



# Einführung

---

*Real Programmers vs. Quiche Eaters ;*  
Die Mersenne'sche Vermutung, Editor,  
Compiler, Computer, Betriebssystem,  
Plattform, Das erste C++-Programm

# *Real Programmers vs. Quiche Eaters*

## *The Real Programmer*



- ist intuitiver Programmierer (hat's im Blut und ist generell Autodidakt)
- kann mit "seiner" Programmiersprache schon alles und lehnt andere Programmiersprachen deshalb ab
- kann und braucht keine Theorie

# *Real Programmers vs. Quiche Eaters*

## *The Real Programmer*



- ist intuitiver Programmierer (hat's im Blut und ist generell Autodidakt)
- kann mit "seiner" Programmiersprache schon alles und lehnt andere Programmiersprachen deshalb ab
- kann und braucht keine Theorie

## *The Quiche Eater*



- macht sich Gedanken über "Gutes Programmieren"
- will mit "seiner" Programmiersprache gar nicht alles können, sondern nur das Richtige
- ist meist Akademiker

# *Real Programmers vs. Quiche Eaters*

## *The Real Programmer*



- ist intuitiver Programmierer (hat's im Blut und ist generell Autodidakt)
- kann mit "seiner" Programmiersprache schon alles und lehnt andere Programmiersprachen deshalb ab
- kann und braucht keine Theorie

## *The Quiche Eater*



**Niklaus Wirth** (\*1934)

- Bekanntester CH-Informatiker
- 1968 – 1999 Professor an der ETH
- Entwickler der Programmiersprachen **Pascal** (ab 1968), **Modula-2** (ab 1977) und **Oberon** (ab 1986)

# *Real Programmers vs. Quiche Eaters*

## *The Real Programmer*



- Erfunden 1983 von Ed Post im Informatik-Klassiker *Real Programmers don't use Pascal* (siehe VL-Webseite)
- programmiert in **C** oder **Fortan**

## *The Quiche Eater*

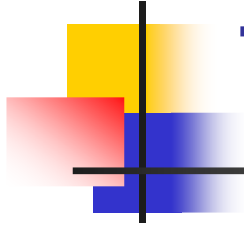


**Niklaus Wirth** (\*1934)

- Bekanntester CH-Informatiker
- 1968 – 1999 Professor an der ETH
- Entwickler der Programmiersprachen **Pascal** (ab 1968), **Modula-2** (ab 1977) und **Oberon** (ab 1986)

*The Real Quiche Eater –*  
Thesen des Dozenten

---





# *The Real Quiche Eater* – Thesen des Dozenten

---

- Jeder Programmierer wird irgendwann einmal erwachsen (und zum *Quiche Eater*), aber die kindliche Seite (der *Real Programmer*) stirbt niemals ganz.



# *The Real Quiche Eater* – Thesen des Dozenten

---

- Jeder Programmierer wird irgendwann einmal erwachsen (und zum *Quiche Eater*), aber die kindliche Seite (der *Real Programmer*) stirbt niemals ganz.
- Wir brauchen eine Programmiersprache, die *beide* Seiten zufriedenstellt (C, Fortran, Pascal, Modula-2, Oberon tun dies nicht).





# *The Real Quiche Eater* – Thesen des Dozenten

---

- Jeder Programmierer wird irgendwann einmal erwachsen (und zum *Quiche Eater*), aber die kindliche Seite (der *Real Programmer*) stirbt niemals ganz.
- Wir brauchen eine Programmiersprache, die *beide* Seiten zufriedenstellt (C, Fortran, Pascal, Modula-2, Oberon tun dies nicht).
- **C++** ist die einzige Lösung!



# Programmieren

---

- Welche Programmiersprache?
  - C++



# Programmieren

---

- Welche Programmiersprache?
  - C++
- Warum überhaupt ???
  - Da hätte ich ja gleich Informatik studieren können...
  - Es gibt doch schon für alles Programme...
  - Programmieren interessiert mich nicht...



# Die Mersenne'sche Vermutung

---

Mersenne (1644):

Die Zahlen der Form  $2^n - 1$  sind Primzahlen für  $n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257$ , aber für kein anderes  $n < 257$ .



# Die Mersenne'sche Vermutung

---

Mersenne (1644):

Die Zahlen der Form  $2^n - 1$  sind Primzahlen für  $n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257$ , aber für kein anderes  $n < 257$ .

- $2^2 - 1 = 3$
- $2^3 - 1 = 7$
- $2^5 - 1 = 31$



# Die Mersenne'sche Vermutung

Mersenne (1644):

Die Zahlen der Form  $2^n - 1$  sind Primzahlen für  $n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257$ , aber für kein anderes  $n < 257$ .

- $2^2 - 1 = 3$   ■  $2^{19} - 1$   (1644)
- $2^3 - 1 = 7$   ■  $2^{31} - 1$   (1772, Euler)
- $2^5 - 1 = 31$

# Die Mersenne'sche Vermutung

Mersenne (1644):

Die Zahlen der Form  $2^n - 1$  sind Primzahlen für  $n = 2, 3, 5, 7, 13, 17, 19, 31, 67, 127, 257$ , aber für kein anderes  $n < 257$ .

- $2^2 - 1 = 3$   ■  $2^{19} - 1$   (1644)
- $2^3 - 1 = 7$   ■  $2^{31} - 1$   (1772, Euler)
- $2^5 - 1 = 31$   ■  $2^{67} - 1$   (1876, Lucas)



# Die Mersenne'sche Vermutung

---

- Lucas' Beweis von 1876 ist nicht konstruktiv, er liefert keine Faktorisierung von

$$2^{67}-1 = 147573952589676412927.$$





# Die Mersenne'sche Vermutung

---

- Lucas' Beweis von 1876 ist nicht konstruktiv, er liefert keine Faktorisierung von

$$2^{67}-1 = 147573952589676412927.$$

- Faktorisierung blieb offen bis 1903



# Der Vortrag von Cole

---

- Frank Nelson Cole: *On the Factorization of large numbers*, Treffen der *American Mathematical Society* 1903



# Der Vortrag von Cole

---

- Frank Nelson Cole: *On the Factorization of large numbers*, Treffen der *American Mathematical Society* 1903
- Vielleicht der erste und einzige Vortrag, der ohne ein einziges Wort auskam



# Der Vortrag von Cole

---

761838257287 x 193707721

-----

761838257287

6856544315583

2285514771861

5332867801009

5332867801009

5332867801009

1523676514574

761838257287

-----

147573952589676412927



# Der Vortrag von Cole

---

761838257287 x 193707721

-----

761838257287

6856544315583

2285514771861

5332867801009

5332867801009

5332867801009

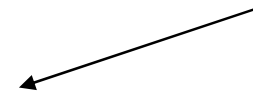
1523676514574

761838257287

-----

147573952589676412927

$2^{67}-1$





# Der Vortrag von Cole

---

- Frank Nelson Cole: *On the Factorization of large numbers*, Treffen der *American Mathematical Society* 1903
- Vielleicht der erste und einzige Vortrag, der ohne ein einziges Wort auskam
- Ergebnis: *Standing ovations* für Cole



# Der Vortrag von Cole

---

- Frank Nelson Cole: *On the Factorization of large numbers*, Treffen der *American Mathematical Society* 1903
- Vielleicht der erste und einzige Vortrag, der ohne ein einziges Wort auskam
- Ergebnis: *Standing ovations* für Cole
- und für seine drei Jahre Sonntagsarbeit!



Was lernen wir daraus?

---





# Was lernen wir daraus?

---

- Wir brauchen ***Werkzeuge*** (damals: Papier, Bleistift, Kopfrechnen; heute auch Computer)



# Was lernen wir daraus?

---

- Wir brauchen ***Werkzeuge*** (damals: Papier, Bleistift, Kopfrechnen; heute auch Computer)
- Wir brauchen ***Problemlösungskompetenz*** (damals wie heute: Theorie hinter dem Problem kennen; wie setzt man die Werkzeuge effektiv ein?)



# Was lernen wir daraus?

---

- Wir brauchen ***Programmierfähigkeiten***, um das neue Werkzeug Computer (das Cole noch nicht kannte) effektiv einsetzen zu können.
  - Anwendungsprogramme lösen heute viele Routine-Aufgaben
  - Für alles, was darüber hinausgeht, muss man den Computer selbst programmieren!

# Die Mersenne'sche Vermutung heute



---

- $n = 67$  und  $n = 257$ :  $2^n - 1$  keine Primzahl

# Die Mersenne'sche Vermutung heute



---

- $n = 67$  und  $n = 257$ :  $2^n - 1$  keine Primzahl
- Mersenne hat andererseits  $n = 61, 89, 107$  "vergessen"

# Die Mersenne'sche Vermutung heute



---

- $n = 67$  und  $n = 257$ :  $2^n - 1$  keine Primzahl
- Mersenne hat andererseits  $n = 61, 89, 107$  "vergessen"
- grösste bisher bekannte Primzahl der Form  $2^n - 1$  ist  $2^{43112609} - 1$ , gefunden mit massivem Computereinsatz und Spezialsoftware



# Programmieren

---

- Welche Programmiersprache?
  - C++
- Warum überhaupt ???
  - Um den Computer als Werkzeug effektiv einsetzen zu können
- Wie programmiert man eigentlich?



# Editor

---

Programm zum

- Ändern
- Erfassen
- Speichern

von (Programm)-Text





# Editor

---

Programm zum

- Ändern
- Erfassen
- Speichern

von (Programm)-Text

Beispiele:  
Microsoft Word,  
Emacs



# Compiler

---

## Motivation:

- ❑ Sprache, die der Computer versteht, ist sehr primitiv (Maschinensprache)
- ❑ Selbst einfache Operationen müssen in viele Einzelschritte aufgeteilt werden
- ❑ Verstandene Sprache variiert von Computer zu Computer



# Compiler

---

Programm zur Übersetzung von

- visuell lesbarem
  - computermodell-unabhängigem
- Programmtext in Maschinensprache.

Idee der **höheren Programmiersprache**



# Compiler

---

Programm zur Übersetzung von

- visuell lesbarem
  - computermodell-unabhängigem
- Programmtext in Maschinensprache.

Idee der **höheren Programmiersprache**

Beispiele: Pascal, Oberon, C++, Java



# Computer

---

Zutaten der *von-Neumann-Architektur*:

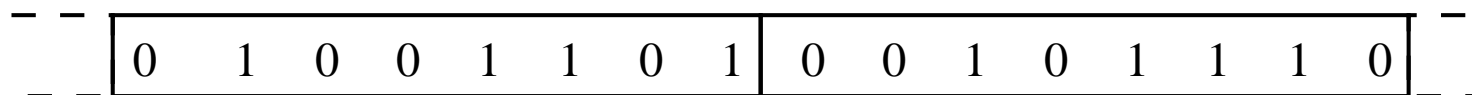
- Hauptspeicher (RAM) für Programme *und* Daten
- Prozessor (CPU) zur Verarbeitung der Programme und Daten



# Hauptspeicher

---

- Folge von *Bits* aus  $\{0,1\}$
- Programmzustand: Werte aller Bits
- Zusammenfassung von Bits zu *Speicherzellen*
- Jede Speicherzelle hat eine *Adresse*
- *Random Access*: Zugriffszeit auf Speicherzelle unabhängig von ihrer Adresse



Adresse: 17

Adresse: 18



# Prozessor

---

- führt Programminstruktionen 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)



# Betriebssystem

---

Programm fuer grundlegende Abläufe:

- Editor starten
- Programm erfassen und speichern
- Compiler aufrufen
- Übersetztes Programm starten





# Betriebssystem

---

Programm fuer grundlegende Abläufe:

- Editor starten
- Programm erfassen und speichern
- Compiler aufrufen
- Übersetztes Programm starten

Beispiele:

Windows, Unix

Linux, MacOS



# Plattform

---

Ideale Welt:

- Programm, das in einer höheren Programmiersprache geschrieben wurde, verhält sich überall gleich



# Plattform

---

Ideale Welt:

- Programm, das in einer höheren Programmiersprache geschrieben wurde, verhält sich überall gleich

Reale Welt (gerade bei C++):

- Verhalten *kann* von Compiler, Computer, Betriebssystem abhängen



# Plattform

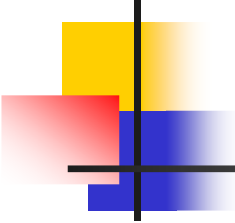
---

Plattform:

- Compiler, Computer, Betriebssystem

Ziel für uns:

- Plattformunabhängige Programme
- Voraussetzung dafür: Verzicht auf maschinennahe Features von C++



# 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;
}
```



# Syntax und Semantik

---

## Syntax:

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

## Semantik:

- Was *bedeutet* ein C++ Programm?
- Welchen Algorithmus realisiert es?



# Syntax und Semantik

---

Der ISO/IEC Standard 14822 (1998)

- ist das "Gesetz" von C++
- legt Grammatik und Bedeutung von C++ Programmen fest
- wird weiterentwickelt: Neuauflage 2009



# Beispiel: `power8.cpp`

---

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





# Kommentare

---

- hat jedes *gute* Programm
- // ab Doppel-Slash bis Zeilenende
- dokumentieren, *was* das Programm *wie* macht
- werden vom Compiler ignoriert



# Layoutelemente

---

- Kommentare
- Leerzeilen, Leerschläge
- Einrückungen, die die Programmlogik widerspiegeln
- werden vom Compiler ignoriert



# Kommentare und Layout

---

Dem Compiler ist's egal...

```
#include<iostream>
int main(){std::cout<<"Compute a^8 for a =? ";
int a;std::cin>>a;int b=a*a;b=b*b;std::cout<<
a<<"^8 = "<<b*b<<".\n";return 0;}
```



# Kommentare und Layout

---

Dem Compiler ist's egal...

```
#include<iostream>
int main(){std::cout<<"Compute a^8 for a =? ";
int a;std::cin>>a;int b=a*a;b=b*b;std::cout<<
a<<"^8 = "<<b*b<<".\n";return 0;}
```

...aber uns nicht!



# Include-Direktiven

---

C++ besteht aus

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

```
#include <iostream>
```

- macht Ein/Ausgabe verfügbar



# Die `main`-Funktion

---

- hat jedes C++ Programm
- wird vom Betriebssystem aufgerufen
- wie mathematische Funktion...
  - Argumente (bei `power8.cpp`: keine)
  - Rückgabewert (bei `power8.cpp`: 0)



# Die `main`-Funktion

---

- hat jedes C++ Programm
- wird vom Betriebssystem aufgerufen
- wie mathematische Funktion...
  - Argumente (bei `power8.cpp`: keine)
  - Rückgabewert (bei `power8.cpp`: 0)
- ...aber mit zusätzlichem **Effekt!**
  - Lies Zahl ein und gib 8-te Potenz aus



# Werte und Effekte

---

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





# Typen und Funktionalität

---

**int :**

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



# 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)



# 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)



# Typen und Funktionalität

---

C++ enthält *fundamentale* Typen für

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



# Literale

---

- repräsentieren konstante Werte
- haben einen festen Typ
- sind "syntaktische Werte"



# Literale

---

- repräsentieren konstante Werte
- haben einen festen Typ
- sind "syntaktische Werte"

Beispiele:

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

# Variablen

- repräsentieren wechselnde Werte
- haben
  - Namen
  - Typ
  - Wert
  - Adresse
- sind im Programmtext "sichtbar"





# Variablen

---

- repräsentieren (wechselnde) Werte

- haben

- Namen
- Typ
- Wert
- Adresse

```
int a;           definiert Variable mit
```

- Namen: `a`
- Typ: `int`
- Wert: `undefiniert`
- Adresse: `durch Compiler bestimmt`

- sind im Programmtext "sichtbar"





# Konstanten

---

- sind Variablen mit unveränderbarem Wert

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



# Konstanten

---

- sind Variablen mit unveränderbarem Wert

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

- Verwendung: **const** vor der Definition
- Compiler kontrolliert Einhaltung des **const**-Versprechens

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



# Konstanten

---

- sind Variablen mit unveränderbarem Wert

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

- Verwendung: **const** vor der Definition
- Compiler kontrolliert Einhaltung des **const**-Versprechens

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

```
...
```

```
speed_of_light = 300000000; // Rundung
```

Fehler-  
meldung!





# Die `const`-Richtlinie

---

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



# Die `const`-Richtlinie

---

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

- Ein Programm, das diese Richtlinie befolgt, heisst `const`-korrekt



# 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)



# Objekte

---

- repräsentieren Werte im Hauptspeicher
- haben
  - Typ
  - Adresse
  - Wert (Speicherinhalt an der Adresse)



# Objekte

---

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





# Objekte

---

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

Ein Programm hat eine *feste* Anzahl von Variablen.

Um eine grössere Anzahl von Werten behandeln zu können, braucht es "anonyme" Adressen, die über temporäre Namen angesprochen werden können.



# Ausdrücke (Expressions)

---

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



# Ausdrücke (Expressions)

---

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

a \* a



zusammengesetzter Ausdruck:



# Ausdrücke (Expressions)

---

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

a \* a

zusammengesetzter Ausdruck:

Variablenname



# Ausdrücke (Expressions)

---

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

a \* a

zusammengesetzter Ausdruck:

Variablenname, **Operatorsymbol**



# Ausdrücke (Expressions)

---

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

a \* a

zusammengesetzter Ausdruck:

Variablenname, Operatorsymbol,

Variablenname



# Ausdrücke (Expressions)

---

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

a \* a

zusammengesetzter Ausdruck:

Primäre  
Ausdrücke

Variablenname, Operatorsymbol,  
Variablenname

# Ausdrücke (Expressions)

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

Klammern erlaubt:

$a * a$  "="  $(a * a)$

zusammengesetzter Ausdruck:

Primäre  
Ausdrücke

Variablenname, Operatorsymbol,  
Variablenname





# Ausdrücke (Expressions)

---

- haben
  - Typ
  - Wert
  - Effekt (potentiell)



# Ausdrücke (Expressions)

---

- haben
  - Typ
  - Wert
  - Effekt (potentiell)

`a * a`

□ Typ: `int` (Typ der Operanden)

□ Wert: Produkt von `a` und `a`

□ Effekt: keiner



# Ausdrücke (Expressions)

---

- haben
  - Typ
  - Wert
  - Effekt (potentiell)

`b = b * b`

□ Typ: `int` (Typ der Operanden)

□ Wert: Produkt von `b` und `b`

□ Effekt: Weise `b` diesen Wert zu



# Ausdrücke (Expressions)

---

- haben
  - Typ
  - Wert
  - Effekt (potentiell)
- Typ eines Ausdrucks ist fest, aber Wert und Effekt werden durch *Auswertung* des Ausdrucks bestimmt



# L-Werte und R-Werte

---

L-Wert:

- Ausdruck mit Adresse
- Wert ist der Wert des Objekts an dieser Adresse
- gibt Objekt einen (temporären) Namen



# L-Werte und R-Werte

---

L-Wert:

- Ausdruck mit Adresse
- Wert ist der Wert des Objekts an dieser Adresse
- gibt Objekt einen (temporären) Namen

Beispiel: Variablenname



# L-Werte und R-Werte

---

L-Wert:

- Ausdruck mit Adresse
- Wert ist der Wert des Objekts an dieser Adresse
- gibt Objekt einen (temporären) Namen

Beispiel: Variablenname

Andere Beispiele: etwas später...



# L-Werte und R-Werte

---

R-Wert:

- Ausdruck, der kein L-Wert ist





# L-Werte und R-Werte

---

R-Wert:

- Ausdruck, der kein L-Wert ist

Beispiel: Literal



# L-Werte und R-Werte

---

R-Wert:

- Ausdruck, der kein L-Wert ist

Beispiel: Literal

- Jeder L-Wert kann als R-Wert benutzt werden, aber nicht umgekehrt!
- R-Wert kann seinen Wert *nicht* ändern



# Operatoren

---

\* : Multiplikationsoperator

## 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* (hier immer 2)



# Multiplikationsoperator \*

---

- erwartet zwei R-Werte vom gleichen arithmetischem Typ als Operanden
- “gibt Produkt als R-Wert des gleichen Typs zurück”:
  - Der zusammengesetzte Ausdruck ist ein R-Wert; sein Wert ist das Produkt der Werte der beiden Operanden



# Multiplikationsoperator \*

---

- erwartet zwei R-Werte vom gleichen arithmetischem Typ als Operanden
- “gibt Produkt als R-Wert des gleichen Typs zurück”:
  - Der zusammengesetzte Ausdruck ist ein R-Wert; sein Wert ist das Produkt der Werte der beiden Operanden

Beispiele:  $a * a$ ,  $b * b$



# 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



# 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

Beispiel:  $b = b * b$

**:= in Mathe**

# 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

Beispiel:  $b = b * b$





# 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

Beispiel:  $b = b * b$



## 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



## 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)



## 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
- **Eingabestrom muss 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



# 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)



# 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
- **Ausgabestrom muss 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")
```



L-Wert (Ausgabestrom)





# 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")
```



L-Wert, der *kein* Variablenname ist!



# Ausgabeoperator <<

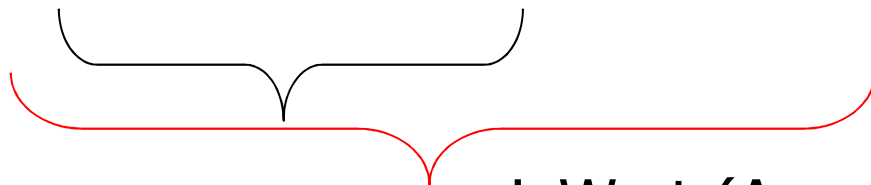
Warum Rückgabe des Ausgabestroms?

- erlaubt Bündelung von Ausgaben:

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

ist wie folgt logisch geklammert:

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



L-Wert (Ausgabestrom)



# Ausgabeoperator <<

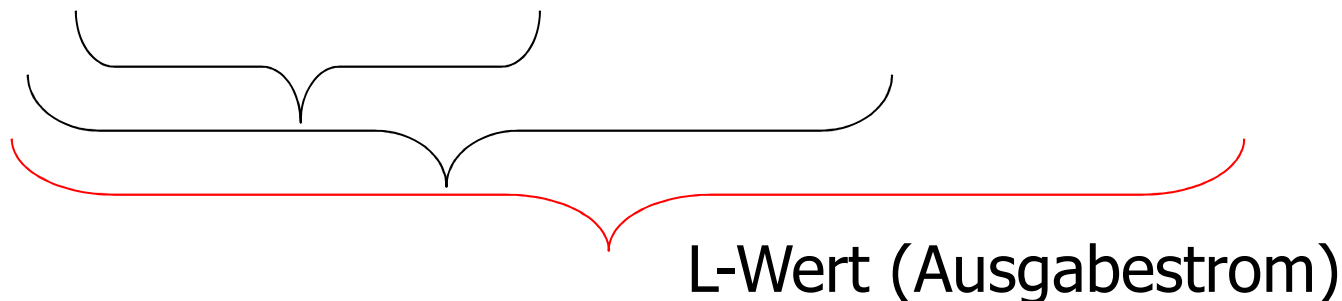
Warum Rückgabe des Ausgabestroms?

- erlaubt Bündelung von Ausgaben:

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

ist wie folgt logisch geklammert:

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





# Ausgabeoperator <<

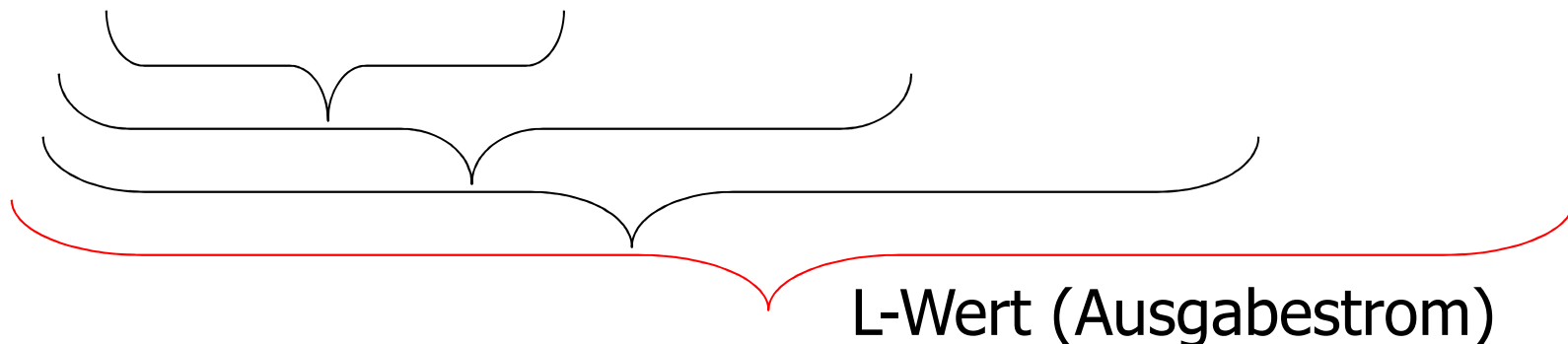
Warum Rückgabe des Ausgabestroms?

- erlaubt Bündelung von Ausgaben:

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

ist wie folgt logisch geklammert:

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





# Anweisungen

---

- Bausteine eines C++ Programms
- werden (von oben nach unten) *ausgeführt* und haben *Effekte*
- enden mit einem Semikolon



# Ausdrucksanweisungen

---

- haben die Form

*expr* ;

wobei *expr* ein Ausdruck ist

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



# 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;`



# Deklarationsanweisungen

---

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





# Deklarationsanweisungen

---

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

Beispiel: `int a;`



# Deklarationsanweisungen

---

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

Beispiel: `int a;`

- können Variablen auch initialisieren



# Deklarationsanweisungen

---

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

Beispiel: `int a;`

- können Variablen auch initialisieren

Beispiel: `int b = a * a;`



# Rückgabeanweisungen

---

- treten nur in Funktionen auf und sind von der Form

`return expr ;`

wobei *expr* ein Ausdruck ist

- spezifizieren Rückgabewert der Funktion



# Rückgabeanweisungen

---

- treten nur in Funktionen auf und sind von der Form

`return expr ;`

wobei *expr* ein Ausdruck ist

- spezifizieren Rückgabewert der Funktion

Beispiel: `return 0 ;`



## power8\_exact.cpp

---

- Problem mit `power8.cpp`: grosse Eingaben werden nicht korrekt behandelt



## power8\_exact.cpp

---

- Problem mit `power8.cpp`: grosse Eingaben werden nicht korrekt behandelt
- Grund: Wertebereich des Typs `int` ist beschränkt (siehe nächste VL)



## `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` (siehe Programm `power8_exact.cpp`)