1. Einführung

Mersenne'sche Vermutung, Höhere Programmiersprachen, Editor, Compiler, Computer, Betriebssystem, das erste C++ Programm und seine syntaktischen und semantischen Bestandteile

Inhalt der Vorlesung

- Systematisches Problemlösen mit dem Computer und der Programmiersprache $\mathrm{C}{+}{+}$.
- Also: *nicht nur*
  aber auch Programmierkurs.

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 weiteres $n < 257$.

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

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 weiteres $n < 257$.

$2^2 - 1 = 3$ ✓ $2^{19} - 1$ ✓ (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 weiteres $n < 257$.

$2^2 - 1 = 3$ ✓	$2^{19} - 1$ ✓ (1644)
$2^3 - 1 = 7$ ✓	$2^{31} - 1$ ✓ (1772, Euler)
$2^5 - 1 = 31$ ✓	$2^{67} - 1$ $\otimes$ (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 je ohne ein einziges Wort auskam.

Der Vortrag von Cole

- Frank Nelson Cole: *On the Factorization of large numbers* , Treffen der American Mathematical Society 1903
- $$\begin{array}{r}
761838257287 \times 193707721 \\
\hline
761838257287 \\
6856544315583 \\
2285514771861 \\
5332867801009 \\
5332867801009 \\
5332867801009 \\
1523676514574 \\
761838257287 \\
\hline
147573952589676412927
\end{array}$$

Der Vortrag von Cole

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

- $$\begin{array}{r} 761838257287 \times 193707721 \\ \hline 761838257287 \\ 6856544315583 \\ 2285514771861 \\ 5332867801009 \\ 5332867801009 \\ 5332867801009 \\ 1523676514574 \\ 761838257287 \\ \hline 147573952589676412927 = 2^{67} - 1 \end{array}$$

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 je ohne ein einziges Wort auskam.
- Ergebnis: Standing ovations für Cole
- und für seine drei Jahre Sonntagsarbeit!

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 (und das ist eine ganze Menge!), muss man den Computer selbst programmieren!

Die Mersenne'sche Vermutung heute

- Für $n = 67$ und $n = 257$ ist $2^n - 1$ keine Primzahl.

Die Mersenne'sche Vermutung heute

- Für $n = 67$ und $n = 257$ ist $2^n - 1$ keine Primzahl.
- Mersenne hat andererseits die Exponenten $n = 61, 89, 107$ „vergessen".

Die Mersenne’sche Vermutung heute

- Für $n = 67$ und $n = 257$ ist $2^n - 1$ keine Primzahl.
- Mersenne hat andererseits die Exponenten $n = 61, 89, 107$ „vergessen”.
- Die grösste bisher bekannte Primzahl der Form $2^n - 1$ ist $2^{57,885,161} - 1$, gefunden im Januar 2013 mit massivem Computereinsatz und Spezialsoftware

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*[1] 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.

[1] 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

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

Warum Programmieren?

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

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.

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!

Höhere 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

$\rightarrow$ Abstraktion!

Höhere Programmiersprachen

Pionier an der ETH: Niklaus Wirth (Prof. an der ETH von 1986 bis 1999)

- Entwickler von Pascal, Modula-2, Oberon
- Noch immer sehr aktiv, u.a. im Bereich Processor Design auf FPGAs[2].

Niklaus Wirth

[2] Field Programmable Gate Arrays

Warum $\mathrm{C}{+}{+}$?

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

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++.

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++.

Was braucht es zum Programmieren?

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

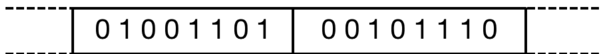
Computer

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

Hauptspeicher

- 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

- 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)

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 $\mathrm{C}{+}{+}$ Programm?
- Ist es *grammatikalisch* korrekt?

Semantik

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

Syntax und Semantik

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. . .

Syntax und Semantik

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.

Verhalten eines Programmes

Zur Compilationszeit:

- vom Compiler akzeptiertes Programm (syntaktisch korrektes $\mathrm{C}{+}{+}$)
- Compiler-Fehler

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)

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

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;}
```

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-Direktiven

C++ besteht aus

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

```
#include <iostream>
```

- macht Ein/Ausgabe verfügbar

Die Hauptfunktion

Die `main`-Funtion

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

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

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".

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

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;
```

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;
```

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.

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* 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.

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

L-Wert

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

Beispiel: Variablenname,
weitere Beispiele etwas später....

L-Werte und R-Werte

R-Wert

- Ausdruck der kein L-Wert ist

  Beispiel: Literal

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

L-Werte und R-Werte

R-Wert

- Ausdruck der kein L-Wert ist

  Beispiel: Literal

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

Jedes e-Bike kann als normales Fahrrad benutzt werden, aber nicht umgekehrt.

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`

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)

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)

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")
```

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.

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;`

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

Beispiel: **`return 0;`**

`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 <IFM/integer.h>

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

  // computation
  ifm::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;
}
```