können *Vektoren von Vektoren* verwendet werden:

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


Mehrdimensionale Initialisierung

Vektoren:

Mittels Initialisierungslisten:

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

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

Mehrdimensionale Initialisierung

Vektoren:

Auf bestimmte Grösse füllen:

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

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

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

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

Mehrdimensionale Vektoren und Typ-Aliasse

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

```
using Name = Typ;
```

Name, unter dem der Typ neu auch angesprochen werden kann

bestehender Typ

Typ-Alias: Beispiel

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

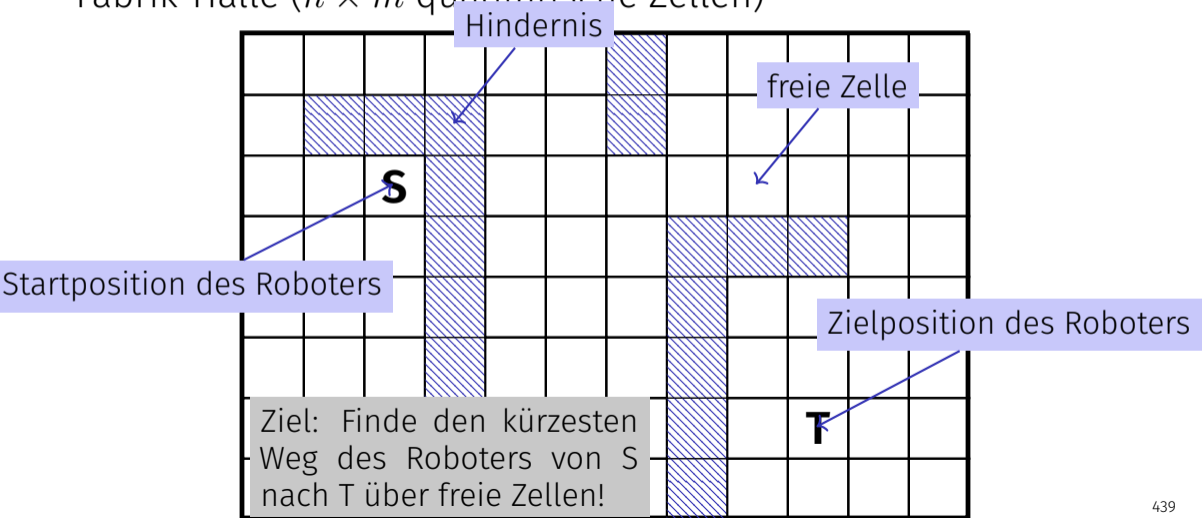
// POST: Matrix 'm' was output to stream 'out'
void print(const imatrix& m, std::ostream& out);

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

Erinnerung: **const**-Referenz für Effizienz (keine Kopie) und Sicherheit (unveränderlich)

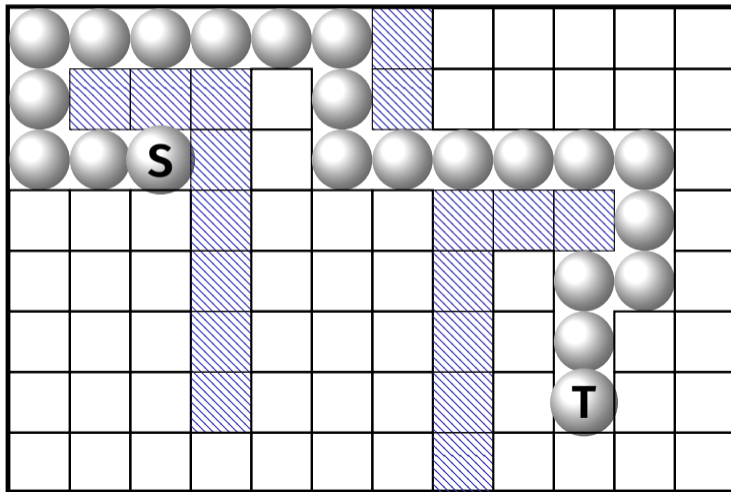
Anwendung: Kürzeste Wege

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



Anwendung: Kürzeste Wege

Lösung



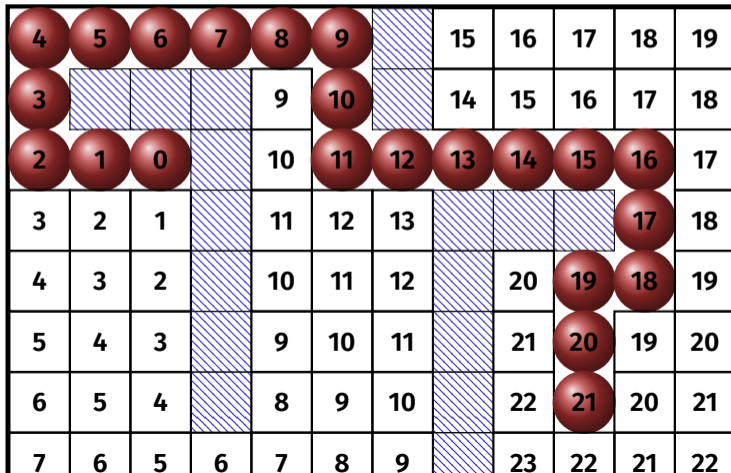
Ein (scheinbar) anderes Problem

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

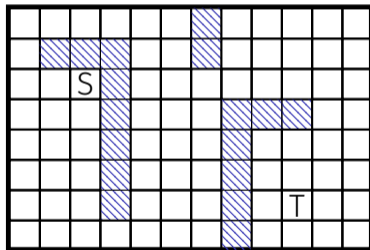
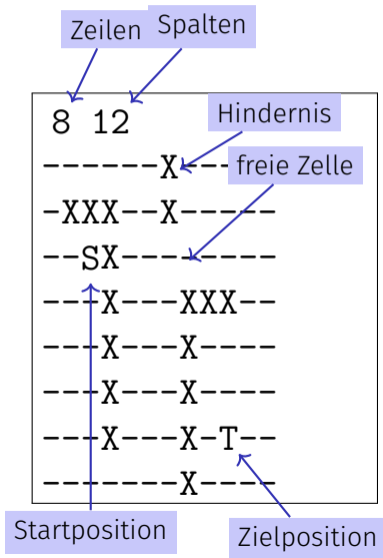


Ein (scheinbar) anderes Problem

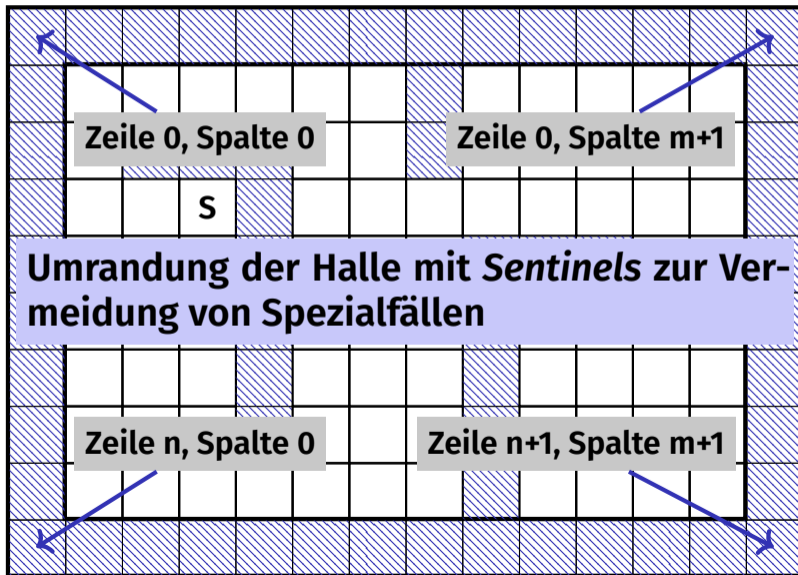
Finde die *Längen* der kürzesten Wege zu *allen* möglichen Zielen



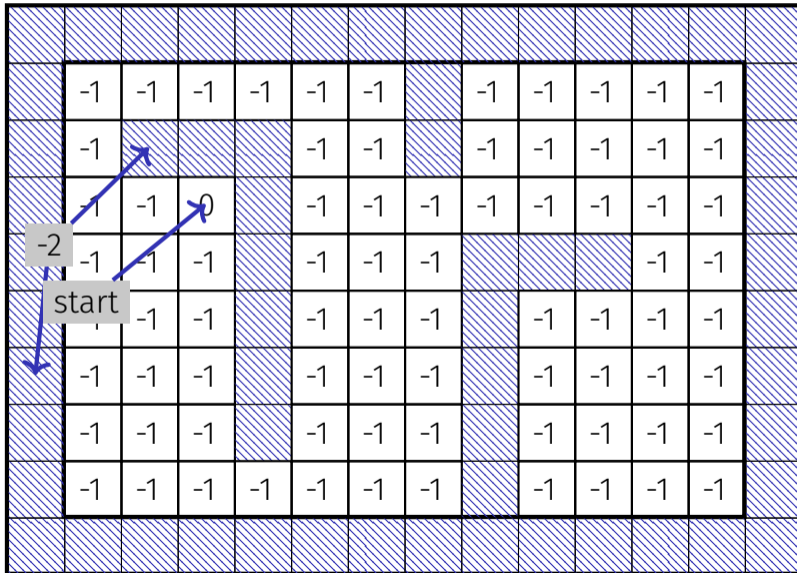
Vorbereitung: Eingabeformat



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

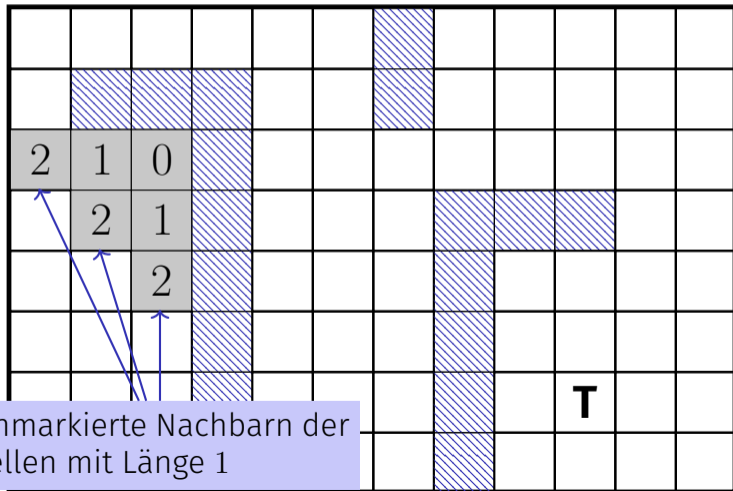
- Hinzufügen der umschliessenden „Wände“

```
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 $i = 1, 2, 3, \dots$

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

Das Kürzeste-Wege-Programm

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

```
2int r = tr; int c = tc;2
3while (floor[r][c] > 0)3 {
4    const int d = floor[r][c] - 1;4
5    floor[r][c] = -3;5
6    if      (floor[r-1][c] == d) --r;
7    else if (floor[r+1][c] == d) ++r;
8    else if (floor[r][c-1] == d) --c;
9    else ++c; // (floor[r][c+1] == d)
6}
```

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";
}
```



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


Das Kürzeste-Wege-Programm

- Algorithmus: *Breitensuche* (Breiten- vs. *Tiefensuche* wird typischerweise in Algorithmmik-Vorlesungen diskutiert)
- Das Programm kann recht langsam sein, weil für jedes i alle Zellen durchlaufen werden
- Verbesserung: Für Markierung i , durchlaufe nur die Nachbarn der Zellen mit Markierung $i - 1$
- Verbesserung: Stoppe, sobald das Ziel erreicht wurde

16. Rekursion 1

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

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 **garantierten Fortschritt**
Richtung einer Abbruchbedingung (\approx Basisfall)

Beispiel `fac(n)`:

- Rekursion endet falls $n \leq 1$
- Rekursiver Aufruf mit neuem Argument $< n$
- Abbruchbedingung wird daher garantiert erreicht

```
unsigned int fac(  
    unsigned int n) {  
  
    if (n <= 1)  
        return 1;  
    else  
        return n * fac(n-1);  
}
```

Rekursive Funktionen: Auswertung

```
int fac(int n) {  
    if (n <= 1)  
        return 1;  
    else  
        return n * fac(n-1);  
}  
  
...  
std::cout << fac(4);
```

$\text{fac}(4) \rightsquigarrow \text{int } n = 4$

$\hookrightarrow \text{fac}(n - 1) \rightsquigarrow \text{int } n = 3$

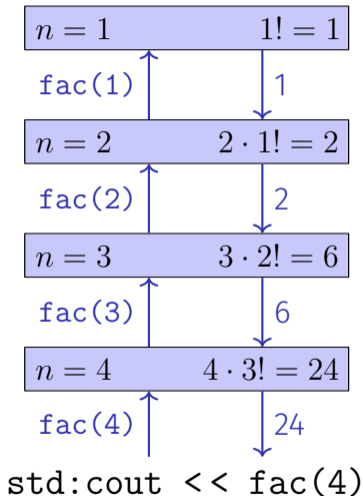
\vdots

*Jeder Aufruf von **fac** arbeitet mit seinem eigenen **n***

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 $\text{gcd}(a, b)$ zweier natürlicher Zahlen a und b
- basiert auf folgender mathematischen Rekursion (Beweis im Skript):

$$\text{gcd}(a, b) = \begin{cases} a, & \text{falls } b = 0 \\ \text{gcd}(b, a \bmod b), & \text{andernfalls} \end{cases}$$

Euklidischer Algorithmus in C++

$$\text{gcd}(a, b) = \begin{cases} a, & \text{falls } b = 0 \\ \text{gcd}(b, a \bmod b), & \text{andernfalls} \end{cases}$$

```
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 . . .

Fibonacci-Zahlen in C++

Laufzeit

`fib(50)` dauert „ewig“, denn es berechnet
 F_{48} 2-mal, F_{47} 3-mal, F_{46} 5-mal, F_{45} 8-mal, F_{44} 13-mal,
 F_{43} 21-mal ... F_1 ca. 10^9 mal (!)

```
unsigned int fib(unsigned int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
    return fib(n-1) + fib(n-2); // n > 1  
}
```

Schnelle Fibonacci-Zahlen

Idee:

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

Kann rekursiv und iterativ implementiert werden, letzteres ist einfacher/direkter

Schnelle Fibonacci-Zahlen in C++

```
unsigned int fib(unsigned int n) {  
    if (n == 0) return 0;  
    if (n == 1) return 1;  
  
    unsigned int a = 0; // F_0  
    unsigned int b = 1; // F_1  
  
    for (unsigned int i = 2; i <= n; ++i) {  
        unsigned int a_old = a; //  $F_{i-2}$   
        a = b; // a becomes  $F_{i-1}$   
        b += a_old; // b becomes  $F_{i-1} + F_{i-2}$ , i.e.  $F_i$   
    }  
    return b;  
}
```

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

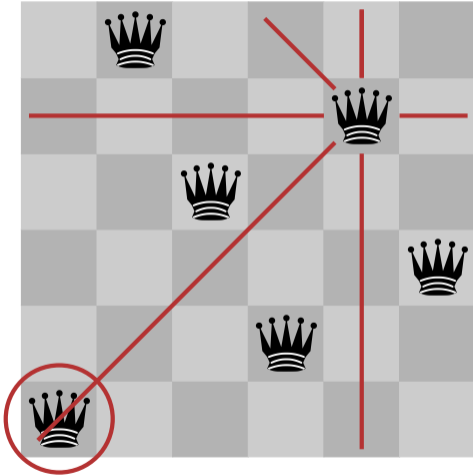
- Iteration (Schleifen)
- expliziten „Aufrufstapel“ (z.B. mittels eines Vektors).

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

Die Macht der Rekursion

- Einige Probleme scheinen ohne Rekursion kaum lösbar zu sein. Mit Rekursion werden sie plötzlich deutlich einfacher lösbar.
- Beispiele: *das n -Damen-Problem*, Die Türme von Hanoi, Parsen von Ausdrücken, Sudoku-Löser, Umgekehrte Aus- oder Eingabe, Suchen in Bäumen, Divide-And-Conquer (z.B. Sortieren) , ...
- ...sowie die 2. Bonusaufgabe: Nonogramme

Das n -Damen Problem

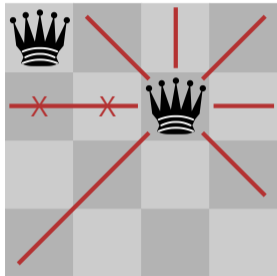


- Gegeben sei ein $n \times n$ Schachbrett
- Zum Beispiel $n = 6$
- Frage: ist es möglich n Damen so zu platzieren, dass keine zwei Damen sich bedrohen?
- Falls ja, wie viele Lösungen gibt es?

Lösung?

- Durchprobieren aller Möglichkeiten?
- $\binom{n^2}{n}$ Möglichkeiten. Zu viele!
- Nur eine Dame pro Zeile: n^n Möglichkeiten. Besser – aber auch noch zu viele.
- Idee: Unsinnige Versuche gar nicht erst weiterverfolgen, stattdessen falsche Züge zurücknehmen \Rightarrow *Backtracking*

Lösung mit Backtracking

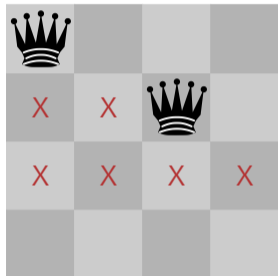


Nächste Dame in
nächster Zeile
(keine Kollision)

queens

0
2
0
0

Lösung mit Backtracking

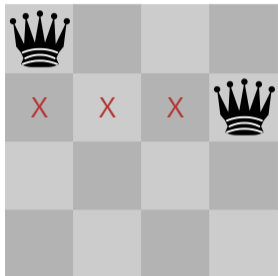


Alle Felder in nächster Zeile verboten. Zurück! (Backtracking!)

queens

0
2
4
0

Lösung mit Backtracking

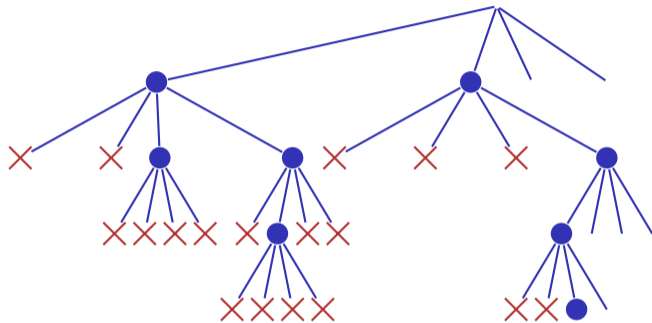
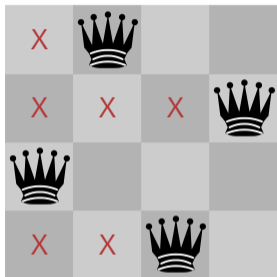


Dame eins weiter
setzen und wieder
versuchen

queens

0
3
0
0

Suchstrategie als Baum visualisiert



Prüfe Dame

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

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

Rekursion: Finde eine Lösung

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


Rekursion: Zähle alle Lösungen

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

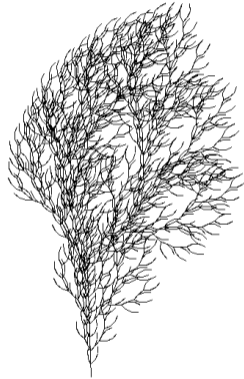
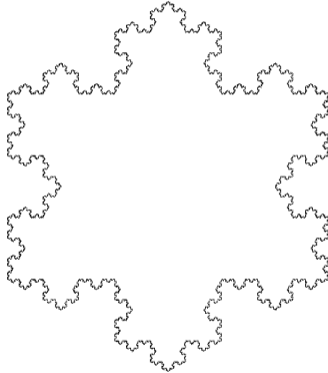
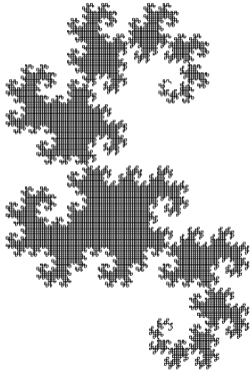
Hauptprogramm

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

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

Lindenmayer-Systeme (L-Systeme)

Fraktale aus Strings und Schildkröten



- L-Systeme wurden vom ungarischen Biologen Aristid Lindenmayer (1925–1989) zur Modellierung von Pflanzenwachstum erfunden.
- Rekursion ist natürlich klausurrelevant die L-Systeme an sich sind es jedoch nicht

Definition und Beispiel

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

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

c	$P(c)$
F	F + F +
+	+
-	-

- F

Definition

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

Die beschriebene Sprache

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

$$P(F) = F + F +$$

$$w_0 := s_0$$

$$w_0 := F$$

$$w_1 := P(w_0)$$

$$w_1 := \begin{array}{c} F + F + \\ \boxed{F} \boxed{+} \boxed{F} \boxed{+} \end{array}$$

$$w_2 := P(w_1)$$

$$w_2 := \begin{array}{c} \boxed{F + F +} \boxed{+} \boxed{F + F +} \boxed{+} \\ P(F)P(+)P(F)P(+) \end{array}$$

⋮

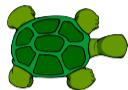
⋮

Definition

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

Turtle-Grafik

Schildkröte mit Position und Richtung

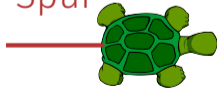


Schildkröte versteht 3 Befehle:

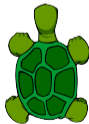
F: Gehe einen Schritt vorwärts



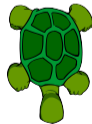
Spur



+: Drehe dich um 90 Grad ✓

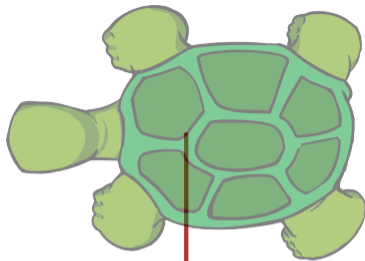
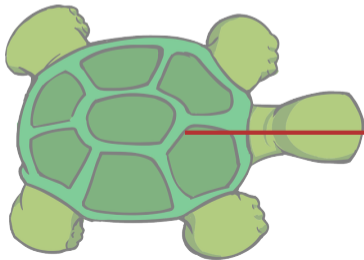


-: Drehe dich um -90 Grad ✓



Wörter zeichnen!

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



lindenmayer:

Hauptprogramm

Wort $w_0 \in \Sigma^*$:

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

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

    return 0;
}
```

$w = w_0 = F$

lindenmayer:

production

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

lindenmayer:

replace

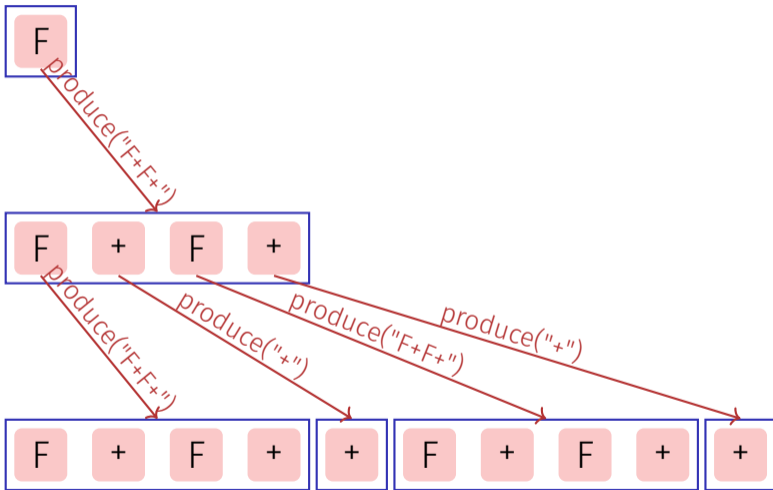
```
// POST: returns the production of c
std::string replace(const char c) {
    switch (c) {
        case 'F':
            return "F+F+";
        default:
            return std::string (1, c); // trivial production c -> c
    }
}
```

lindenmayer:

draw

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

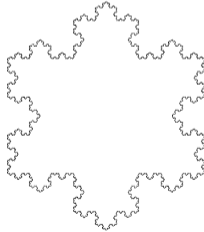
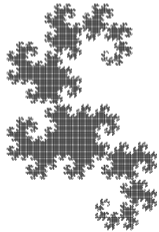
Die Rekursion



(Obige Implementierung realisiert eine *Tiefensuche*)

L-Systeme: Erweiterungen

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



17. Rekursion 2

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

Motivation: Taschenrechner

Ziel: Bau eines Kommandozeilenrechners

```
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
- Fließkommarithmetik
- 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

Eingabe:

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

Muss gespeichert bleiben,
damit jetzt ausgewertet
werden kann!

Analyse des Problems

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

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

Wir werden die Symbole elegant mittels Rekursion zwischenspeichern.

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

Formale Grammatiken

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

Eine formale Grammatik definiert, welche Sätze gültig sind.

Zur Beschreibung der Grammatik verwenden wir:

Extended Backus Naur Form (EBNF)

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

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

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

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

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

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

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

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

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

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

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

Zahl

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

■ eine Ziffer , oder 2

■ eine Ziffer gefolgt von einer Folge von Ziffern 2 0 1 9

`unsigned_integer = digits .`

`digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .`

`digits = digit | digit digits .`

Alternative

Terminales Symbol

Nicht terminales Symbol

Zahl (nicht-rekursiv)

Eine Zahl ist eine Folge von Ziffern. Eine Folge von Ziffern ist

■ eine Ziffer, oder

2

■ eine Ziffer gefolgt von **beliebig vielen Ziffern**

2 0 1 9

`unsigned_integer = digits .`

`digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .`

`digits = digit { digit } .`

Optionale Repetition

Zahl, erweitert

Eine Fließkommazahl ist

- eine Folge von Ziffern, oder
- eine Folge von Ziffern gefolgt von `.` gefolgt von Ziffern

Float = `Digits | Digits "." Digits.`

Ausdrücke

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

Was benötigen wir in einer Grammatik?

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

Faktor

Term

Ausdruck

Die EBNF für Ausdrücke

Ein Faktor ist

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

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

Nicht-terminales Symbol

Terminales Symbol

Alternative

Die EBNF für Ausdrücke

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

Implikation: Ein Faktor beginnt mit

- einer Ziffer, oder
- mit „(“ , oder
- mit „-“.

Die EBNF für Ausdrücke

Ein Term ist

- Faktor,
- Faktor * Faktor, Faktor / Faktor,
- Faktor * Faktor * Faktor, Faktor / Faktor * Faktor, ...
- ...

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

Optionale Repetition

Die EBNF für Ausdrücke

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

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

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

Parsen

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

Regeln

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

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

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

Funktionen

(Parser)

Ausdruck wird aus einem [Eingabestrom](#) gelesen.

```
// POST: returns true if and only if in_stream = factor ...  
//       and in this case extracts factor from in_stream  
bool factor (std::istream& in_stream);
```

```
// POST: returns true if and only if in_stream = term ...,  
//       and in this case extracts all factors from in_stream  
bool term (std::istream& in_stream);
```

```
// POST: returns true if and only if in_stream = expression ...,  
//       and in this case extracts all terms from in_stream  
bool expression (std::istream& in_stream);
```

Funktionen (Parser mit Auswertung)

Ausdruck wird aus einem **Eingabestrom** gelesen.

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

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

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


Vorausschau von einem Zeichen...

...um jeweils die richtige Alternative zu finden.

```
// POST: the next character at the stream is returned  
//       without being consumed. returns 0 if stream ends.
```

```
char peek (std::istream& input){  
    if (input.eof()) return 0; // end of stream  
    return input.peek(); // next character in input  
}
```

```
// POST: leading whitespace characters are extracted from input  
//       and the first non-whitespace character on input returned
```

```
char lookahead (std::istream& input) {  
    input >> std::ws; // skip whitespaces  
    return peek(input);  
}
```

Zahlen parsen

```
bool isDigit(char ch){
    return ch >= '0' && ch <= '9';
}
// POST: returns an unsigned integer consumed from the stream
// number = digit {digit}.
unsigned int unsigned_number (std::istream& input){
    char ch = lookahead(input);
    assert(isDigit(ch));
    unsigned int num = 0;
    while(isDigit(ch) && input >> ch){ // read remaining digits
        num = num * 10 + ch - '0';
        ch = peek(input);
    }
    return num;
}
```

unsigned_number = digit { digit }.

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Rosinenpickerei

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

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

Faktoren auswerten

```
double factor (std::istream& in_stream)
{
    double value;
    if (consume(in_stream, '(')) {
        value = expression (in_stream);
        consume(in_stream, ')');
    } else if (consume(in_stream, '-')) {
        value = -factor (in_stream);
    } else {
        value = unsigned_number(in_stream);
    }
    return value;
}
```

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

Terme auswerten

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

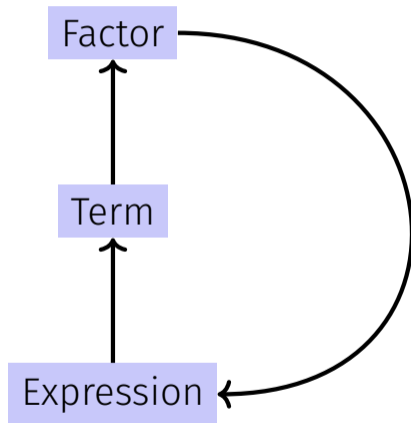
term = factor { "*" factor | "/" factor }.

Ausdrücke auswerten

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

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

Rekursion!



EBNF – Und es funktioniert!

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

```
factor      = unsigned_number  
             | "(" expression ")"  
             | "-" factor.  
  
term        = factor { "*" factor | "/" factor }.  
  
expression = term { "+" term | "-" term }.
```

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


18. Structs

Rationale Zahlen, Struct-Definition

Rechnen mit rationalen Zahlen

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

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

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

Invariante: spezifiziert gültige Wertkombinationen (informell).

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
};
```

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

```
// 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;
}
```

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

Name des neuen Typs (Bezeichner)

Namen der zugrundeliegenden Typen

```
struct T {  
  T1 name1;  
  T2 name2;  
  :  
  :  
  Tn namen;  
};
```

Namen der Member-Variablen

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

Struct-Definitionen: Beispiele

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

Zugrundeliegende Typen können fundamentale aber auch *benutzerdefinierte* Typen sein.

Struct-Definitionen: Beispiele

```
struct extended_int {  
    // represents value if is_positive==true  
    // and -value otherwise  
    unsigned int value;  
    bool is_positive;  
};
```

Die zugrundeliegenden Typen können natürlich auch *verschieden* sein.

Structs: Member-Zugriff

Ausdruck vom Struct-Typ T .

Name einer Member-Variablen des Typs T .

$expr.name_k$

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

Member-Zugriff-Operator $.$

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

Initialisierung:

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

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

Structs: Initialisierung und Zuweisung

Zuweisung:

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

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

Structs: Initialisierung und Zuweisung

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

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

`rational u = t;` ← Memberweise Kopie

`t = u;` ← Memberweise Kopie

`rational v = add (u,t);` ← Memberweise Kopie

Structs vergleichen?

Für jeden fundamentalen Typ (`int`, `double`, ...) gibt es die Vergleichsoperatoren `==` und `!=`, aber nicht für Structs!
Warum?

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

Structs als Funktionsargumente

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

Call by Value !

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

Structs als Funktionsargumente

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

Call by Reference

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

Benutzerdefinierte Operatoren

Statt

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

würden wir lieber

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

schreiben.

Das geht mit *Überladen von Operatoren* (\rightarrow nächste Woche).

19. Klassen

Funktions- und Operatorüberladung, Datenkapselung, Klassen, Memberfunktionen, Konstruktoren

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*:

operator*op*

- 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 <<

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 ($\frac{3}{5} \rightarrow 0.6$)

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

Der Kunde will mehr

“Können wir rationale Zahlen mit erweitertem Wertebereich bekommen?”

■ Klar, kein Problem, z.B.:

```
struct rational {  
    int n;  
    int d;  
};
```



```
struct rational {  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

Neue Version von RAT PACK®



Nichts geht mehr!

- 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;
};
```

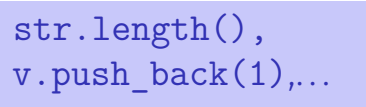
Wir sind schuld!

- 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.

⇒ RAT PACK[®] ist Geschichte...

Idee der Datenkapselung (Information Hiding)

- Ein Typ ist durch *Wertebereich* und *Funktionalität* eindeutig definiert.
- Die *Repräsentation* soll *nicht sichtbar* sein.
- \Rightarrow Dem Kunden wird keine *Repräsentation*, sondern **Funktionalität** angeboten.



```
str.length(),  
v.push_back(1),...
```

Klassen

- sind das Konzept zur **Datenkapselung** in C++
- sind eine Variante von Structs
- gibt es in vielen *objektorientierten Programmiersprachen*

Datenkapselung: `public` / `private`

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

Wird statt `struct` verwendet, wenn überhaupt etwas "versteckt" werden soll.

Einziger Unterschied:

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

Datenkapselung: `public` / `private`

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

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

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

Anwendungscode:

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

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

Memberfunktionen: Deklaration

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

öffentlicher Bereich

Memberfunktion

Memberfunktionen haben Zugriff auf private Daten

Gültigkeitsbereich von Mem-
bern in einer Klasse ist die
ganze Klasse, unabhängig von
der Deklarationsreihenfolge

Memberfunktionen: Aufruf

```
// Definition des Typs
class rational {
    ...
};
...
// Variable des Typs
rational r; Member-Zugriff

int n = r.numerator(); // Zaehler
int d = r.denominator(); // Nenner
```

Memberfunktionen: Definition

```
// POST: returns numerator of this instance
int numerator () const
{
    return n;
}
```



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

const und Memberfunktionen

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

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

Das **const** an einer Memberfunktion liefert das Versprechen, dass eine Instanz nicht über diese Funktion verändert wird. **const** Objekte dürfen nur **const** Memberfunktionen aufrufen!

This rational vs. dieser Bruch

So wäre es **in etwa** ...

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

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

... ohne Memberfunktionen

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

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

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

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

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

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

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

- So geht's auch.

Konstruktoren

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

Initialisierung? Konstruktoren!

```
class rational
{
public:
    rational (int num, int den)
        : n (num), d (den) ← Initialisierung der
                               Membervariablen
    {
        assert (den != 0); ← Funktionsrumpf.
    }
    ...
};
...
rational r (2,3); // r = 2/3
```

Konstruktoren: Aufruf

- direkt

```
rational r (1,2); \small // 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
```

Der Default-Konstruktor

```
class rational
{
public:
    ...
    rational () ← Leere Argumentliste
        : n (0), d (1)
    {}
    ...
};
...
rational r; // r = 0
```

⇒ Es gibt keine uninitialisierten Variablen vom Typ rational mehr!

Alternative: Default-Konstruktor löschen

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

⇒ Es gibt keine uninitialisierten Variablen vom Typ rational mehr!

Benutzerdefinierte Konversionen

sind definiert durch Konstruktoren mit genau *einem* Argument

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

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

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

RAT PACK[®] Reloaded ...

Kundenprogramm sieht nun so aus:

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

- Wir können die Memberfunktionen zusammen mit der Repräsentation anpassen. ✓

RAT PACK[®] Reloaded ...

vorher

```
class rational {  
    ...  
private:  
    int n;  
    int d;  
};
```

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

nachher

```
class rational {  
    ...  
private:  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

```
int numerator () const {  
    if (is_positive)  
        return n;  
    else {  
        int result = n;  
        return -result;  
    }  
}
```

RAT PACK[®] Reloaded ?

```
class rational {  
    ...  
private:  
    unsigned int n;  
    unsigned int d;  
    bool is_positive;  
};
```

```
int numerator () const  
{  
    if (is_positive)  
        return n;  
    else {  
        int result = n;  
        return -result;  
    }  
}
```

- Wertebereich von Zähler und Nenner wieder wie vorher
- Dazu noch möglicher Überlauf

Datenkapselung noch unvollständig

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

```
class rational {
public:
    // POST: returns numerator of *this
    int numerator () const;
    ...
private:
    // none of my business
};
```

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

Fix: „Unser“ Typ `rational::integer`

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

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

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

RAT PACK[®] Revolutions

Endlich ein Kundenprogramm, das stabil bleibt:

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

Deklaration und Definition getrennt

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

rational.h

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

rational.cpp

Klassenname

::

Membername

20. Dynamische Datenstrukturen I

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

Wiederholung: `vector<T>`

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

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

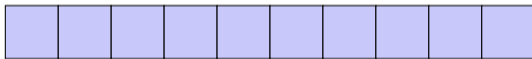
- Ein Vektor ist eine *dynamische Datenstruktur*, deren Grösse sich zur Laufzeit ändern kann

Unser eigener Vektor!

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

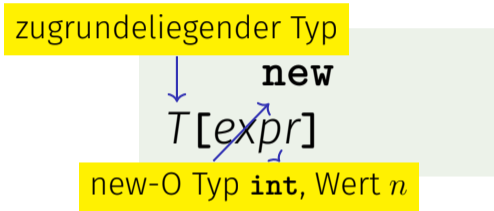
Vektoren im Speicher

Bereits bekannt: Ein Vektor belegt einen *zusammenhängenden* Speicherbereich



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

Der `new`-Ausdruck für Arrays



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



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

Der `new`-Ausdruck für Arrays

zugrundeliegender Typ

```
p = new  
T[expr]
```

new-O Typ `int`, Wert n

- **Wert:** Die Startadresse des Speicherbereichs



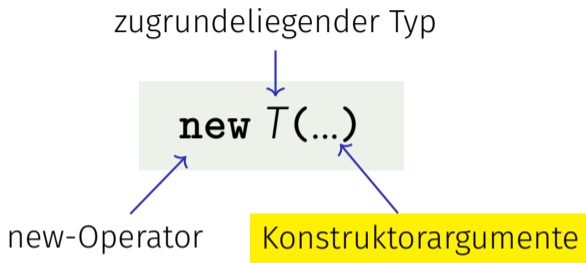
- **Typ:** Ein Zeiger T^* (mehr dazu gleich)

Ausblick: `new` und `delete`

```
new  
T[expr]
```

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

Der `new`-Ausdruck (ohne Arrays)



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

Zeiger-Typen

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

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

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


Zeiger-Typen

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

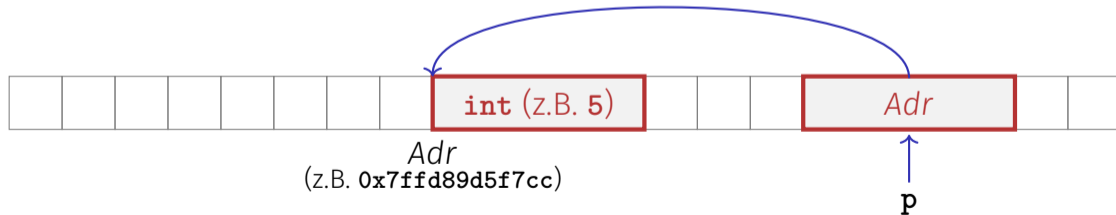
Wo ein T^* draufsteht, muss auch ein T drin sein

```
int* p = ...;  
std::string* q = p; // Kompilerfehler!
```

Zeiger-Typen

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

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



Adress-Operator

Frage: Wie kommt man an die Adresse eines Objekts?

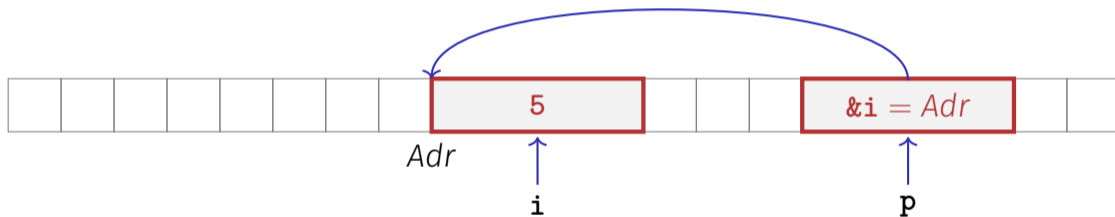
1. Direkt beim Erzeugen eines neuen Objekts mittels **new**
2. Für bestehende Objekte: Mittels des *Adress-Operators* **&**

&expr ← expr: L-Wert vom Typ *T*

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

Adress-Operator

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



Nächste Frage: Wie „folgt“ man einem Zeiger?

Dereferenz-Operator

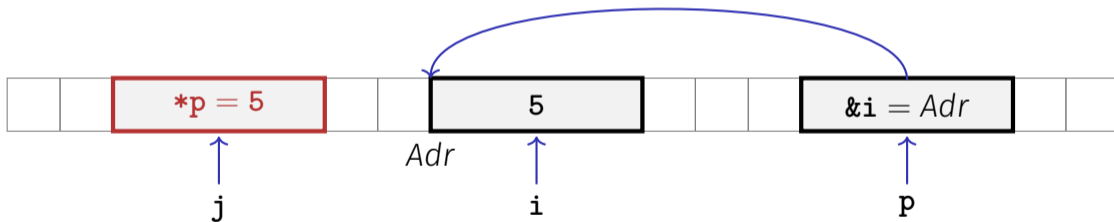
Antwort: Mittels des *Dereferenz-Operators* *

$*expr$ ← expr: R-Wert vom Typ T^*

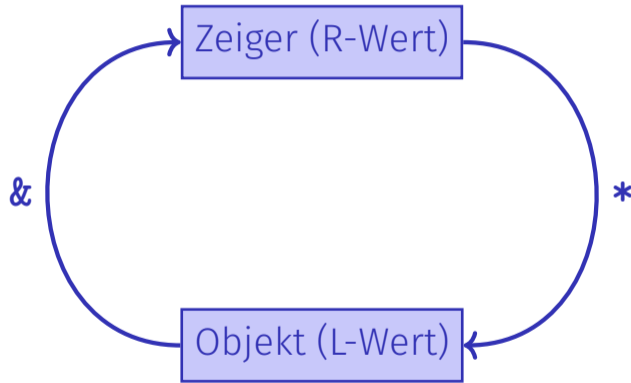
- **Wert** des Ausdrucks: Der Wert des Objekts an der durch *expr* bestimmten Adresse
- **Typ** des Ausdrucks: T

Dereferenz-Operator

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



Adress- und Dereferenzoperator




Eselsbrücke

Die Deklaration

```
T* p; // p ist vom Typ „Zeiger auf T“
```

kann gelesen werden als

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



Obwohl das legal ist,
schreiben wir es nicht
so!

Null-Zeiger

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

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

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

```
int* p; // Zugriff auf p ist undefiniertes Verhalten  
int* q = nullptr; // q hingegen zeigt explizit ins Nichts
```

Zeiger-Arithmetik: Zeiger plus int

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

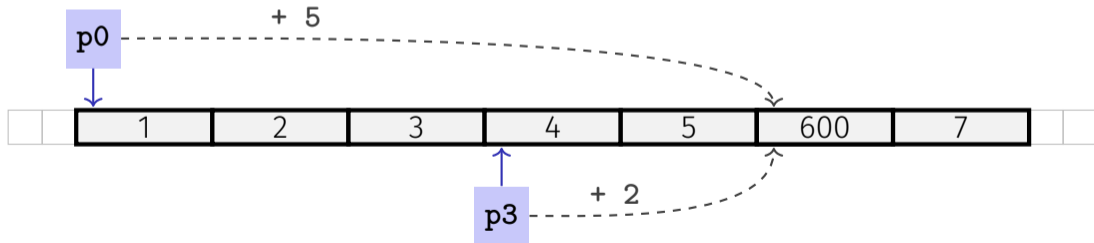


Frage: Wie zeigt man auf hintere Elemente? → mittels *Zeiger-Arithmetik*:

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

Zeiger-Arithmetik: Zeiger plus int

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

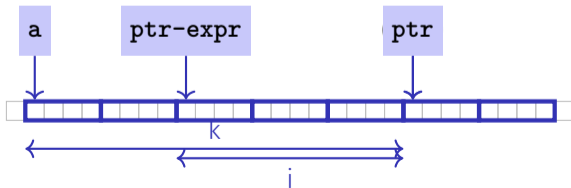


Zeiger-Arithmetik: Zeiger minus `int`

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

$ptr - expr$

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



Zeigersubtraktion

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

$p1 - p2$ hat den Wert $k_1 - k_2$



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

- Die Zeigerdifferenz beschreibt, wie weit die Elemente voneinander entfernt im Speicher liegen

Zeigeroperatoren

Beschreibung	Op	Stellig- keit	Präzedenz	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

Zeigerwerte sind keine Ganzzahlen

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

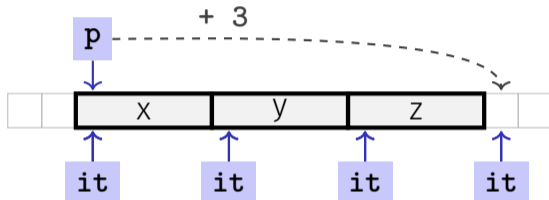
`ptr + 1` ist **nicht** die nächste Hausnummer, sondern die s -nächste, wobei s der Speicherbedarf eines Objekts des Typs ist, der `ptr` zugrundeliegt.

- Zeiger und Ganzzahlen sind nicht kompatibel:

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

Sequenzielle Zeiger-Iteration

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

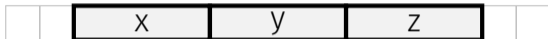


```
for (char* it = p; it zeigt aufs erste Element  
    it != p + 3; Abbruch falls Ende erreicht  
    ++it) { Zeiger elementweise voranschieben
```

```
    std::cout << *it << ' '; /// Aktuelles Element ausgeben: 'x'  
}
```


Wahlfreier Zugriff auf Arrays

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



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

Wahlfreier Zugriff auf Arrays

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

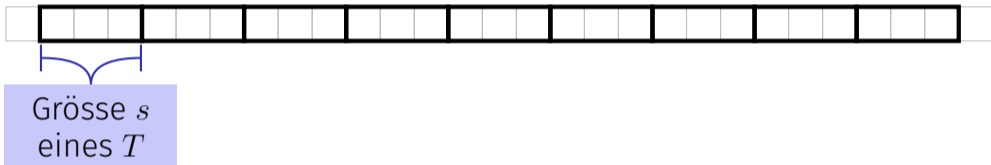
```
char* p = new char[3]{'x', 'y', 'z'};

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

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

Wahlfreier Zugriff auf Arrays

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



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

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

Wahlfreier Zugriff

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

Sequenzieller Zugriff

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

Statische Arrays

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

Arrays in Funktionen

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



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

Arrays in (mutierenden) Funktionen: `fill`

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

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

Funktionen mit/ohne Effekt

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

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


Positionierung von Const

Zur Determinierung der Zugehörigkeit des **const** Modifiers:
const *T* ist äquivalent zu *T* **const** (und kann auch so geschrieben werden):

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

Beide Schreibweisen werden in der Praxis genutzt

Const und Zeiger

Lies Deklaration von rechts nach links

```
int const p1;
```

p1 ist eine konstante Ganzzahl

```
int const* p2;
```

p2 ist ein Zeiger auf eine konstante Ganzzahl

```
int* const p3;
```

p3 ist ein konstanter Zeiger auf eine Ganzzahl

```
int const* const p4;
```

p4 ist ein konstanter Zeiger auf eine konstante Ganzzahl

Nicht-mutierende Funktionen: `print`

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

```
// PRE: [begin, end) ist ein gültiger Bereich  
// POST: Die Werte in [begin, end) wurden ausgegeben
```

```
void print(  
    int const* const begin,  
    const int* const end) {
```

```
    for (int const* p = begin; p != end; ++p)
```

```
        std::cout << *p << " ";  
}
```

Const-Zeiger auf const-int

Ebenfalls (aber andere Schreibweise)

Zeiger, *nicht* const, auf const-int

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

const ist nicht absolut

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

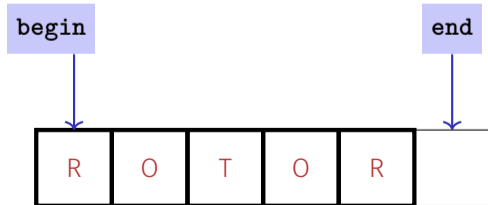
Beispiel

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

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

Wow – Palindrome!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
    while (begin < end)
        if (*(begin++) != *(--end)) return false;
    return true;
}
```



Arrays, `new`, Zeiger: Abschluss

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

Array-basierter Vektor

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

Unser eigener Vektor!

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

Array-basierter Klassensignatur

Vektor

avec:

```
class avec {  
    // Private (internal) state:  
    int* elements; // Pointer to first element  
    unsigned int count; // Number of elements  
  
public: // Public interface:  
    avec(unsigned int size); // Constructor  
    unsigned int size() const; // Size of vector  
    int& operator[](int i); // Access an element  
    void print(std::ostream& sink) const; // Output elems.  
}
```


Konstruktor `avec::avec()`

```
avec::avec(unsigned int size)
    : count(size) ← { Grösse speichern

    elements = new int[size]; ← Speicher allozieren
}
```

Nebenbemerkung: Vektor wird nicht mit einem Standardwert initialisiert

Exkurs: Zugriff auf Membervariablen

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

- **elements** ist eine Membervariable unserer **avec**-Instanz
- Diese Instanz ist mittels des *Zeigers* **this** zugreifbar
- **elements** ist eine Kurzschreibweise von **(*this).elements**
- Dereferenzieren eines Zeigers (***this**) gefolgt von einem Memberzugriff (**.elements**) ist eine so häufig genutzte Operation, dass sie verkürzt geschrieben werden kann als **this->elements**
- Eselsbrücke: „Folge dem Zeiger zur Membervariablen“

Funktion `avec::size()`

```
int avec::size() const ← {  
    return this->count; ←  
}
```

Verändert den Vektor nicht


Grösse zurückgeben

Anwendungsbeispiel:

```
avec v = avec(7);  
assert(v.size() == 7); // ok
```

Funktion `avec::operator []`

```
int& avec::operator [] (int i) {  
    return this->elements[i];  
}
```

 `i`-tes Element zurückgeben

Elementzugriff mit Indexüberprüfung:

```
int& avec::at(int i) const {  
    assert(0 <= i && i < this->count);  
  
    return this->elements[i];  
}
```

Funktion `avec::operator []`

```
int& avec::operator [](int i) {  
    return this->elements[i];  
}
```

Anwendungsbeispiel:

```
avec v = avec(7);  
std::cout << v[6]; // Outputs a "random" value  
v[6] = 0;  
std::cout << v[6]; // Outputs 0
```

Funktion `avec::operator[]` braucht's doppelt

```
int& avec::operator[](int i) { return elements[i]; }  
const int& avec::operator[](int i) const { return elements[i]; }
```

- Die erste Memberfunktion ist *nicht const* und gibt eine *nicht-const*-Referenz zurück

```
avec v = ...; // A non-const vector  
std::cout << v.get[0]; // Reading elements is allowed  
v.get[0] = 123; // Modifying elements is allowed
```

- Sie wird auf nicht-const-Vektoren aufgerufen

Funktion `avec::operator[]` braucht's doppelt

```
int& avec::operator[](int i) { return elements[i]; }  
const int& avec::operator[](int i) const { return elements[i]; }
```

- Die zweite Memberfunktion *ist const* und gibt eine *const*-Referenz zurück

```
const avec v = ...; // A const vector  
std::cout << v.get[0]; // Reading elements is allowed  
v.get[0] = 123; // Compiler error: modifications are not allowed
```

- Sie wird auf *const*-Vektoren aufgerufen

Siehe auch das diesem PDF angehängte Beispiel

[getters_and_const.cpp](#)

Funktion `avec::print()`

Elemente mittels sequenziellem Zugriff ausgeben:

```
void avec::print(std::ostream& sink) const {  
    for (int* p = this->elements; ← Zeiger auf erstes Element  
         p != this->elements + this->count; ←  
         ++p) ← Zeiger elementweise vorans Abbruch falls hin-  
    {                                             ter letztem Ele-  
        sink << *p << ' '; ← Aktuelles Element ausgeben  
    }  
}
```


Funktion `avec::print()`

Abschluss: Ausgabeoperator überladen:

```
_____ operator<<(_____ sink,  
                    _____ vec) {  
    vec.print(sink);  
    return _____;  
}
```

```
std::ostream& operator<<(std::ostream& sink,  
                        const avec& vec) {  
    vec.print(sink);  
    return sink;  
}
```

Beobachtungen:

- Konstante Referenz auf `vec`, da unverändert

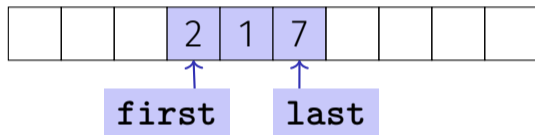
Weitere Funktionen

```
class avec {  
    ...  
    void push_front(int e)      // Prepend e to vector  
    void push_back(int e)     // Append e to vector  
    void remove(unsigned int i) // Cut out ith element  
    ...  
}
```

Gemeinsamkeit: Diese Operationen müssen die *Grösse* des Vektors verändern

Arrays vergrössern/verkleinern

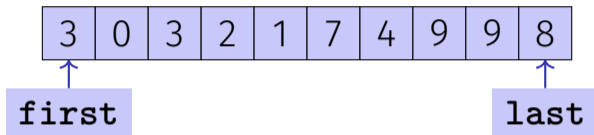
Ein allozierter Speicherblock (z.B. `new int [3]`) kann nicht nachträglich vergrössert/verkleinert werden



Möglichkeit:

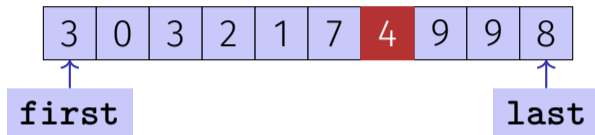
- Mehr Speicher als initial nötig allozieren
- Befüllen aus der Mitte heraus, mittels Zeigern auf erstes und letztes Element

Arrays vergrössern/verkleinern

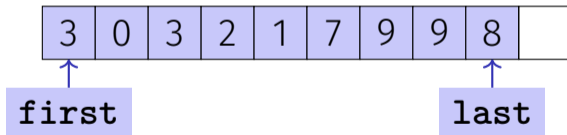


- Aber irgendwann sind alle Plätze belegt
- Dann nötig: Grösseren Speicherblock allozieren und Daten umkopieren

Arrays vergrössern/verkleiner



Elemente löschen erfordert verschieben (via kopieren) aller vorhergehenden oder nachfolgenden Elemente



Ähnlich: Einfügen an beliebiger Position

21. Dynamische Datenstrukturen II

Verkettete Listen, Vektoren als verkettete Listen

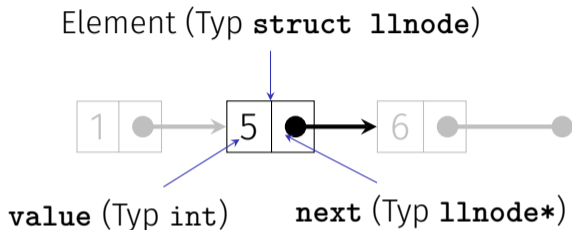
Anderes Speicherlayout: Verkettete Liste

- **Kein** zusammenhängender Speicherbereich und **kein** wahlfreier Zugriff
- Jedes Element zeigt auf seinen Nachfolger
- Einfügen und Löschen **beliebiger** Elemente ist einfach



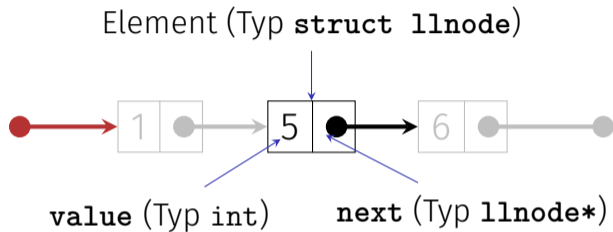
⇒ Unser Vektor kann als verkettete Liste realisiert werden

Verkettete Liste: Zoom



```
struct llnode {  
    int value;  
    llnode* next;  
  
    llnode(int v, llnode* n): value(v), next(n) {} // Constructor  
};
```






Vektor = Zeiger aufs erste Element



```
class llvec {  
    llnode* head;  
public: // Public interface identical to avec's  
    llvec(unsigned int size);  
    unsigned int size() const;  
    ...  
};
```

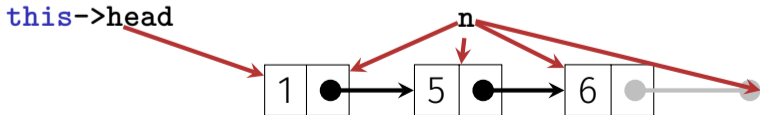
Funktion `llvec::print()`

```
struct llnode {  
    int value;  
    llnode* next;  
    ...  
};
```

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  Zeiger auf erstes Element  
        n != nullptr;  Abbruch falls Ende erreicht  
        n = n->next)  Zeiger elementweise voranschieben  
    {  
        sink << n->value << ' ' << ' ';  Aktuelles Element ausgeben  
    }  
}
```

Funktion `llvec::print()`

```
void llvec::print(std::ostream& sink) const {  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
    {  
        sink << n->value << ' '; // 1 5 6  
    }  
}
```



Funktion `llvec::operator []`

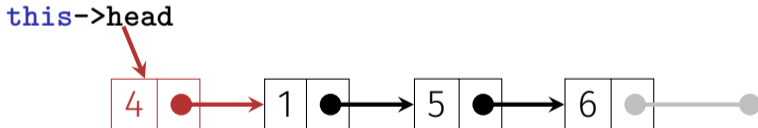
Zugriff auf i -tes Element ähnlich implementiert wie `print()`:

```
int& llvec::operator [] (unsigned int i) {  
    llnode* n = this->head; ← Zeiger auf erstes Element  
  
    for (; 0 < i; --i) | ← Bis zum  $i$ -ten voranschreiten  
        n = n->next;  
  
    return n->value; ←  $i$ -tes Element zurückgeben  
}
```

Funktion `llvec::push_front()`

Vorteil `llvec`: Elemente am Anfang anfügen ist sehr einfach:

```
void llvec::push_front(int e) {  
    this->head =  
        new llnode{e, this->head};  
}
```



Achtung: Wäre der neue `llnode` nicht *dynamisch* alloziert, dann würde er am Ende von `push_front` sofort wieder gelöscht (= Speicher dealloziert)

Funktion `llvec::llvec()`

Konstruktor kann mittels `push_front()` implementiert werden:


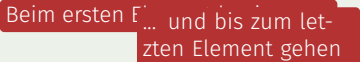

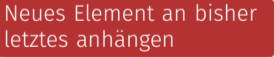
```
llvec::llvec(unsigned int size) {  
    this->head = nullptr; ← head zeigt zunächst ins Nichts  
  
    for (; 0 < size; --size) | ← size mal 0 vorne anfügen  
        this->push_front(0);  
}
```

Anwendungsbeispiel:

```
llvec v = llvec(3);  
std::cout << v; // 0 0 0
```

Funktion `llvec::push_back()`

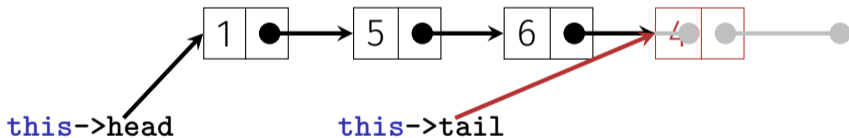
Einfach, aber ineffizient: Verkettete Liste bis ans Ende traversieren und neues Element anhängen

```
void llvec::push_back(int e) {  
    llnode* n = this->head;    
  
    for (; n->next != nullptr; n = n->next);   
  
    n->next =    
        new llnode{e, nullptr};   
}
```

Funktion `llvec::push_back()`

- Effizienter, aber auch etwas komplexer:




1. Zweiter Zeiger, der auf das letzte Element zeigt: `this->tail`
2. Mittels dieses Zeigers kann direkt am Ende angehängt werden



- **Aber:** Verschiedene Grenzfälle, z.B. Vektor noch leer, müssen beachtet werden

Funktion `llvec::size()`

Einfach, aber ineffizient: Grösse durch abzählen *berechnen*

```
unsigned int llvec::size() const {  
    unsigned int c = 0;   
  
    for (llnode* n = this->head;  
         n != nullptr;  
         n = n->next)  
        ++c;   
  
    return c;   
}
```

Funktion `llvec::size()`

Effizienter, aber etwas komplexer: Grösse als Membervariable *nachhalten*

1. Membervariable `unsigned int count` zur Klasse `llvec` hinzufügen
2. `this->count` muss nun bei *jeder* Operation, die die Grösse des Vektors verändert (z.B. `push_front`), aktualisiert werden

Effizienz: Arrays vs. Verkettete Listen

- Speicher: Unser **avec** belegt ungefähr n ints (Vektorgröße n), unser **llvec** ungefähr $3n$ ints (ein Zeiger belegt i.d.R. 8 Byte)
- Laufzeit (mit **avec** = `std::vector`, **llvec** = `std::list`):

```
prepending (insert at front) [100,000x]:
  ▶ avec: 675 ms
  ▶ llvec: 10 ms
appending (insert at back) [100,000x]:
  ▶ avec: 2 ms
  ▶ llvec: 9 ms
removing first [100,000x]:
  ▶ avec: 675 ms
  ▶ llvec: 4 ms
removing last [100,000x]:
  ▶ avec: 0 ms
  ▶ llvec: 4 ms

removing randomly [10,000x]:
  ▶ avec: 3 ms
  ▶ llvec: 113 ms
inserting randomly [10,000x]:
  ▶ avec: 16 ms
  ▶ llvec: 117 ms
fully iterate sequentially (5000 elements) [5,000x]:
  ▶ avec: 354 ms
  ▶ llvec: 525 ms
```

22. Container, Iteratoren und Algorithmen

Container, Mengen, Iteratoren, const-Iteratoren, Algorithmen, Templates

Vektoren sind Container

- Abstrakt gesehen ist ein Vektor
 1. Eine Ansammlung von Elementen
 2. Plus Operationen auf dieser Ansammlung
- In C++ heissen **vector** $\langle T \rangle$ und ähnliche „Ansammlungs“-Datenstrukturen *Container*
- In manchen Sprachen, z.B. Java, *Collections* genannt

Container-Eigenschaften

- Jeder Container hat bestimmte *charakteristische Eigenschaften*
- Ein Array-basierter Vektor z.B. die folgenden:
 - Effizienter, index-basierter Zugriff ($v[i]$)
 - Effiziente Speichernutzung: Nur die Elemente selbst belegen Platz (plus Elementezähler)
 - Einfügen/Entfernen an beliebigem Index ist potenziell ineffizient
 - Suchen eines bestimmten Elements ist potenziell ineffizient
 - Kann Elemente mehrfach enthalten
 - Elemente sind in Einfügereihenfolge enthalten (geordnet aber unsortiert)

Container in C++

- Fast jede Anwendung erfordert die Verwaltung und Manipulation von beliebig vielen Datensätzen
- Aber mit unterschiedlichen Anforderungen (z.B. Elemente nur hinten anhängen, fast nie entfernen, oft suchen, ...)
- Deswegen enthält die Standardbibliothek von C++ diverse Container mit unterschiedlichen Eigenschaften, siehe <https://en.cppreference.com/w/cpp/container>
- Viele weitere sind über Bibliotheken Dritter verfügbar, z.B. https://www.boost.org/doc/libs/1_68_0/doc/html/container.html, <https://github.com/abseil/abseil-cpp>

Beispiel-Container:

`std::unordered_set<T>`

- Eine *mathematische Menge* ist eine ungeordnete, duplikatfreie Zusammenfassung von Elementen:

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

- In C++: `std::unordered_set<T>`

- Eigenschaften:

- Kann kein Element doppelt enthalten
- Elemente haben keine bestimmte Reihenfolge
- Kein indexbasierter Zugriff (`s[i]` nicht definiert)
- Effiziente „Element enthalten?“-Prüfung
- Effizientes Einfügen und Löschen von Elementen

- Randbemerkung: Implementiert als Hash-Tabelle

Anwendungsbeispiel

`std::unordered_set<T>`

Problem:

- Gegeben eine Sequenz an Paaren (*Name*, *Prozente*) von Code-Expert-Submissions ...

```
// Input: file submissions.txt
Friedrich 90
Schwerhoff 10
Lehner 20
Schwerhoff 11
```

- ... bestimme die Abgebenden, die mindestens 50% erzielt haben

```
// Output
Friedrich
```

Anwendungsbeispiel

`std::unordered_set<T>`

```
std::ifstream in("submissions.txt");  
std::unordered_set<std::string> names;  
  
std::string name;  
unsigned int score;  
  
while (in >> name >> score) {  
    if (50 <= score)  
        names.insert(name);  
}  
  
std::cout << "Unique submitters: "  
          << names << '\n';
```

← Öffne `submissions.txt`

← Namen-Menge, initial leer

← Paar (*Name, Punkte*)

← Nächstes Paar einlesen
Namen merken falls
Punkte ausreichen

← Gemerkte Namen ausgeben

Beispiel-Container: `std::set<T>`

- Fast gleich wie `std::unordered_set<T>`, aber die Elemente sind *geordnet*

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

- Elemente suchen, einfügen und löschen weiterhin effizient (besser als bei `std::vector<T>`), aber weniger effizient als bei `std::unordered_set<T>`
- Denn das Beibehalten der Ordnung zieht etwas Aufwand nach sich
- Randbemerkung: Implementiert als Rot-Schwarz-Baum

Anwendungsbeispiel `std::set<T>`

```
std::ifstream in("submissions.txt");
std::set<std::string> names; ← set statt unordered_set ...

std::string name;
unsigned int score;

while (in >> name >> score) {
    if (50 <= score)
        names.insert(name);
}

std::cout << "Unique submitters: "
           << names << '\n'; ← ... und die Ausgabe erfolgt alphabetisch sortiert
```

Container Ausgeben

- Bereits gesehen: `avec::print()` und `llvec::print()`
- Wie sieht's mit der Ausgabe von `set`, `unordered_set`, ... aus?
- Gemeinsamkeit: Über Container-Elemente iterieren und diese ausgeben

Ähnliche Funktionen

- Viele weitere nützliche Funktionen können mittels Container-Iteration implementiert werden:
- `contains(c, e)`: wahr gdw. Container `c` Element `e` enthält
- `min/max(c)`: Gibt das grösste/kleinste Element zurück
- `sort(c)`: Sortiert die Elemente von `c`
- `replace(c, e1, e2)`: Ersetzt alle `e1` in `c` mit `e2`
- `sample(c, n)`: Wählt zufällig `n` Elemente aus `c` aus
- ...

Zur Erinnerung: Iterieren mit Zeigern

■ Iteration über ein *Array*:

- Auf Startelement zeigen: `p = this->arr`
- Auf aktuelles Element zugreifen: `*p`
- Überprüfen, ob Ende erreicht:
`p == this->arr + size`
- Zeiger vorrücken: `p = p + 1`



■ Iteration über eine *verkettete Liste*:

- Auf Startelement zeigen: `p = this->head`
- Auf aktuelles Element zugreifen: `p->value`
- Überprüfen, ob Ende erreicht:
`p == nullptr`
- Zeiger vorrücken: `p = p->next`

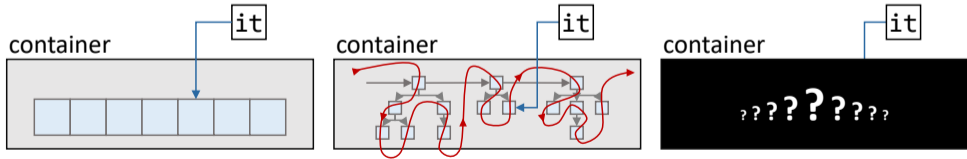


Iteratoren

- Iteration erfordert nur die vier eben gesehenen Operationen
- Aber deren Implementierung hängt vom Container ab
- \Rightarrow Jeder C++-Container implementiert seinen eigenen *Iterator*
- Gegeben ein Container `c`:
 - `it = c.begin()`: Iterator aufs erste Element
 - `it = c.end()`: Iterator *hinters* letzte Element
 - `*it`: Zugriff aufs aktuelle Element
 - `++it`: Iterator um ein Element verschieben
- Iteratoren sind quasi gepimpte Zeiger

Iteratoren

- Iteratoren ermöglichen Zugriff auf verschiedene Container auf *uniforme* Weise: `*it`, `++it`, etc.
- Nutzer bleiben unabhängig von der Container-Implementierung
- Iterator weiss, wie man die Elemente „seines“ Containers abläuft
- Nutzer müssen und sollen interne Details nicht kennen
- \Rightarrow Containerimplementierung kann geändert werden, ohne das Nutzer Code ändern müssen



Beispiel: Iteration über `std::vector`

```
std::vector<int> v = {1, 2, 3};
```

`it` ist ein zu `std::vector<int>` passender Iterator

`it` zeigt initial aufs erste Element

```
for (std::vector<int>::iterator it = v.begin(),
```

```
it != v.end();
```

Abbruch falls `it` Ende erreicht hat

```
++it)
```

`it` elementweise vorwärtssetzen

```
{
```

```
    *it = -*it;
```

Aktuelles Element negieren ($e \rightarrow -e$)

```
}
```

```
std::cout << v; // -1 -2 -3
```

Beispiel: Iteration über `std::vector`

Zur Erinnerung: Type-Aliasse können genutzt werden um oft genutzte Typnamen abzukürzen

```
using ivit = std::vector<int>::iterator; // int-vector iterator

for (ivit it = v.begin();
     ...
```

Negieren als Funktion

Wie zuvor: Übergabe eines *Arbeitsbereichs* (-intervalls)

```
void neg(std::vector<int>::iterator begin;  
        std::vector<int>::iterator end) {  
  
    for (std::vector<int>::iterator it = begin;  
         it != end;  
         ++it) {  
  
        *it = -*it;  
    }  
}
```

← Elemente im Intervall [begin, end) negieren

Negieren als Funktion

Wie zuvor: Übergabe eines *Arbeitsbereichs* (-intervalls)

```
void neg(std::vector<int>::iterator begin;  
        std::vector<int>::iterator end);
```

```
// in main():  
std::vector<int> v = {1, 2, 3};  
neg(v.begin(), v.begin() + (v.size() / 2)); ← Erste Hälfte negieren
```

Algorithmen-Bibliothek in C++

- Die C++-Standardbibliothek enthält viele nützliche Algorithmen (Funktionen), die auf durch Iteratoren bestimmten Intervallen [*Anfang*, *Ende*) arbeiten
- Zum Beispiel **find**, **fill** and **sort**; siehe auch <https://en.cppreference.com/w/cpp/algorithm>
- Dank Iteratoren können diese ≥ 100 (!) Algorithmen auf beliebigen* Containern ausgeführt werden: Den 17 (!) Standardcontainern von C++, auf unserem **avec** und **llvec** (kommt gleich), etc.
- Gäbe es diesen uniformen Zugriff auf Container-elemente nicht, müsste *sehr* viel Code dupliziert werden

Ein Iterator für `llvec`

Wir brauchen:







1. Einen `llvec`-spezifischen Iterator mit mindestens folgender Funktionalität:
 - Zugriff aktuelles Element: `operator*`
 - Iterator vorwärtssetzen: `operator++`
 - Ende-Erreicht-Prüfung: `operator!=` (oder `operator==`)
2. Memberfunktionen `begin()` und `end()` für `llvec` um einen Iterator auf den Anfang bzw. hinter das Ende zu erhalten

Iterator `l1vec::iterator` (Schritt 1/2)

```
class l1vec {  
    ...  
public:  
    class iterator {  
        ...  
    };  
  
    ...  
}
```

- Der Iterator gehört zu unserem Vektor, daher ist `iterator` eine öffentliche *innere Klasse* von `l1vec`
- Instanzen unseres Iterators sind vom Typ `l1vec::iterator`

Iterator `llvec::iterator` (Schritt 1/2)

```
class iterator {  
    llnode* node;  Zeiger auf aktuelles Vektor-Element  
  
public:  
    iterator(llnode* n);  Erzeuge Iterator auf bestimmtes Element  
    iterator& operator++();  Iterator ein Element vorwärtssetzen  
    int& operator*() const;  Zugriff auf aktuelles Element  
    bool operator!=(const iterator& other) const;   
};  
 Vergleich mit anderem Iterator
```

Iterator `llvec::iterator` (Schritt 1/2)

```
// Constructor
```

```
llvec::iterator::iterator(llnode* n): node(n) ← {}
```

Iterator initial auf `n` zeigen lassen

```
// Pre-increment
```

```
llvec::iterator& llvec::iterator::operator++() {  
    assert(this->node != nullptr);
```

```
    this->node = this->node->next; ← Iterator ein Element vorwärtssetzen
```

```
    return *this; ← Referenz auf verschobenen Iterator zurückgeben
```

```
}
```

Iterator `llvec::iterator` (Schritt 1/2)

```
// Element access
int& llvec::iterator::operator*() const {
    return this->node->value; ← Zugriff auf aktuelles Element
}
```

```
// Comparison: when are two iterators not equal?
bool llvec::iterator::operator!=(
    const llvec::iterator& other) const
{
    return this->node != other.node; ←
}
```

this Iterator verschieden von **other** falls sie auf unterschiedliche Elemente zeigen

Ein Iterator für `l1vec` (Wiederholung)

Wir brauchen:

1. Einen `l1vec`-spezifischen Iterator mit mindestens folgender Funktionalität:

- Zugriff aktuelles Element: `operator*`
- Iterator vorwärtssetzen: `operator++`
- Ende-Erreicht-Prüfung: `operator!=` (oder `operator==`)



2. Memberfunktionen `begin()` und `end()` für `l1vec` um einen Iterator auf den Anfang bzw. hinter das Ende zu erhalten

Iterator `llvec::iterator` (Schritt 2/2)

```
class llvec {  
    ...  
public:  
    class iterator {...};  
  
    iterator begin();  
    iterator end();  
  
    ...  
}
```

`llvec` braucht Memberfunktionen um Iteratoren *auf den Anfang* bzw. *hinter das Ende* des Vektors herausgeben zu können

Iterator `llvec::iterator` (Schritt 2/2)

```
llvec::iterator llvec::begin() {  
    return llvec::iterator(this->head);  
}
```

Iterator auf erstes Vektorelement

```
llvec::iterator llvec::end() {  
    return llvec::iterator(nullptr);  
}
```

Iterator hinter letztes Vektorelement

Const-Iteratoren

- Neben `iterator` sollte jeder Container auch einen *Const-Iterator* `const_iterator` bereitstellen
- Const-Iteratoren gestatten nur Lesezugriff auf den darunterliegenden Container
- Zum Beispiel für `llvec`:

```
llvec::const_iterator llvec::cbegin() const;
llvec::const_iterator llvec::cend() const;

const int& llvec::const_iterator::operator*() const;
...
```

- Daher nicht möglich (Compilerfehler): `*(v.cbegin()) = 0`

Const-Iteratoren

Const-Iterator *kann* verwendet werden um nur Lesen zu erlauben:

```
llvec v = ...;
for (llvec::const_iterator it = v.cbegin(); ...)
    std::cout << *it;
```

Hier könnte auch der nicht-const **iterator** verwendet werden

Const-Iteratoren

Const-Iterator *muss* verwendet werden falls Vektor selbst const ist:

```
const llvec v = ...;
for (llvec::const_iterator it = v.cbegin(); ...)
    std::cout << *it;
```

Hier kann nicht der **iterator** verwendet werden (Compilerfehler)

Bereichsbasierte for-Schleife

- Sequenzielle Iteration mittels eines Iterators über einen `llvec` (const-Iterator möglich; andere Container möglich):

```
llvec v(3); // v == {0, 0, 0}
for (llvec::iterator it = v.begin(); it != v.end(); ++it)
    std::cout << *it; // 000
```

- Kann alternativ auch wie folgt geschrieben werden:

```
for (int i : v) std::cout << i; // 000
```

Wird dann zu Iterator-basierter Schleife übersetzt.

- Modifizierender Zugriff ist auch möglich:

```
for (int& i : v) i += 3;
for (int i : v) std::cout << i; // 369
```

Typ-generischer Container

Typ-spezifischer Container



Typ-generischer Container



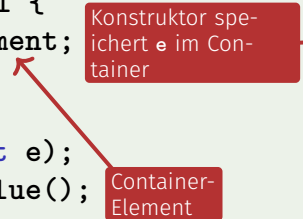
https://upload.wikimedia.org/wikipedia/commons/d/df/Container_01_KMJ.jpg (CC BY-SA 3.0)

P.S.: Templates sind nicht klausurrelevant


Typ-generischer Container

Klasse `cell`: Ein einfacher, 1-elementiger Container für `int`

```
class cell {  
    int element;  
  
public:  
    cell(int e);  
    int& value();  
};
```



```
cell::cell(int e)  
: element(e) {}  
  
int& cell::value() {  
    return this->element;  
}
```



Besser: Generische `cell<E>` für jeden Elementtyp `E` (analog zu `std::vector<E>`)

Typ-generischer Container mit Templates

Templates ermöglichen *Typ-generische* Funktionen und Klassen:

```
template<typename E> ← Sei E ein beliebiger Typ ...  
class cell {  
    E element;  
  
public:  
    cell(E e);  
    E& value();  
};  
    ← ... dann verwaltet cell ein Element vom Typ E
```

- Typen können als *Parameter* genutzt werden
- Typ-Parameter sind im „templatierten“ Gültigkeitsbereich

Typ-generischer Container mit Templates

- Signaturen und Implementierungen müssen „templatiert“ werden
- Bei separaten Implementierungen muss das Klassen-Präfix in generischer Form angegeben werden

```
template<typename E>
class cell {
    E element;

public:
    cell(E e);
    E& value();
};
```

```
template<typename E>
cell<E>::cell(E e)
    : element(e) {}

template<typename E>
E& cell<E>::value() {
    return this->element;
}
```

Typ-generischer Container mit Templates

```
cell<int> c1(313);  
cell<std::string> c2("terrific!")
```

- Typ-Parameter müssen bei *Deklarationen*, z.B. `cell<int>`, explizit angegeben werden ...
- ...aber überall sonst werden sie vom Compiler *inferriert*, z.B. bei `c1(313)`, d.h. beim Aufruf des generischen Konstruktors `cell(E e)` (wobei Typparameter **E** vom Compiler mit `int` instanziiert wird)

Mehr Templates: generischer Ausgabe-Operator

- **Ziel:** Ein *generischer* Ausgabe-Operator `<<` für *iterierbare Container*: `llvec`, `avec`, `std::vector`, `std::set`, ...
- D.h. `std::cout << c << '\n'` soll für jeden solchen Container `c` funktionieren

Mehr Templates: generischer Ausgabe-Operator

- Generischer Ausgabe-Operator mit zwei Typ-Parametern

```
template <typename S, typename C>  
S& operator<<(S& sink, const C& container);
```

Intuition: Operator funktioniert für jeden Ausgabestrom `sink` vom Typ `S` und jeden Container `container` vom Typ `C`

Mehr Templates: generischer Ausgabe-Operator

- Generischer Ausgabe-Operator mit zwei Typ-Parametern

```
template <typename S, typename C>  
S& operator<<(S& sink, const C& container);
```

- Der Compiler inferiert passende Typen aus den Aufrufargumenten

```
std::set<int> s = ...;  
std::cout << s << '\n'; ← S = std::ostream, C = std::set<int>
```

Mehr Templates: generischer Ausgabe-Operator

Implementierung von `<<` *schränkt S und C ein* (Compilerfehler falls nicht erfüllt):

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container) {
    for (typename C::const_iterator it = container.begin();
         it != container.end();
         ++it) {
        sink << *it << ' ';
    }

    return sink;
}
```

C muss Iteratoren bereitstellen – mit passenden Funktionen

Mehr Templates: generischer Ausgabe-Operator

Implementierung von `<<` *schränkt S und C ein* (Compilerfehler falls nicht erfüllt):

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container) {
    for (typename C::const_iterator it = container.begin();
         it != container.end();
         ++it) {

        sink << *it << ' ' << '; ← S muss Ausgabe von Elementen
                                   (*it) und Zeichen ( ' ' ) unterstützen
    }

    return sink;
}
```

Templates: Abschluss

- Templates realisieren in C++ *statische Codegenerierung* bzw. *statische Metaprogrammierung*
- Template-Code wird pro Typ-Instanziierung *kopiert*. Bei Benutzung von `cell<int>` und `cell<std::string>` legt der Compiler zwei *instanziierte Kopien* des `cell`-Codes an: sozusagen die beiden (nicht mehr generischen) Klassen `cell_int` und `cell_stdstring`.
- Templates reduzieren Codeduplikation und fördern Code-Wiederverwendbarkeit
- Leider sind Compiler-Fehlermeldungen, die sich auf Templates beziehen, oft noch komplexer, als es C++-Fehlermeldungen ohnehin schon oft sind

23. Dynamische Datentypen und Speicherverwaltung

Problem

Letzte Woche: Dynamischer Datentyp

Haben im Vektor dynamischen Speicher angelegt, aber nicht wieder freigegeben. Insbesondere: keine Funktionen zum Entfernen von Elementen aus **llvec**.

Heute: Korrektes Speichermanagement!

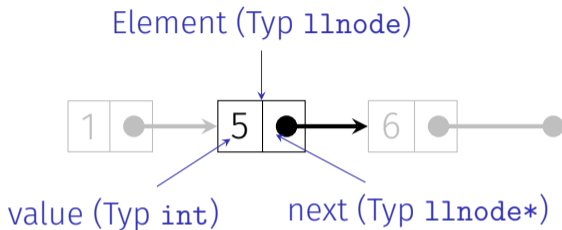
Ziel: Stapelklasse mit Speichermanagement

```
class stack{
public:
    // post: Element auf den Stapel legen
    void push(int value);
    // pre: Stack nicht leer
    // post: Entfernt oberstes Element vom Stapel
    void pop();
    // pre: Stack nicht leer
    // post: Gibt Wert des obersten Elementes zurück
    int top() const;
    // post: gibt zurück, ob Stack leer ist
    bool empty() const;
    // post: gibt den Stapel aus
    void print(std::ostream& out) const;

```

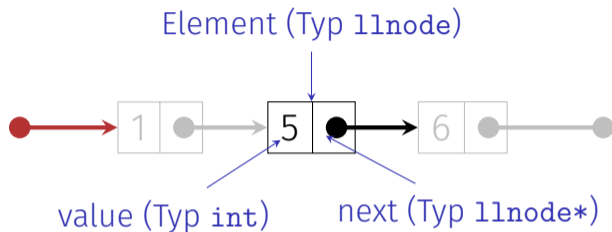
...

Erinnerung: Verkettete Liste



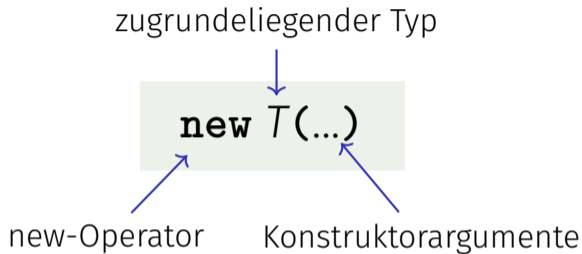
```
struct llnode {  
    int value;  
    llnode* next;  
    // constructor  
    llnode (int v, llnode* n) : value (v), next (n) {}  
};
```

Stapel = Zeiger aufs oberste Element



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

Erinnerung: der `new`-Ausdruck



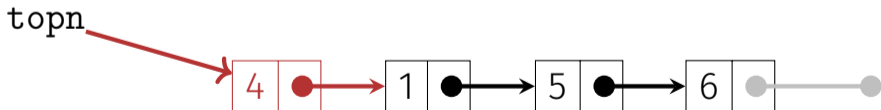
- **Effekt:** Speicher für ein neues Objekt vom Typ T wird alloziert ...
- ... und mit Hilfe des passenden Konstruktors initialisiert
- **Wert:** Adresse des neuen T -Objekt, **Typ:** Zeiger \mathbf{T}^* !

Der `new`-Ausdruck:

`push(4)`

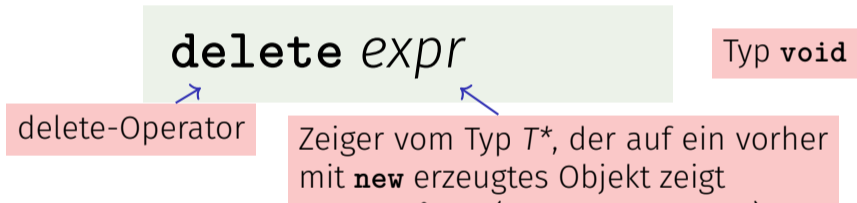
- **Effekt:** Neues Objekt vom Typ T wird im Speicher angelegt
...
- ... und mit Hilfe des passenden Konstruktors initialisiert.
- **Wert:** Adresse des neuen Objekts

```
void stack::push(int value) {  
    topn = new llnode(value, topn);  
}
```



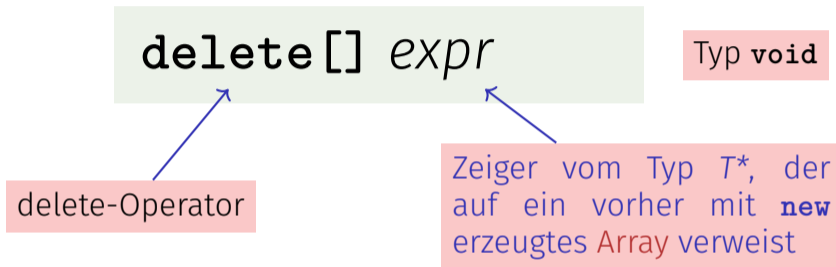
Der delete-Ausdruck

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



- **Effekt:** Objekt wird **dekonstruiert** (Erklärung folgt)
... und **Speicher wird freigegeben**.

Der delete-Ausdruck für Arrays



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

Wer geboren wird, muss sterben...

Richtlinie “Dynamischer Speicher”

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

Nichtbeachtung führt zu *Speicherlecks*:

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

Aufpassen mit `new` und `delete`!

```
rational* t = new rational; ← Speicher für t wird angelegt  
rational* s = t; ← Auch andere Zeiger können auf das Objekt zeigen..  
delete s; ← ... und zur Freigabe verwendet werden.  
int nominator = (*t).denominator(); ← Fehler: Speicher freigegeben!
```

Dereferenzieren eines „dangling pointers“

- Zeiger auf freigegebene Objekte: hängende Zeiger (*dangling pointers*)
- Mehrfache Freigabe eines Objektes mit **delete** ist ein ähnlicher schwerer Fehler.

Weiter mit dem Stapel:

pop()

```
void stack::pop(){  
    assert (!empty());  
    llnode* p = topn;  
    topn = topn->next;  
    delete p;  
}
```

Erinnerung: Abkürzung für (*topn).next

topn

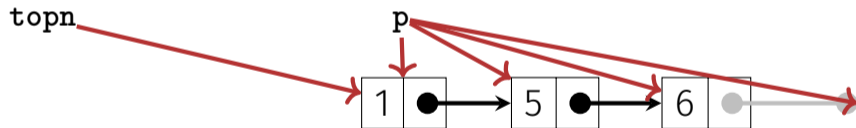
p



Stapel ausgeben:

print()

```
void stack::print (std::ostream& out) const {  
    for(const llnode* p = topn; p != nullptr; p = p->next)  
        out << p->value << " "; // 1 5 6  
}
```



Stapel ausgeben:

operator«

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

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

empty(), top()

```
bool stack::empty() const {  
    return top == nullptr;  
}
```

```
int stack::top() const {  
    assert(!empty());  
    return topn->value;  
}
```

Leerer Stapel

```
class stack{
public:
    stack() : topn (nullptr) {} // default constructor

    void push(int value);
    void pop();
    void print(std::ostream& out) const;
    int top() const;
    bool empty() const;
private:
    llnode* topn;
}
```

Zombie-Elemente

```
{  
  stack s1; // lokale Variable  
  s1.push (1);  
  s1.push (3);  
  s1.push (2);  
  std::cout << s1 << "\n"; // 2 3 1  
}  
// s1 ist gestorben (nicht mehr zugreifbar)
```

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

Der Destruktor

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

$$\sim T ();$$

- Wird automatisch aufgerufen, wenn die Speicherdauer eines Klassenobjekts vom Typ \mathbf{T} endet – z.B. bei Aufruf von **delete** auf einem Objekt vom Typ \mathbf{T}^* oder wenn der Gültigkeitsbereich eines Objektes vom Typ \mathbf{T} endet.
- Falls kein Destruktor deklariert ist, so wird er automatisch erzeugt und ruft die Destruktoren für die Membervariablen auf (Zeiger \mathbf{topn} , kein Effekt – Grund für Zombie-Elemente)

Mit dem Destruktor klappt's!

```
// POST: the dynamic memory of *this is deleted
stack::~~stack(){
    while (topn != nullptr){
        llnode* t = topn;
        topn = t->next;
        delete t;
    }
}
```

- löscht automatisch alle Stapелеlemente, wenn der Stapel ungültig wird
- Unsere Stapel-Klasse scheint jetzt die Richtlinie “Dynamischer Speicher” zu befolgen (?)

Stapel fertig?

Offenbar noch nicht...

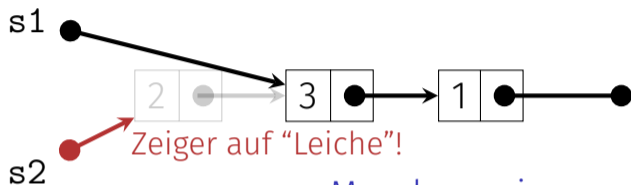
```
stack s1;  
s1.push (1);  
s1.push (3);  
s1.push (2);  
std::cout << s1 << "\n"; // 2 3 1
```

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

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

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

Was ist hier schiefgegangen?



Memberweise

Initialisierung:

kopiert nur den topn-Zeiger

```
...  
stack s2 = s1; ←  
std::cout << s2 << "\n"; // 2 3 1  
  
s1.pop ();  
std::cout << s1 << "\n"; // 3 1  
  
s2.pop (); // Oops, Programmabsturz!
```

Das eigentliche Problem

Schon das geht schief:

```
{  
    stack s1;  
    s1.push(1);  
    stack s2 = s1;  
}
```

Beim Verlassen des Gültigkeitsbereiches werden beide Stacks aufgeräumt (dekonstruiert). Aber beide Stacks versuchen dieselben Daten zu löschen, denn sie haben **Zugriff auf denselben Zeiger**.

Lösungsmöglichkeiten

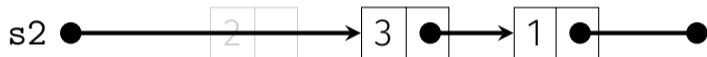
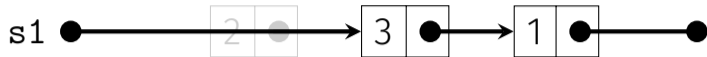
Smart-Pointers (werden hier nicht weiter vertieft):

- Zähle die Anzahl Zeiger, die auf ein Objekt verweisen. Lösche nur wenn diese Anzahl auf 0 zurückfällt:
std::shared_pointer
- Verhindere, dass mehrere Zeiger auf ein Objekt zeigen können: **std::unique_pointer**.

oder:

- Wir erstellen eine echte Kopie aller Daten – wie folgt.

Wir erstellen eine echte Kopie!



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

Der Copy-Konstruktor

- Der Copy-Konstruktor einer Klasse T ist der eindeutige Konstruktor mit Deklaration

$$T(\text{const } T\& x);$$

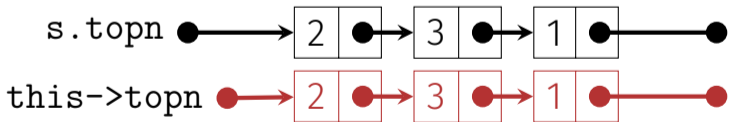
- wird automatisch aufgerufen, wenn Werte vom Typ T mit Werten vom Typ T *initialisiert* werden

$$T\ x = t; \quad (t \text{ vom Typ } T)$$
$$T\ x(t);$$

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

Mit dem Copy-Konstruktor klappt's!

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
    if (s.topn == nullptr) return;
    topn = new llnode(s.topn->value, nullptr);
    llnode* prev = topn;
    for(llnode* n = s.topn->next; n != nullptr; n = n->next){
        llnode* copy = new llnode(n->value, nullptr);
        prev->next = copy;
        prev = copy;
    }
}
```



prev

NB: rekursives Kopieren

```
llnode* copy (node* that){  
    if (that == nullptr) return nullptr;  
    return new llnode(that->value, copy(that->next));  
}
```

Elegant, oder? Warum haben wir das nicht gleich so gemacht? Grund: verkettete Listen können sehr lang werden. Dann könnte **copy** zum Stapelüberlauf⁶ führen. Aufrufstapel ist nämlich meist kleiner als Heapspeicher.

⁶nicht von dem Stapel, den wir gerade implementieren, sondern vom Aufrufstapel der Rekursion

Initialisierung \neq Zuweisung!

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

```
stack s2;  
s2 = s1; // Zuweisung
```

```
s1.pop ();  
std::cout << s1 << "\n"; // 3 1  
s2.pop (); // Oops, Programmabsturz!
```

Der Zuweisungsoperator

- Überladung von **operator=** als Memberfunktion
- Wie Copy-Konstruktor ohne Initialisierer, aber zusätzlich
 - Freigabe des Speichers für den „alten“ Wert
 - Prüfen auf Selbstzuweisungen (`s1=s1`), die keinen Effekt haben sollen
- Falls kein Zuweisungsoperator deklariert ist, so wird er automatisch erzeugt (und weist memberweise zu – Grund für obiges Problem)

Mit dem Zuweisungsoperator klappt's!

```
// POST: *this (left operand) becomes a
//           copy of s (right operand)
stack& stack::operator= (const stack& s){
    if (topn != s.topn){ // keine Selbstzuweisung
        stack copy = s; // Kopierkonstruktor
        std::swap(topn, copy.topn); // copy hat nun den Müll!
    } // copy wird aufgeräumt -> Dekonstruktion
    return *this; // Rueckgabe als L-Wert (Konvention)
}
```

Cooler Trick! 😊

Fertig

```
class stack{
public:
    stack(); // constructor
    ~stack(); // destructor
    stack(const stack& s); // copy constructor
    stack& operator=(const stack& s); // assignment operator

    void push(int value);
    void pop();
    int top() const;
    bool empty() const;
    void print(std::ostream& out) const;
private:
    llnode* topn;
}
```

Dynamischer Datentyp

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Stapel)
 - Mindestfunktionalität:
 - Konstruktoren
 - Destruktor
 - Copy-Konstruktor
 - Zuweisungsoperator
- Dreierregel:** definiert eine Klasse eines davon, so muss sie auch die anderen zwei definieren!

Bäume

Bäume sind

- Verallgemeinerte Listen: Knoten können mehrere Nachfolger haben
- Spezielle Graphen: Graphen bestehen aus Knoten und Kanten. Ein Baum ist ein zusammenhängender, gerichteter, azyklischer Graph.

Bäume

Verwendung

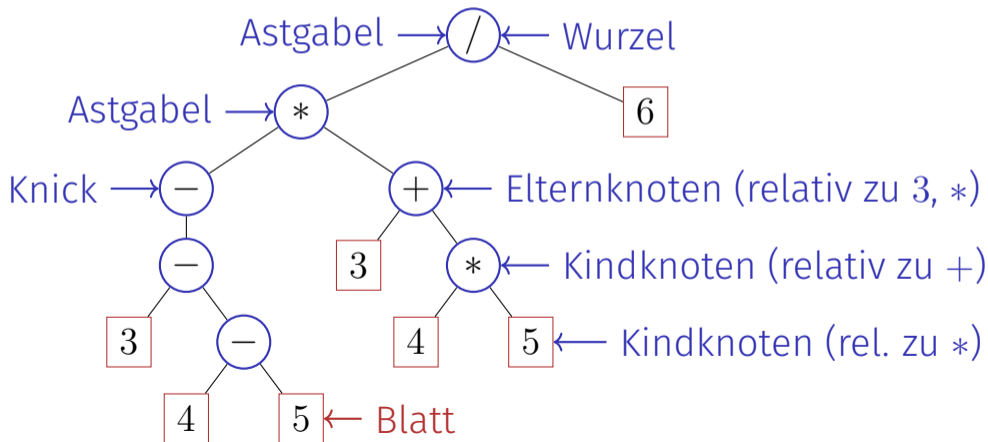
- Entscheidungsbäume: Hierarchische Darstellung von Entscheidungsregeln
- Codebäume: Darstellung eines Codes, z.B. Morsealphabet, Huffman Code
- Suchbäume: ermöglichen effizientes Suchen eines Elementes
- Syntaxbäume: Parsen und Traversieren von Ausdrücken, z.B. in einem Compiler



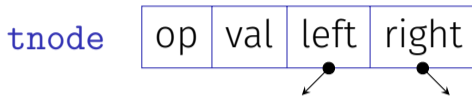
Bäumen werden in anderen Vorlesungen vertiefter behandelt (Datenstrukturen und Algorithmen (RW) / Algorithmen und Komplexität (Mathematik Bachelor))

(Ausdrucks-)Bäume

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



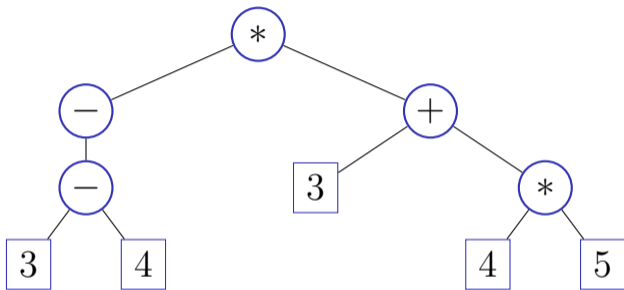
Knoten (struct tnode)



```
struct tnode {
    char op; // leaf node: op is '='
             // internal node: op is '+', '-', '*', or '/'
    double val;
    tnode* left; // == nullptr for unary minus
    tnode* right;

    tnode(char o, double v, tnode* l, tnode* r)
        : op(o), val(v), left(l), right(r) {}
};
```

Grösse = Knoten in Teilbäumen zählen



- Grösse eines Blattes: 1
- Grösse anderer Knoten: 1 + Gesamtgrösse aller Kindknoten
- Z.B. Grösse des „+“-Knoten ist 5

Knoten in Teilbäumen zählen

```
// POST: returns the size (number of nodes) of
//       the subtree with root n
int size (const tnode* n) {
    if (n){ // shortcut for n != nullptr
        return size(n->left) + size(n->right) + 1;
    }
    return 0;
}
```



Teilbäume auswerten

```
// POST: evaluates the subtree with root n
```

```
double eval(const tnode* n){
```

```
    assert(n);
```

```
    if (n->op == '=') return n->val; ← Blatt...
```

```
    double l = 0; ...oder Astgabel:
```

```
    if (n->left) l = eval(n->left); ← op unär, oder linker Ast
```

```
    double r = eval(n->right); ← rechter Ast
```

```
    switch(n->op){
```

```
        case '+': return l+r;
```

```
        case '-': return l-r;
```

```
        case '*': return l*r;
```

```
        case '/': return l/r;
```

```
        default: return 0;
```

```
    }
```

```
}
```



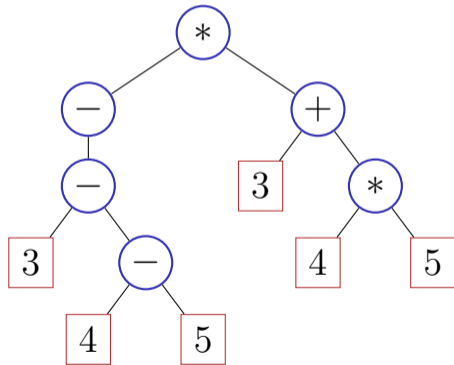
Teilbäume klonen

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



Teilbäume fällen

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



Teilbäume nutzen

```
// Construct a tree for 1 - (-(3 + 7))
tnode* n1 = new tnode( '=', 3, nullptr, nullptr);
tnode* n2 = new tnode( '=', 7, nullptr, nullptr);
tnode* n3 = new tnode( '+', 0, n1, n2);
tnode* n4 = new tnode( '-', 0, nullptr, n3);
tnode* n5 = new tnode( '=', 1, nullptr, nullptr);
tnode* root = new tnode( '-', 0, n5, n4);

// Evaluate the overall tree
std::cout << "1 - (-(3 + 7)) = " << eval(root) << '\n';

// Evaluate a subtree
std::cout << "3 + 7 = " << eval(n3) << '\n';

clear(root); // free memory
```


Bäume pflanzen

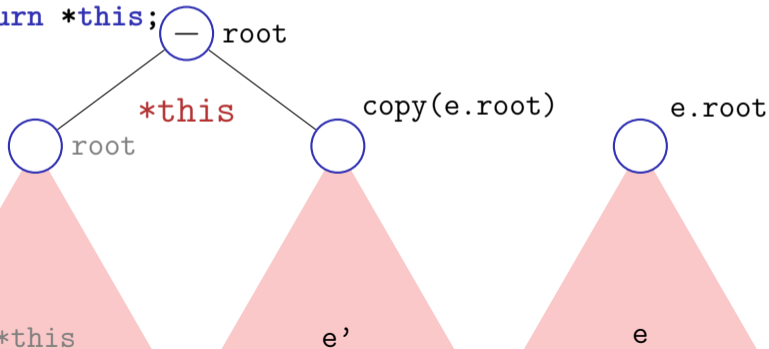
```
class texpression {  
public:  
    texpression (double d)   
        : root (new tnode ('=', d, 0, 0)) {}  
    ...  
private:  
    tnode* root;  
};
```

erzeugt Baum mit
einem Blatt



Bäume wachsen lassen

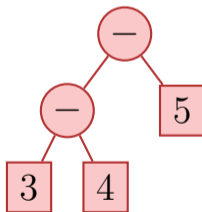
```
texpression& texpression::operator-= (const texpression& e)
{
    assert (e.root);
    root = new tnode ('-', 0, root, copy(e.root));
    return *this;
}
```



Bäume züchten

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

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



Dreierregel: Bäume klonen, reproduzieren und fällen

```
texpression::~~texpression(){  
    clear(root);  
}
```

```
texpression::texpression (const texpression& e)  
    : root(copy(e.root)) { }
```

```
texpression& texpression::operator=(const texpression& e){  
    if (root != e.root){  
        texpression cp = e;  
        std::swap(cp.root, root);  
    }  
    return *this;  
}
```

Zusammengefasst

```
class texpression{
public:
    texpression (double d); // constructor
    ~texpression(); // destructor
    texpression (const texpression& e); // copy constructor
    texpression& operator=(const texpression& e); // assignment op
    texpression operator-();
    texpression& operator-=(const texpression& e);
    texpression& operator+=(const texpression& e);
    texpression& operator*=(const texpression& e);
    texpression& operator/=(const texpression& e);
    double evaluate();
private:
    tnode* root;
};
```

Werte zu Bäumen!

```
using number_type = texpression ;
```

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

```
number_type term (std::istream& is){
```

```
    number_type value = factor (is);
```

```
    while (true) {
```

```
        if (consume (is, '*'))
```

```
            value *= factor (is);
```

```
        else if (consume (is, '/'))
```

```
            value /= factor (is);
```

```
    else
```

```
        return value;
```

```
    }
```

```
}
```

double_calculator.cpp

(Ausdruckswert)

→

texpression_calculator.cpp

(Ausdrucksbaum)

Abschliessende Bemerkung

- Wir haben in dieser Vorlesung die Knoten für Liste und Baum bewusst ohne Memberfunktionen implementiert. Wir betrachten sie nämlich als reine Datencontainer ohne eigene Intelligenz.⁷
- Wenn Vererbung und Polymorphie im Spiel ist, ist die Implementation der Funktionalität wie **evaluate**, **print**, **clear**, **copy** (etc.) mit Memberfunktionen vorzuziehen.
- In jedem Falle implementiert man die Speicherverwaltung der zusammengesetzten Datenstruktur Liste / Baum nicht in den Knotenklassen.

⁷Teile der Implementation waren so sogar einfacher, da der Fall `n==nullptr` einfacher abgefangen werden kann

24. Subtyping, Polymorphie und Vererbung

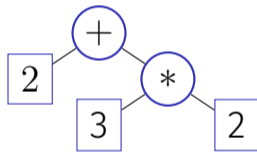
Ausdrückbäume, Aufgabenteilung und Modularisierung, Typhierarchien, virtuelle Funktionen, dynamische Bindung, Code-Wiederverwendung, Konzepte der objektorientierten Programmierung

Letzte Woche: Ausdrucksbäume

- Ziel: Arithmetische Ausdrücke repräsentieren, z.B.

$$2 + 3 * 2$$

- Arithmetische Ausdrücke bilden eine *Baumstruktur*

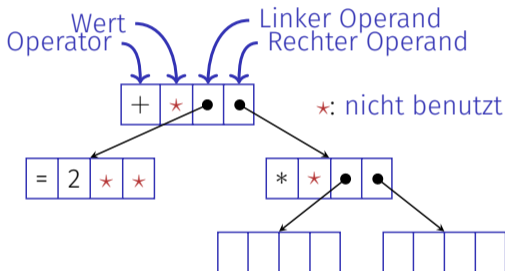


- Ausdrucksbäume bestehen aus *unterschiedlichen* Knoten: Literale (z.B. 2), binäre Operatoren (z.B. +), unäre Operatoren (z.B. $\sqrt{\quad}$), Funktionsanwendungen (z.B. \cos), etc.

Nachteile

Implementiert mittels *eines einzigen* Knotentyps:

```
struct tnode {
  char op; // Operator ('=' for literals)
  double val; // Literal's value
  tnode* left; // Left child (or nullptr)
  tnode* right; // ...
  ...
};
```



Beobachtung: `tnode` ist die „Summe“ aller benötigten Knoten (Konstanten, Addition, ...) \Rightarrow Speicherverschwendung, unelegant

Nachteile

Beobachtung: `tnode` ist die „Summe“ aller benötigten Knoten – und jede Funktion muss diese „Summe“ wieder „auseinander nehmen“, z.B.:

```
double eval(const tnode* n) {
    if (n->op == '=') return n->val; // n is a constant
    double l = 0;
    if (n->left) l = eval(n->left); // n is not a unary operator
    double r = eval(n->right);
    switch(n->op) {
        case '+': return l+r; // n is an addition node
        case '*': return l*r; // ...
        ...
    }
}
```

⇒ Umständlich und somit fehleranfällig

Nachteile

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

```
double eval(const tnode* n) {  
    if (n->op == '=') return n->val;  
    double l = 0;  
    if (n->left) l = eval(n->left);  
    double r = eval(n->right);  
    switch(n->op) {  
        case '+': return l+r;  
        case '*': return l*r;  
        ...  
    }
```

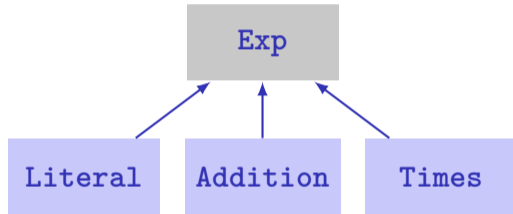
Dieser Code ist nicht *modular* – das ändern wir heute!

Neue Konzepte heute

1. Subtyping

- Typhierarchie: **Exp** repräsentiert allgemeine Ausdrücke, **Literal** etc. sind konkrete Ausdrücke
- Jedes **Literal** etc. ist auch ein **Exp** (Subtyp-Beziehung)
- Deswegen kann ein **Literal** etc. überall dort genutzt werden, wo ein **Exp** erwartet wird:

```
Exp* e = new Literal(132);
```



Neue Konzepte heute

2. Polymorphie und dynamische Bindung

- Eine Variable vom *statischen* Typ **Exp** kann Ausdrücke mit unterschiedlichen *dynamischen* Typen „beherbergen“:

```
Exp* e = new Literal(2); // e is the literal 2
e = new Addition(e, e); // e is the addition 2 + 2
```

- Ausgeführt werden die Memberfunktionen des *dynamischen* Typs:

```
Exp* e = new Literal(2);
std::cout << e->eval(); // 2

e = new Addition(e, e);
std::cout << e->eval(); // 4
```

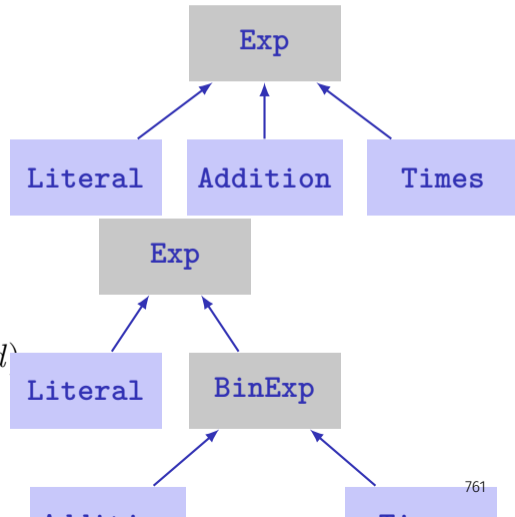
Neue Konzepte heute

3. Vererbung

- Manche Funktionalität ist für mehrere Mitglieder der Typhierarchie gleich
- Z.B. die Berechnung der Grösse (Verschachtelungstiefe) binärer Ausdrücke (**Addition**, **Times**):

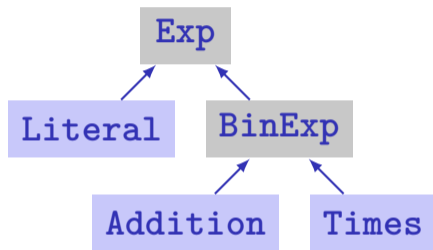
$1 + \text{size}(\text{left operand}) + \text{size}(\text{right operand})$

⇒ Funktionalität einmal implementieren und dann an Subtypen *vererben*



Vorteile

- Subtyping, Polymorphie und dynamische Bindung ermöglichen *Modularisierung durch Spezialisierung*
- Vererbung erlaubt gemeinsamen Code trotz Modularisierung
⇒ *Codeduplikation vermeiden*



```
Exp* e = new Literal(2);  
std::cout << e->eval();  
  
e = new Addition(e, e);  
std::cout << e->eval();
```


Syntax und Terminologie

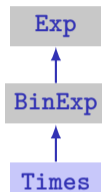
```
struct Exp {  
    ...  
}  
  
struct BinExp : public Exp {  
    ...  
}  
  
struct Times : public BinExp {  
    ...  
}
```



Anmerkung: Wir konzentrieren uns heute auf die neuen Konzepte (Subtyping, ...) und ignorieren den davon unabhängigen Aspekt der Kapselung (**class, private** vs. **public** Membervariablen)

Syntax und Terminologie

```
struct Exp {  
    ...  
}  
  
struct BinExp : public Exp {  
    ...  
}  
  
struct Times : public BinExp {  
    ...  
}
```



- **BinExp** ist eine von **Exp** *abgeleitete Klasse*¹
- **Exp** ist die *Basisklasse*² von **BinExp**
- **BinExp** *erbt* von **Exp**
- Die Vererbung von **Exp** zu **BinExp** ist *öffentlich* (**public**), daher ist **BinExp** ein *Subtyp* von **Exp**
- Analog: **Times** und **BinExp**
- Subtyprelation ist transitiv: **Times** ist ebenfalls ein Subtyp von **Exp**

¹Subklasse, Kindklasse

²Superklasse, Elternklasse

Abstrakte Klasse Exp und konkrete Klasse Literal

```
struct Exp {  
    virtual int size() const = 0;  
    virtual double eval() const = 0;  
};
```

... das macht Exp zu einer *abstrakten* Klasse

Erzwingt Implementierung durch abgeleitete Klassen ...

Aktiviert dynamische Bindung

```
struct Literal : public Exp {  
    double val;  
  
    Literal(double v);  
    int size() const;  
    double eval() const;  
};
```

Literal erbt von Exp ...

... ist aber ansonsten eine ganz normale Klasse

Literal: Implementierung

```
Literal::Literal(double v): val(v) {}
```

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

```
double Literal::eval() const {  
    return this->val;  
}
```

Subtyping: Ein Literal ist ein Ausdruck

Ein Zeiger auf einen Subtyp kann überall dort verwendet werden, wo ein Zeiger auf einen Supertyp gefordert ist:

```
Literal* lit = new Literal(5);  
Exp* e = lit; // OK: Literal is a subtype of Exp
```

Aber nicht umgekehrt:

```
Exp* e = ...  
Literal* lit = e; // ERROR: Exp is not a subtype of Literal
```

Polymorphie: Ein Literal verhält sich wie ein Literal

```
struct Exp {  
    ...  
    virtual double eval();  
};  
  
double Literal::eval() {  
    return this->val;  
}
```

```
Exp* e = new Literal(3);  
std::cout << e->eval(); // 3
```

- *Virtuelle* Memberfunktionen: der *dynamische* Typ (hier: **Literal**) bestimmt die auszuführenden Memberfunktionen
⇒ *dynamische Bindung*
- Ohne **virtual** bestimmt der *statische* Typ (hier: **Exp**) die auszuführende Funktion
- Wir vertiefen das nicht weiter

Weitere Ausdrücke: Addition und Times

```
struct Addition : public Exp {  
    Exp* left; // left operand  
    Exp* right; // right operand  
    ...  
};
```

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

```
struct Times : public Exp {  
    Exp* left; // left operand  
    Exp* right; // right operand  
    ...  
};
```

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



Aufgabenteilung



Codeduplizierung

Gemeinsamkeiten auslagern ...: BinExp

```
struct BinExp : public Exp {  
    Exp* left;  
    Exp* right;  
  
    BinExp(Exp* l, Exp* r);  
    int size() const;  
};
```

```
BinExp::BinExp(Exp* l, Exp* r): left(l), right(r) {}
```

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


...Gemeinsamkeiten erben: Addition

```
struct Addition : public BinExp {  
    Addition(Exp* l, Exp* r);  
    double eval() const;  
};
```

← Addition erbt Membervariablen (`left`, `right`) und Funktionen (`size`) von `BinExp`

```
Addition::Addition(Exp* l, Exp* r): BinExp(l, r) {}
```

```
double Addition::eval() const {  
    return  
        this->left->eval() +  
        this->right->eval();  
}
```

↑ Aufruf des *Superkonstruktors* (Konstruktor von `BinExp`) zwecks Initialisierung der Membervariablen `left` und `right`

...Gemeinsamkeiten erben: Times

```
struct Times : public BinExp {  
    Times(Exp* l, Exp* r);  
    double eval() const;  
};
```

```
Times::Times(Exp* l, Exp* r): BinExp(l, r) {}
```

```
double Times::eval() const {  
    return  
        this->left->eval() *  
        this->right->eval();  
}
```

Beobachtung: `Additon::eval()` und `Times::eval()` sind sich sehr ähnlich und könnten ebenfalls zusammengelegt werden. Das dafür notwendige Konzept der *funktionalen Programmierung* geht jedoch über diesen Kurs hinaus.

Weitere Ausdrücke und Operationen

- Weitere Ausdrücke, als von **Exp** abgeleitete Klassen, sind möglich, z.B. $-$, $/$, $\sqrt{\quad}$, \cos , \log
- Eine ehemalige Bonusaufgabe (Teil der heutigen Vorlesungsbeispiele auf Code Expert) veranschaulicht, was alles möglich ist: Variablen, trigonometrische Funktionen, Parsing, Pretty-Printing, numerische Vereinfachungen, symbolische Ableitungen, ...

Mission: Monolithisch \rightarrow modular

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

```
double eval(const tnode* n) {  
    if (n->op == '=') return n->val;  
    double l = 0;  
    if (n->left != 0) l = eval(n->left);  
    double r = eval(n->right);  
    switch(n->op) {  
        case '+': return l + r;  
        case '*': return l * r;  
        case '-': return l - r;  
        case '/': return l / r;  
        default:  
            // unknown operator  
            assert (false);  
    }  
}
```

```
int size (const tnode* n) const { ... }
```

```
...
```

```
struct Literal : public Exp {  
    double val;  
    ...  
    double eval() const {  
        return val;  
    }  
};
```

```
struct Addition : public Exp {  
    ...  
    double eval() const {  
        return left->eval() + right->eval();  
    }  
};
```

```
struct Times : public Exp {  
    ...  
    double eval() const {  
        return left->eval() * right->eval();  
    }  
};
```

```
struct Cos : public Exp {  
    ...  
    double eval() const {  
        return std::cos(argument->eval());  
    }  
};
```



Es gibt noch so viel mehr ...

Nicht gezeigt/besprochen:

- Private Vererbung (`class B : public A`)
- Subtyping und Polymorphie ohne Zeiger
- Nicht-virtuelle Memberfunktionen und statische Bindung (~~`virtual`~~ `double eval()`)
- Überschreiben geerbter Memberfunktionen und Aufrufen der überschriebenen Implementierung
- Mehrfachvererbung (multiple inheritance)
- ...

Objektorientierte Programmierung

Im letzten Kursdrittel wurden einige Konzepte der *objektorientierten Programmierung* vorgestellt, die auf den kommenden Folien noch einmal kurz zusammengefasst werden.

Kapselung (Wochen 10-13):

- Verbergen der Implementierungsdetails von Typen (privater Bereich) vor Benutzern
- Definition einer Schnittstelle (öffentlicher Bereich) zum kontrollierten Zugriff auf Werte und Funktionalität
- Ermöglicht das Sicherstellen von Invarianten, sowie den Austausch von Implementierungen ohne Anpassungen von Benutzercode

Objektorientierte Programmierung

Subtyping (Woche 14):

- Typhierarchien mit Super- und Subtypen können angelegt werden um Verwandtschaftbeziehungen sowie Abstraktionen und Spezialisierungen zu modellieren
- Ein Subtyp unterstützen mindestens die Funktionalität, die auch der Supertyp unterstützt – i.d.R. aber mehr, d.h. Subtypen erweitern die Schnittstelle (den öffentlichen Bereich) ihrer Supertypen
- Daher können Subtypen überall dort eingesetzt werden, wo Supertypen verlangt sind ...
- ... und Funktionen, die auf abstrakteren Typen (Supertypen) operieren können, können auch auf spezialisierteren Typen (Subtypen) operieren
- Die in Woche 7 vorgestellten Streams bilden eine solche Typhierarchie: **ostream** ist der abstrakte Supertyp, **ofstream** etc. sind spezialisierte Subtypen

Objektorientierte Programmierung

Polymorphie und *dynamische Bindung* (Woche 14):

- Ein Zeiger vom statischen Typ T_1 kann zur Laufzeit auf Objekte vom (dynamischen) Typ T_2 zeigen, falls T_2 ein Subtyp von T_1 ist
- Wird eine virtuelle Memberfunktion von einem solchen Zeiger aus aufgerufen, so entscheidet der dynamische Typ darüber, welche Funktion ausgeführt wird
- D.h.: Trotz gleichem statischen Typ kann beim Zugriff auf eine gemeinsame Schnittstelle (Memberfunktionen) eines solchen Zeigers ein anderes Verhalten auftreten
- Zusammen mit Subtyping ermöglicht es dies, neue konkrete Typen (Streams, Ausdrücke, ...) zu einem bestehenden System hinzuzufügen, ohne dieses abändern zu müssen

Objektorientierte Programmierung

Vererbung (Woche 14):

- Abgeleitete Klassen erben die Funktionalität, d.h. die Implementierungen von Memberfunktionen, ihrer Elternklassen
- Dies ermöglicht es, gemeinsam genutzten Code wiederverwenden zu können und vermeidet so Codeduplikation
- Geerbte Implementierungen können auch überschrieben werden, um zu erreichen, dass eine abgeleitete Klasse sich anders verhält als ihre Elternklasse (im Kurs nicht gezeigt)

25. Zusammenfassung

Zweck und Format

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

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

Kapitelüberblick

- 1. Einführung
- 2. Ganze Zahlen
- 3. Wahrheitswerte
- 4. Defensives Programmieren
- 5./6. Kontrollanweisungen
- 7./8. Fließkommazahlen
- 9./10. Funktionen
- 11. Referenztypen
- 12./13. Vektoren und Strings
- 14./15. Rekursion
- 16. Structs und Overloading
- 17. Klassen
- 18./19. Dynamische Datenstrukturen
- 20. Container, Iteratoren und Algorithmen
- 21. Dynamische Datentypen und Speicherverwaltung
- 22. Subtyping, Polymorphie und Vererbung

1. Einführung

- (M) Euklidischer Algorithmus
- (K) Algorithmus, Turingmaschine, Programmiersprachen, Kompilation, Syntax und Semantik
- Werte und Effekte, (Fundamental)typen, Literale, Variablen, Bezeichner, Objekte, Ausdrücke, Operatoren, Anweisungen
- (S) Include-Direktiven `#include <iostream>`
- Hauptfunktion `int main(){...}`
- Kommentare, Layout `// Kommentar`
- Typen, Variablen, L-Wert `a` , R-Wert `a+b`
- Ausdrucksanweisung `b=b*b;` , Deklarationsanweisung `int a;`, Rückgabeeanweisung `return 0;`

2. Ganze Zahlen

- Celsius to Fahrenheit
 - Assoziativität und Präzedenz, Stelligkeit
 - Ausdrucksbäume, Auswertungsreihenfolge
 - Arithmetische Operatoren
 - Binärzahldarstellung, Hexadezimale Zahlen, Wertebereich
 - Zahlendarstellung mit Vorzeichen, Zweierkomplement
- Arithmetische Operatoren `9 * celsius / 5 + 32`
 - Inkrement / Dekrement `expr++`
 - Arithmetische Zuweisungen `expr1 += expr2`
 - Konversion `int` ↔ `unsigned int`
- Celsius to Fahrenheit, Ersatzwiderstand

3. Wahrheitswerte

- Ⓚ
 - Boole'sche Funktionen, Vollständigkeit
 - DeMorgan'sche Regeln
- Ⓢ
 - Der Typ `bool`
 - Logische Operationen `a && !b`
 - Relationale Operationen `x < y`
 - Präzedenzen `7 + x < y && y != 3 * z`
 - Kurzschlussauswertung `x != 0 && z / x > y`
 - Die `assert`-Anweisung, `#include <cassert>`
- ⓑ
 - Div-Mod Identität.

4. Defensives Programmieren

- Ⓚ ■ Assertions und Konstanten
- Ⓢ ■ Die `assert`-Anweisung, `#include <cassert>`
 - `const int speed_of_light=2999792458`
- Ⓟ ■ Assertions für den GGT

5./6. Kontrollanweisungen

- **M** Linearer Kontrollfluss vs. interessante Programme, Spaghetti-Code
- **K** Auswahlanweisungen, Iterationsanweisungen
 - (Vermeidung von) Endlosschleifen, Halteproblem
 - Sichtbarkeits- und Gültigkeitsbereich, Automatische Speicherdauer
 - Äquivalenz von Iterationsanweisungen
- **S** if Anweisungen `if (a % 2 == 0) {..}`
 - for Anweisungen `for (unsigned int i = 1; i <= n; ++i) ...`
 - while und do-Anweisungen `while (n > 1) {...}`
 - Blöcke, Sprunganweisungen `if (a < 0) continue;`
 - Switch Anweisung `switch(grade) {case 6: }`
- **B** Summenberechnung (Gauss), Primzahltest, Collatz-Folge, Fibonacci Zahlen, Taschenrechner, Notenausgabe

7./8. Fließkommazahlen

- ① ■ Richtig Rechnen: Celsius / Fahrenheit
- ② ■ Fixkomma- vs. Fließkommazahldarstellung
■ (Löcher im) Wertebereich
■ Rechnen mit Fließkommazahlen, Umrechnung
■ Fließkommazahlensysteme, Normalisierung, IEEE Standard 754
■ *Richtlinien für das Rechnen mit Fließkommazahlen*
- ③ ■ Typen `float`, `double`
■ Fließkommaliterale `1.23e-7f`
- ④ ■ Celsius/Fahrenheit, Euler, Harmonische Zahlen

9./10. Funktionen

- (M) Potenzberechnung
- (K) Kapselung von Funktionalität
- Funktionen, formale Argumente, Aufrufargumente
- Gültigkeitsbereich, Vorwärts-Deklaration
- Prozedurales Programmieren, Modularisierung, Getrennte Übersetzung
- *Stepwise Refinement*
- (S) Funktionsdeklaration, -definition
`double pow(double b, int e){ ... }`
- Funktionsaufruf `pow (2.0, -2)`
- Der typ `void`
- (B) Potenzberechnung, perfekte Zahlen, Minimum, Kalender

11. Referenztypen

- ① Funktion Swap
- ② Werte-/ Referenzsemantik, Pass by Value / Pass by Reference, Return by Reference
 - Lebensdauer von Objekten / Temporäre Objekte
 - Konstanten
- ③ Referenztyp `int& a`
 - Call by Reference und Return by Reference `int& increment (int& i)`
 - Const-Richtlinie, Const-Referenzen, Referenzrichtlinie
- ④ Swap, Inkrement

12./13. Vektoren und Strings

- Ⓜ
 - Iteration über Daten: Sieb des Eratosthenes
- Ⓚ
 - Vektoren, Speicherlayout, Wahlfreier Zugriff
 - (Fehlende) Grenzenprüfung
 - Vektoren
 - Zeichen: ASCII, UTF8, Texte, Strings
- Ⓢ
 - Vektor Typen `std::vector<int> a {4,3,5,2,1};`
 - Zeichen und Texte, der Typ `char c = 'a';`, Konversion nach `int`
 - Vektoren von Vektoren
 - Ströme `std::istream`, `std::ostream`
- Ⓟ
 - Sieb des Eratosthenes, Caesar-Code, Kürzeste Wege

14./15. Rekursion

- Ⓜ
 - Rekursive math. Funktionen, Das n-Queen Problem, , Lindenmayer-Systeme, Kommandozeilenrechner
- Ⓚ
 - Rekursion
 - Aufrufstapel, Gedächtnis der Rekursion
 - Korrektheit, Terminierung,
 - Rekursion vs. Iteration
 - Backtracking, EBNF, Formale Grammatiken, Parsen
- Ⓟ
 - Fakultät, GGT, Sudoku-Löser, Taschenrechner

16. Structs und Overloading

- ① ■ Datentyp Rationale Zahlen selber bauen
- ② ■ Heterogene Datenstruktur
 - Funktions- und Operator-Overloading
 - Datenkapselung
- ③ ■ Struct Definition `struct rational {int n; int d;};`
 - Mitgliedszugriff `result.n = a.n * b.d + a.d * b.n;`
 - Initialisierung und Zuweisung,
 - Überladen von Funktionen `pow(2)` vs. `pow(3,3)`; , Überladen von Operatoren
- ④ ■ rationale Zahlen, komplexe Zahlen

17. Klassen

- (M) Rationale Zahlen mit Kapselung
- (K) Kapselung, Konstruktion, Mitgliedsfunktionen
- (S) Klassen `class rational { ... };`
 - Zugriffssteuerung `public: / private:`
 - Mitgliedsfunktionen `int rational::denominator () const`
 - Das implizite Argument der Memberfunktionen
- (B) Endlicher Ring, Komplexe Zahlen

18./19. Dynamische Datenstrukturen

- Ⓜ ■ Unser eigener Vektor
- Ⓚ ■ Allokation, Zeiger-Typen, Verkettete Liste, Allokation, Deallokation, Dynamischer Datentyp
- Ⓢ ■ Die **new** Anweisung
 - Zeiger `int* x`;, Nullzeiger `nullptr`.
 - Adress-, Dereferenzoperator `int *ip = &i; int j = *ip;`
 - Zeiger und Const `const int *a;`
- Ⓟ ■ Verkettete Liste, Stack

20. Container, Iteratoren und Algorithmen

- Ⓜ
 - Vektoren sind Container
- Ⓚ
 - Iterieren mit Zeigern
 - Container und Iteratoren
 - Algorithmen
- Ⓢ
 - Iteratoren `std::vector<int>::iterator`
 - Algorithmen der Standardbibliothek `std::fill (a, a+5, 1);`
 - Einen Iterator implementieren
 - Iteratoren und `const`
- Ⓑ
 - Ausgeben eines Vektors, einer Menge

21. Dynamische Datentypen und Speicherverwaltung

- Ⓜ
 - Stack
 - Ausdrucksbaum
- Ⓚ
 - Richtlinie „Dynamischer Speicher“
 - Gemeinsamer Zeiger-Zugriff
 - Dynamischer Datentyp
 - Baumstruktur
- Ⓢ
 - **new** und **delete**
 - Desktruktor `stack::~~stack()`
 - Kopierkonstruktor `stack::stack(const stack& s)`
 - Zuweisungsoperator
`stack& stack::operator=(const stack& s)`
 - Dreierregel
- Ⓟ
 - Binärer Suchbaum

22. Subtyping, Polymorphie und Vererbung

- Ⓜ ■ Erweitern und Verallgemeinern von Ausdrucksbäumen
- Ⓚ ■ Subtyping
 - Polymorphie und dynamische Bindung
 - Vererbung
- Ⓢ ■ Basisklasse `struct Exp{}`
 - Abgeleitete Klasse `struct BinExp: public Exp{}`
 - Abstrakte Klasse
 - `struct Exp{virtual int size() const = 0...}`
 - Polymorphie `virtual double eval()`
- Ⓟ ■ Ausdrucksknoten und Erweiterungen

Ende

Ende der Vorlesung.