

# 1. Einführung

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

# Was ist Informatik?

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

(Wikipedia, nach dem “Duden Informatik”)

# Informatik $\neq$ Computer Science

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

Mike Fellows, US-Informatiker (1991)

# Computer Science $\subseteq$ Informatik

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

# Informatik $\neq$ EDV-Kenntnisse

EDV-Kenntnisse: *Anwenderwissen*

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

Informatik: *Grundlagenwissen*

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

# Inhalt dieser Vorlesung

- Systematisches Problemlösen mit Algorithmen und der Programmiersprache C++.

# 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 überlieferte Algorithmus...

... ist der **Euklidische Algorithmus** (aus Euklids *Elementen*, 3. Jh. v. Chr.)

- Eingabe: zwei natürliche Zahlen  $a > b$
- Ausgabe: grösster gemeinsamer Teiler  $\text{ggT}(a, b)$

Moderner Euklidischer Algorithmus am Beispiel:

$a$	$b$	$a \text{ div } b$	$a \text{ mod } b$
1071	1029	1	42
1029	42	24	21
42	21	2	0 $\Rightarrow \text{ggT} = 21$



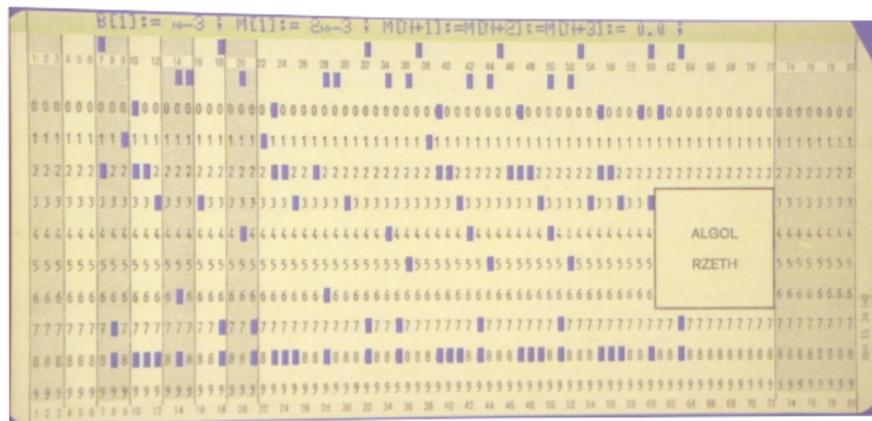
# ETH: Pionierin der modernen Informatik

1956: Inbetriebnahme der ERMETH, entwickelt und gebaut an der ETH von Eduard Stiefel.



# ETH: Pionierin der modernen Informatik

1958–1963: Entwicklung von ALGOL 60 (der ersten formal definierten Programmiersprache), unter anderem durch Heinz Rutishauer, ETH



1964: Erstmals können ETH-Studierende selbst einen Computer programmieren (die CDC 1604, gebaut von Seymour Cray).

# ETH: Pionierin der modernen Informatik



Die Klasse 1964 im Jahr 2015 (mit einigen Gästen)

# ETH: Pionierin der modernen Informatik

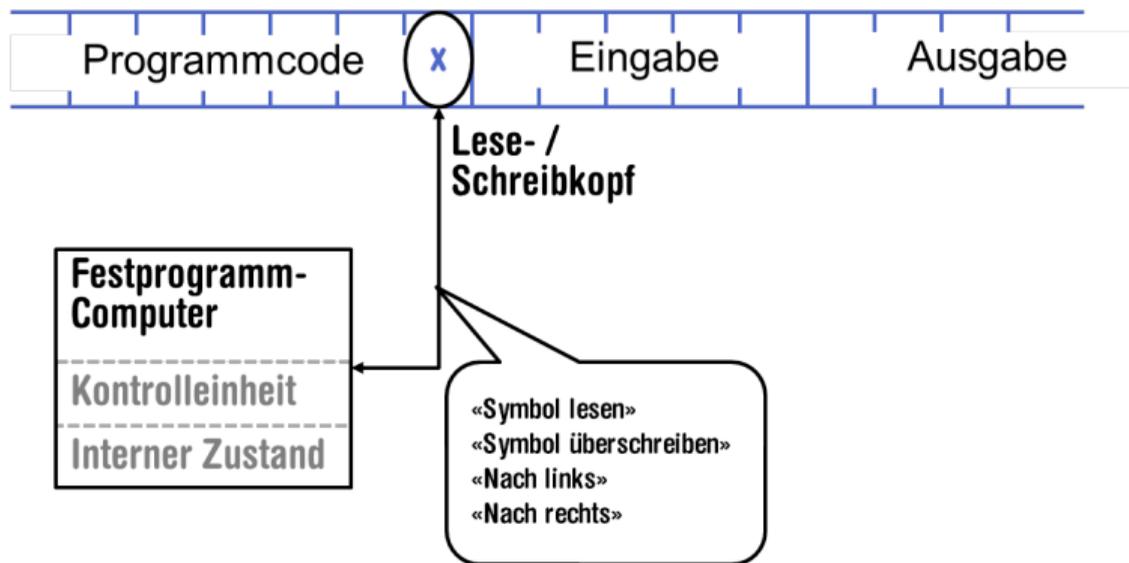
1968–1990: Niklaus Wirth entwickelt an der ETH die Programmiersprachen Pascal, Modula-2 und Oberon und 1980 die *Lilith*, einen der ersten Computer mit grafischer Benutzeroberfläche.



# Computer – Konzept

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

Folge von Symbolen auf Ein- und Ausgabeband

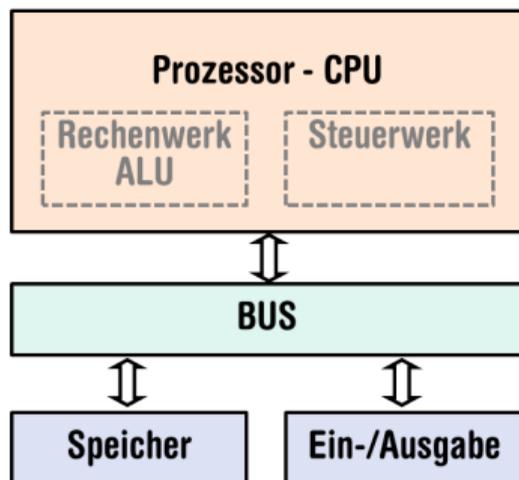


Alan Turing

# Computer – Umsetzung

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

## Von Neumann Architektur



Konrad Zuse



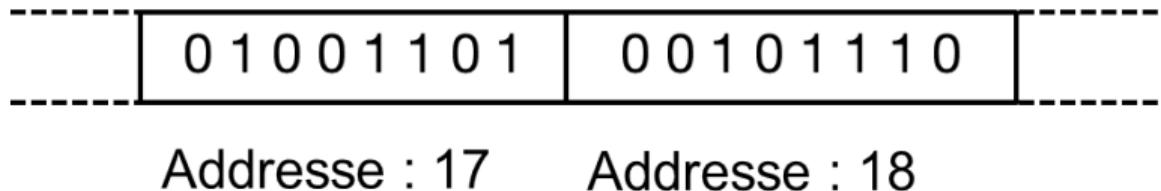
John von Neumann

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.



## Der Prozessor

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

# Rechengeschwindigkeit

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



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

arbeitet ein heutiger Desktop-PC mehr als 100 Millionen Instruktionen ab.<sup>2</sup>

---

<sup>2</sup>Uniprozessor Computer bei 1GHz

# Deklaratives Wissen

Wissen über *Sachverhalte* – formulierbar in Aussagesätzen.

- Es gibt unendlich viele ganze Zahlen.
- Der Computerspeicher ist endlich.
- $x$  ist eine Wurzel von  $y$ , wenn  $y = x^2$ .

# Prozedurales Wissen

Wissen über *Abläufe* – formulierbar in Form von  
Verarbeitungsanweisungen (kurz: Befehle).

Beispiel: *Algorithmus*<sup>3</sup> zur Approximation von  $\sqrt{y}$ :

- 1 Starte mit einem Schätzwert  $s$  von  $\sqrt{y}$ .
- 2 Ist  $s^2$  nahe genug bei  $y$ , dann ist  $x := s$  eine gute Approximation der Wurzel von  $y$ .
- 3 Andernfalls erzeuge eine neue Schätzung durch

$$s_{neu} := \frac{s + y/s}{2}.$$

- 4 Ersetze  $s$  durch  $s_{neu}$  und gehe zurück zu Schritt 2.

---

<sup>3</sup>Newton-Methode

# Programmieren

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



The Harvard Computers, Menschliche Berufsrechner, ca.1890

# Warum Programmieren?

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

# 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)
- Die meisten qualifizierten Jobs benötigen zumindest elementare Programmierkenntnisse.
- Programmieren macht Spass!

# Programmiersprachen

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

*Höhere Programmiersprache: darstellbar als Programmtext, der*

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

# Programmiersprachen – Einordnung

Unterscheidung in

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

# Warum C++?

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

- C++ ist relevant in der Praxis.
- Für das wissenschaftliche Rechnen (wie es in der Mathematik und Physik gebraucht wird), bietet C++ viele nützliche Konzepte.
- C++ ist weit verbreitet und “läuft überall”
- C++ ist standardisiert, d.h. es gibt ein “offizielles” C++.
- Der Dozent mag C++.

# 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 → Typ gebunden, objekt-orientiert

# Deutsch vs. C++

## Deutsch

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

## C++

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

# Syntax und Semantik

- Programme müssen, wie unsere Sprache, nach gewissen Regeln geformt werden.
  - **Syntax**: 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.

# C++: Fehlerarten illustriert an deutscher Sprache

■ Das Auto fuhr zu schnell.

Syntaktisch und semantisch korrekt.

■ DasAuto fuh r zu sxhnell.

Syntaxfehler: Wortbildung.

■ Rot das Auto ist.

Syntaxfehler: Satzstellung.

■ Man empfiehlt dem Dozenten  
nicht zu widersprechen

Syntaxfehler: Satzzeichen fehlen .

■ Sie ist nicht gross und rothaarig.

Syntaktisch korrekt aber mehrdeutig. [kein Analogon]

■ Die Auto ist rot.

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

■ Das Fahrrad galoppiert schnell.

Syntaktisch und grammatikalisch korrekt! Semantisch fehlerhaft. [Laufzeitfehler]

■ Manche Tiere riechen gut.

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

# Syntax und Semantik von C++

## *Syntax*

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

## *Semantik*

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

# Syntax und Semantik von C++

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

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

# Was braucht es zum Programmieren?

- **Editor:** Programm zum Ändern, Erfassen und Speichern von C++-Programmtext
- **Compiler:** Programm zum Übersetzen des Programmtexts in 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
- Objekte
- **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;
    // 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;
}
```

# Verhalten eines Programmes

Zur Compilationszeit:

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

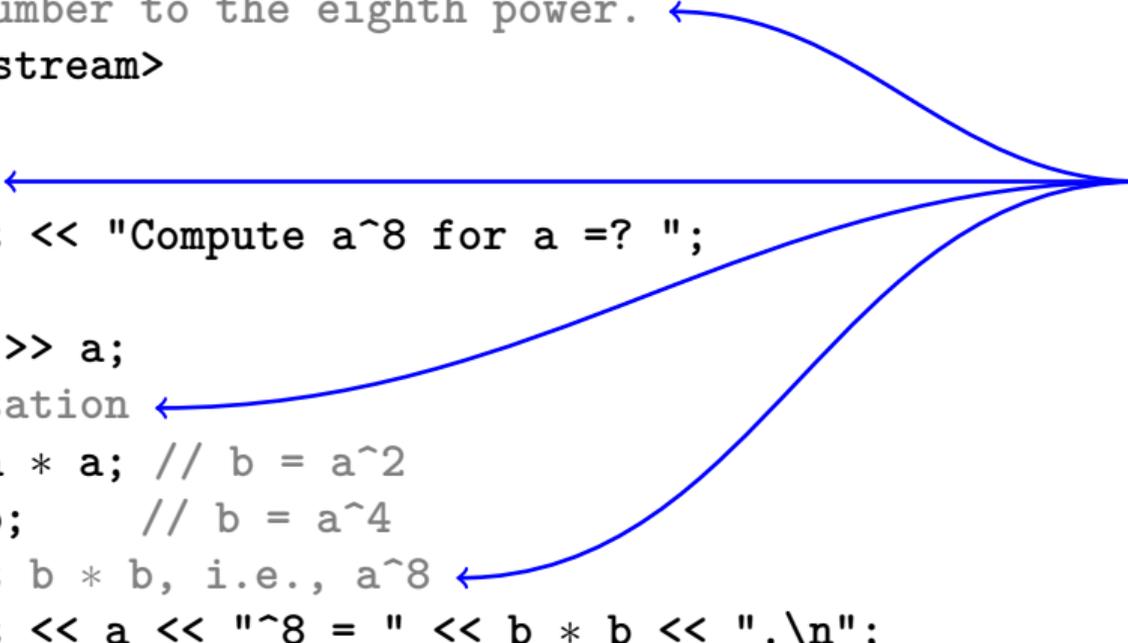
Zur Laufzeit:

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

# 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!**

# Include und Mainfunktion

```
// 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 (Statements)

```
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 (statements)

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

# Ausdrucksanweisungen

- haben die Form

`expr;`

wobei *expr* ein Ausdruck ist

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

Beispiel: `b = b*b;`

# Rückgabeeanweisungen

- treten nur in Funktionen auf und sind von der Form

`return expr;`

wobei *expr* ein Ausdruck ist

- spezifizieren Rückgabewert der Funktion

Beispiel: `return 0;`

# Anweisungen – Werte und 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
    std::cout << a << "^8 = " << b * b << ".\n";
    return 0;
}
```

*Effekt:* Ausgabe des Strings Compute ...

*Effekt:* Eingabe einer Zahl und Speichern in a

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

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

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

*Effekt:* Rückgabe des Wertes 0

# Werte und Effekte

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

# Anweisungen – Variablendefinitionen

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

Typ-  
namen

Deklarationsanweisungen

# Deklarationsanweisungen

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

Beispiel: `int a;`

- können Variablen auch initialisieren

Beispiel: `int b = a * a;`

# Typen und Funktionalität

`int`:

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

In C++ hat jeder Typ einen Namen sowie

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

# Fundamentaltypen

C++ enthält fundamentale Typen für

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

# Literale

- repräsentieren konstante Werte,
- haben festen *Typ* und *Wert*
- sind "syntaktische Werte".

Beispiele:

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

# Variablen

- repräsentieren (wechselnde) Werte,
- haben
  - *Name*
  - *Typ*
  - *Wert*
  - *Adresse*
- sind im Programmtext "sichtbar".

## Beispiel

`int a;` definiert Variable mit

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

# Objekte

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

## Anmerkung

Ein Programm hat eine *feste* Anzahl von Variablen. Um eine variable Anzahl von Werten behandeln zu können, braucht es "anonyme" Adressen, die über temporäre Namen angesprochen werden können.

# Bezeichner und Namen

(Variablen-)Namen sind Bezeichner:

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

Es gibt noch andere Namen:

- `std::cin` (qualifizierter Name)

# Ausdrücke (Expressions)

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

Analogie: Baukasten

# Ausdrücke (Expressions)

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

Zusammengesetzter Ausdruck

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

Zweifach zusammengesetzter Ausdruck

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

return 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

$a * a = (a * a)$

# Ausdrücke (Expressions)

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

## Beispiel

`a * a`

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

## Beispiel

`b = b * b`

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

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

# L-Werte und R-Werte

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

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

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

**R-Wert** (Right Value): Points to the string literal "Compute a^8 for a =? " in the first line.

**L-Wert** (Left Value): Points to the variable `a` in the second line.

**L-Wert** (Left Value): Points to the variable `b` in the third line.

**R-Wert** (Right Value): Points to the expression `b * b` in the fourth line.

**R-Wert** (Right Value): Points to the expression `b * b` in the fifth line.

**R-Wert** (Right Value): Points to the constant `0` in the sixth line.

# L-Werte und R-Werte

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

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

Beispiel: Variablenname

# L-Werte und R-Werte

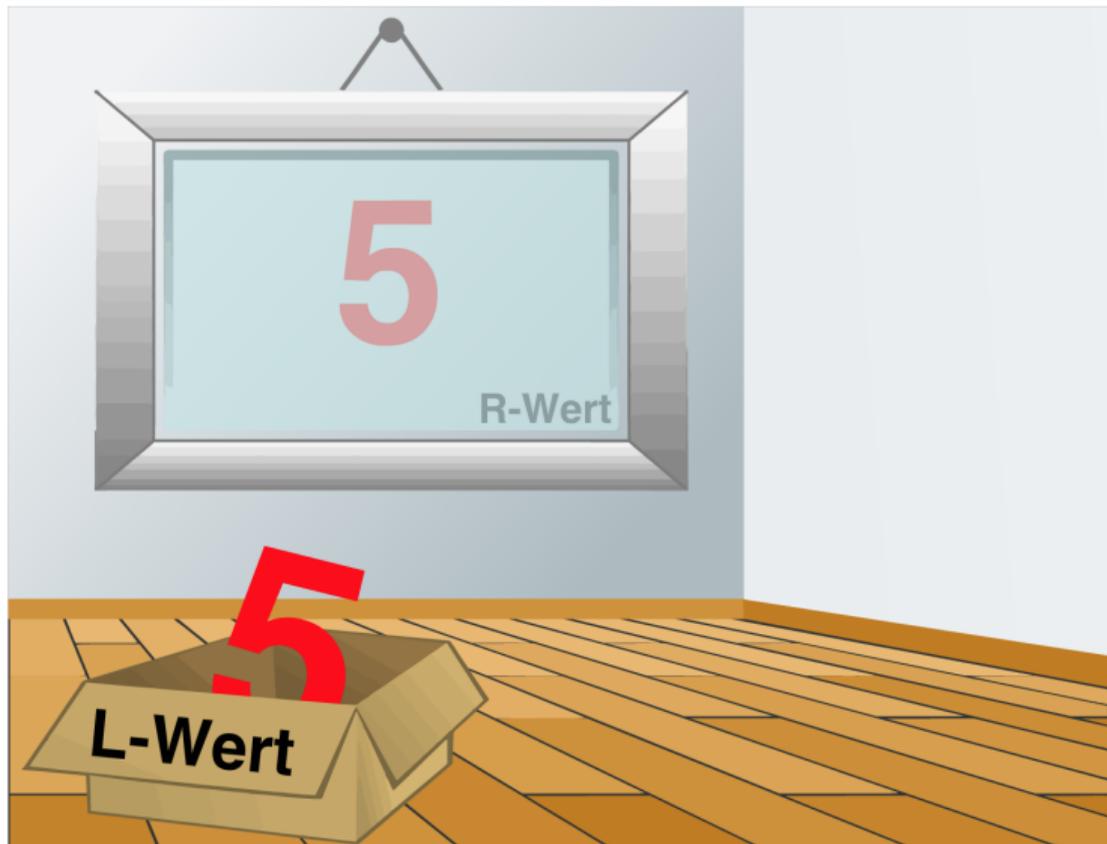
R-Wert (“**R**echts vom Zuweisungsoperator”)

- Ausdruck der kein L-Wert ist

Beispiel: Literal 0

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

# L-Werte und R-Werte



# Operatoren und Operanden

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

// compute
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;
```

Diagram illustrating the operators and operands in the provided C++ code:

- Linker Operand (Ausgabestrom)**: Points to `std::cout` in the first line.
- Ausgabe-Operator**: Points to `<<` in the first line.
- Rechter Operand (String)**: Points to `"Compute a^8 for a =? "` in the first line.
- Rechter Operand (Variablenname)**: Points to `a` in the second line.
- Eingabe-Operator**: Points to `>>` in the second line.
- Linker Operand (Eingabetrom)**: Points to `std::cin` in the second line.
- Zuweisungsoperator**: Points to `=` in the third line.
- Multiplikationsoperator**: Points to `*` in the fourth line.

# Operatoren

## Operatoren

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

# Multiplikationsoperator \*

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

Beispiele:  $a * a$  und  $b * b$

# Zuweisungsoperator =

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

Beispiele:  $b = b * b$  und  $a = b$

## Vorsicht, Falle!

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

# Eingabeoperator >>

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

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

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

# Ausgabeoperator <<

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

Beispiel: `std::cout << a` (meist Bildschirmausgabe)

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

# Ausgabeoperator <<

Warum Rückgabe des Ausgabestroms?

- erlaubt Bündelung von Ausgaben

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

ist wie folgt logisch geklammert

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

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

# power8\_exact.cpp

- Problem mit `power8.cpp`: grosse Eingaben werden nicht korrekt behandelt
- Grund: Wertebereich des Typs `int` ist beschränkt (siehe nächste VL)
- Lösung: verwende einen anderen Typ, z.B. `ifm::integer`

# power8\_exact.cpp

---

```
// Program: power8_exact.cpp
// Raise a number to the eighth power,
// using integers of arbitrary size

#include <iostream>
#include <IFMP/integer.h>

int main()
{
    // input
    std::cout << "Compute a^8 for a =? ";
    ifmp::integer a;
    std::cin >> a;

    // computation
    ifmp::integer 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;
}
```