1. Introduction

Computer Science: Definition and History, Algorithms, Turing Machine, Higher Level Programming Languages, Tools, The first C++Program and its Syntactic and Semantic Ingredients

What is Computer Science?

- The science of **systematic processing of informations**,...
- ...particularly the automatic processing using digital computers.

(Wikipedia, according to "Duden Informatik")

Informatics $\neq$ Science of Computers

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

Mike Fellows, US Computer Scientist (1991)

http://larc.unt.edu/ian/research/cseducation/fellows1991.pdf

Computer Science $\subseteq$ Informatics

- Computer science is also concerned with the development of fast computers and networks...
- ...but not as an end in itself but for the systematic processing **of informations**.

Computer Science $\neq$ Computer Literacy

Computer literacy: *user knowledge*

- Handling a computer
- Working with computer programs for text processing, email, presentations . . .

Computer Science *Fundamental knowledge*

- How does a computer work?
- How do you write a computer program?

This course

- Systematic problem solving with algorithms and the programming language $\mathrm{C{+}{+}}$.
- Hence:
  not only
  but also programming course.

Algorithm: Fundamental Notion of Computer Science

Algorithm:

- Instructions to solve a problem step by step
- Execution does not require any intelligence, but precision (even computers can do it)
- according to *Muhammed al-Chwarizmi*, author of an arabic computation textbook (about 825)

“Dixit algorizmi. . . ” (Latin translation)

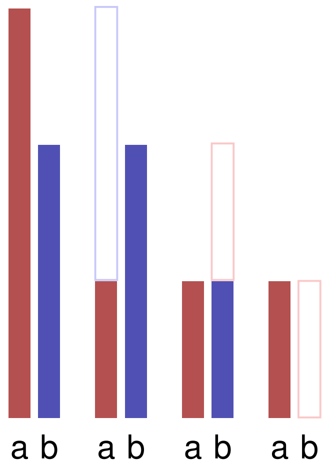
http://de.wikipedia.org/wiki/Algorithmus

Oldest Nontrivial Algorithm

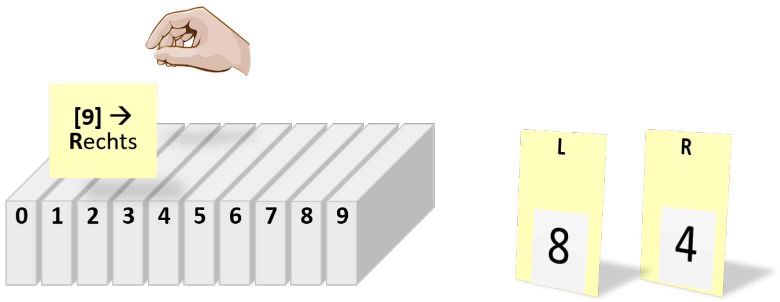
Euclidean algorithm (from the *elements* from Euklid, 3. century B.C.)

- Input: integers $a > 0, b > 0$
- Output: gcd of a und b

While $b \neq 0$
    If $a > b$ then
        $a \leftarrow a - b$
    else:
        $b \leftarrow b - a$
Result: a.



Live Demo: Turing Machine



Speicher

Register

Euklid in the Box

Speicher

0	1	2	3	4	5	6	7	8	9
[8] → **L**	[9] → **R**	**L** $= 0$? *stop*	**R** $>$ **L**? *springe zu* **6**	**L** $-$ **R** → [8]	*springe zu* 0	**R** $-$ **L** → [9]	*springe zu* 0	b	a
Programmcode								Daten	

Daten

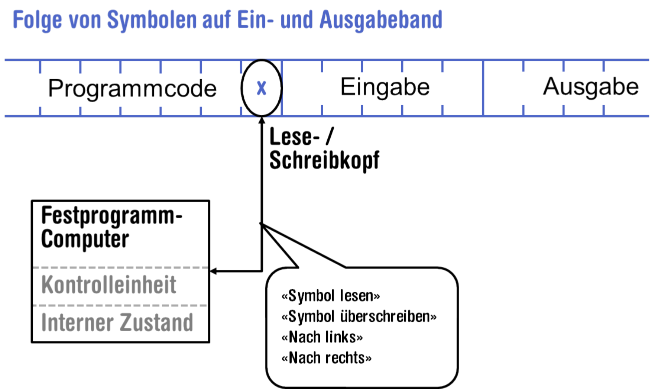
Links	**R**echts
b	a

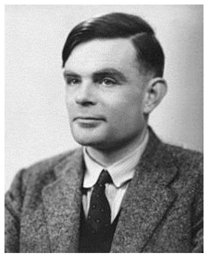
Register

While $b \neq 0$
  If $a > b$ then
    $a \leftarrow a - b$
  else:
    $b \leftarrow b - a$
Ergebnis: a.

Computers – Concept

A bright idea: universal Turing machine (Alan Turing, 1936)





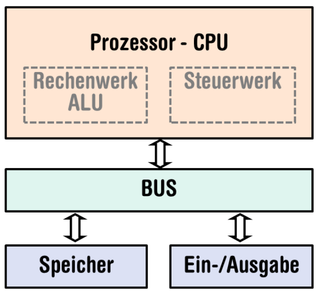
Alan Turing

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

Computer – Implementation

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

Von Neumann Architektur



Konrad Zuse

John von Neumann

http://www.hs.uni-hamburg.de/DE/GNT/hh/biogr/zuse.htm
http://commons.wikimedia.org/wiki/File:John_von_Neumann.jpg

Computer

Ingredients of a *Von Neumann Architecture*

- Memory (RAM) for programs *and* data
- Processor (CPU) to process programs and data
- I/O components to communicate with the world

Memory for data *and* program

- Sequence of bits from $\{0, 1\}$.
- Program state: value of all bits.
- Aggregation of bits to memory cells (often: 8 Bits = 1 Byte)
- Every memory cell has an address.
- Random access: access time to the memory cell is (nearly) independent of its address.

0 1 0 0 1 1 0 1	0 0 1 0 1 1 1 0
Addresse : 17	Addresse : 18

Processor

The processor (CPU)

- executes instructions in machine language
- has an own "fast" memory (registers)
- can read from and write to main memory
- features a set of simplest operations = instructions (e.g. adding to register values)

Computing speed

In the time, onaverage, that the sound takes to travel from from my mouth to you ...

30 m $\widehat{=}$ more than $100.000.000$ instructions

a contemporary desktop PC can process more than 100 millions instructions [2]

[2] Uniprocessor computer at 1 GHz.

Programming

- With a *programming language* we issue commands to a computer such that it does exactly what we want.
- The sequence of instructions is the *(computer) program*

The Harvard Computers, human computers, ca.1890

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

Why programming?

- Do I study computer science or what ...
- There are programs for everything ...
- I am not interested in programming ...
- because computer science is a mandatory subject here, unfortunately...
- . . .

Mathematics used to be the lingua franca of the natural sciences on all universities. Today this is computer science.

Lino Guzzella, president of ETH Zurich, NZZ Online, 1.9.2017

This is why programming!

- Any understanding of modern technology requires knowledge about the fundamental operating principles of a computer.
- Programming (with the computer as a tool) is evolving a cultural technique like reading and writing (using the tools paper and pencil)
- Programming is *the* interface between engineering and computer science – the interdisciplinary area is growing constantly.
- Programming is fun!

Programming Languages

- The language that the computer can understand (machine language) is very primitive.
- Simple operations have to be subdivided into many single steps
- The machine language varies between computers.

Higher Programming Languages

can be represented as program text that

- can be *understood* by humans
- is *independent* of the computer model
  → Abstraction!

Programming langauges – classification

Differentiation into

- Compiled vs. interpreted languages
  - *$C++$*, C#, Pascal, Modula, Oberon, Java
    vs.
    Python, Tcl, Matlab
- *Higher* programming languages vs. Assembler
- *Multi-purpose* programming languages vs. single purpose programming languages
- *Procedural, object oriented*, functional and logical languages.

Why $C++$?

Other popular programming languages: Java, C#, Objective-C, Modula, Oberon, Python . . .

General consensus:

- „The” programming language for systems programming: C
- C has a fundamental weakness: missing (type) safety

Why C++?

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

B. Stroustrup, Design and Evolution of C++, Kap. 4.5

Why C++?

- C++equips C with the power of the abstraction of a higher programming language
- In this course: C++ introduced as high level language, not as better C
- Approach: traditionally procedural $\rightarrow$ object-oriented.

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 and Semantics

- Like our language, programs have to be formed according to certain rules.
  - Syntax: Connection rules for elementary symbols (characters)
  - Semantics: interpretation rules for connected symbols.
- Corresponding rules for a computer program are simpler but also more strict because computers are relatively stupid.

$\mathrm{C}{++}$: Kinds of errors illustrated with German sentences

Das Auto fuhr zu schnell.	Syntaktisch und semantisch korrekt.
DasAuto fuh r zu sxhnell.	Syntaxfehler: Wortbildung.
Rot das Auto ist.	Syntaxfehler: Satzstellung.
Man empfiehlt dem Dozenten nicht zu widersprechen	Syntaxfehler: Satzzeichen fehlen .
Sie ist nicht gross und rothaarig.	Syntaktisch korrekt aber mehrdeutig. [kein Analogon]
Die Auto ist rot.	Syntaktisch korrekt, doch semantisch fehlerhaft: Falscher Artikel. [Typfehler]
Das Fahrrad gallopiert schnell.	Syntaktisch und grammatikalisch korrekt! Semantisch fehlerhaft. [Laufzeitfehler]
Manche Tiere riechen gut.	Syntaktisch und semantisch korrekt. Semantisch mehrdeutig. [kein Analogon]

Syntax and Semantics of $\mathrm{C}{++}$

Syntax

- What *is* a $\mathrm{C}{++}$ program?
- Is it *grammatically* correct?

Semantics

- What does a program *mean*?
- What kind of algorithm does a program implement?

Syntax and semantics of $\mathrm{C}{++}$

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

- is the “law” of $\mathrm{C}{++}$
- defines the grammar and meaning of $\mathrm{C}{++}$programs
- contains new concepts for *advanced* programming . . .
- . . . which is why we will not go into details of such concepts

Programming Tools

- **Editor:** Program to modify, edit and store $\mathrm{C}{++}$program texts
- **Compiler:** program to translate a program text into machine language
- **Computer:** machine to execute machine language programs
- **Operating System:** program to organize all procedures such as file handling, editor-, compiler- and program execution.

Language constructs with an example

- Comments/layout
- Include directive
- the main function
- Values effects
- Types and functionality
- literals
- variables
- constants
- identifiers, names
- objects
- **expressions**
- L- and R- values
- operators
- statements

The first $\mathrm{C}{+}{+}$ program Most important ingredients...

```
// 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;          ← Statements: Do something (read in a)!
    // computation
    int b = a * a; // b = a^2   ← Expressions: Compute a value (a^2)!
    b = b * b;     // b = a^4
    // output b * b, i.e., a^8
    std::cout << a << "^8 = " << b * b << "\n";
    return 0;
}
```


Behavior of a Program

At compile time:

- program accepted by the compiler (syntactically correct)
- Compiler error

During runtime:

- correct result
- incorrect result
- program crashes
- program does not terminate (endless loop)

"Accessories:" Comments

```
// 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;
}
```

comments

Comments and Layout

Comments

- are contained in every good program.
- document *what* and *how* a program does something and how it should be used,
- are ignored by the compiler
- Syntax: “double slash” `// until the line ends.`

The compiler *ignores* additionally

- Empty lines, spaces,
- Indendations that should reflect the program logic

Comments and Layout

The compiler does not care...

```
#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;}
```

... but we do!

“Accessories:” Include and Main Function

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

include directive

declaration of the main function

Include Directives

C++ consists of

- the core language
- standard library
  - in-/output (header iostream)
  - mathematical functions (cmath)
  - ...

`#include <iostream>`

- makes in- and output available

The main Function

the **`main`**-function

- is provided in any C++ program
- is called by the operating system
- like a mathematical function ...
  - arguments
  - return value
- ... but with an additional *effect*
  - Read a number and output the 8th power.

Statements: Do something!

```
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;
}
```

expression statements

return statement

Statements

- building blocks of a C++ program
- are *executed* (sequentially)
- end with a semicolon
- Any statement has an *effect* (potentially)

Expression Statements

- have the following form:

  expr;

  where *expr* is an expression
- Effect is the effect of *expr*, the value of *expr* is ignored.

Example: `b = b*b;`

Return Statements

- do only occur in functions and are of the form

  `return` expr;

  where *expr* is an expression
- specify the return value of a function

Example: `return 0;`

Statements – Effects

```
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;
}
```

effect: output of the string `Compute ...`

Effect: input of a number stored in a

Effect: saving the computed value of a*a into b

Effect: saving the computed value of b*b into b

Effect: return the value 0

Effect: output of the value of a and the computed value c

Values and Effects

- determine what a program does,
- are purely semantical concepts:
  - Symbol 0 means Value $0 \in \mathbb{Z}$
  - `std::cin >> a;` means effect "read in a number"
- depend on the program state (memory content, inputs)

Statements – Variable Definitions

```
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;
}
```

declaration statement

type names

Declaration Statements

- introduce new names in the program,
- consist of declaration and semicolon

Example: `int a;`

- can initialize variables

Example: `int b = a * a;`

Types and Functionality

`int`:

- $\mathrm{C}{+}{+}$ integer type
- corresponds to $(\mathbb{Z}, +, \times)$ in math

In $\mathrm{C}{+}{+}$ each type has a name and

- a domain (e.g. integers)
- functionality (e.g. addition/multiplication)

Fundamental Types

$\mathrm{C}{+}{+}$ comprises fundamental types for

- integers (`int`)
- natural numbers (`unsigned int`)
- real numbers (`float`, `double`)
- boolean values (`bool`)
- ...

Literals

- represent constant values
- have a fixed *type* and *value*
- are "syntactical values".

Examples:

- `0` has type `int`, value 0.
- `1.2e5` has type `double`, value $1.2 \cdot 10^5$.

Variables

- represent (varying) values,
- have
  - *name*
  - *type*
  - *value*
  - *address*
- are "visible" in the program context.

Beispiel

`int a;` defines a variable with

- name: `a`
- type: `int`
- value: (initially) undefined
- Address: determined by compiler

Objects

- represent values in main memory
- have *type*, *address* and *value* (memory content at the address)
- can be named (variable) ...
- ... but also anonymous.

Remarks

A program has a *fixed* number of variables. In order to be able to deal with a variable number of value, it requires "anonymous" addresses that can be address via temporary names.

Identifiers and Names

(Variable-)names are identifiers

- allowed: A,...,Z; a,...,z; 0,...,9;_
- First symbol needs to be a character.

There are more names:

- `std::cin` (Qualified identifier)

Expressions: compute a value!

- represent *Computations*
- are either primary (`b`)
- or composed (`b*b`)...
- ...from different expressions, using operators
- have a type and a value

Analogy: building blocks

Expressions — Building Blocks

composite expression

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

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

Two times composed expression

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

return (
```

Four times composed expression

Expressions

- represent *computations*
- are *primary* or *composite* (by other expressions and operations)

`a * a`
composed of
variable name, operator symbol,variable name
variable name: primary expression

- can be put into parantheses

`a * a` is equivalent to `(a * a)`

Expressions

have *type*, *value* und *effect* (potentially).

Example

`a * a`

- type: `int` (type of the operands)
- Value: product of `a` and `a`
- Effect: none.

Example

`b = b * b`

- type: `int` (Typ der Operanden)
- Value: product of `b` and `b`
- effect: assignment of the product value to `b`

The type of an expression is fixed but the value and effect are only determined by the *evaluation* of the expression

L-Values and R-Values

R-Value

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

L-value (expression + address)

L-value (expression + address)

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

R-Value

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

R-Value (expression that is not an L-value)

L-Values and R-Values

L-Wert ("**L**eft of the assignment operator")

- Expression with *address*
- *Value* is the content at the memory location according to the type of the expression.
- L-Value can change its value (e.g. via assignment)

> Example: variable name

L-Values and R-Values

R-Wert ("**R**ight of the assignment operator")

- Expression that is no L-value

> Example: literal 0

- Any L-Value can be used as R-Value (but not the other way round)
- An R-Value *cannot change* its value

Operators and Operands — Building Blocks

left operand (output stream)
output operator
right operand (string)

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

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

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

right operand (variable name)
input operator
left operand (input stream)
assignment operator
multiplication operator

Operators

Operators

- combine expressions (*operands*) into new composed expressions
- specify for the operands and the result the types and if the have to be L- or R-values.
- have an arity

Multiplication Operator *

- expects two R-values of the same type as operands (arity 2)
- "returns the product as R-value of the same type", that means formally:
  - The composite expression is an R-value; its value is the product of the value of the two operands

Examples: `a * a` and `b * b`

Assignment Operator =

- **L**eft operand is **L**-value,
- **R**ight operand is **R**-value of the same type.
- Assigns to the left operand the value of the right operand and returns the left operand as L-value

Examples: `b = b * b` and `a = b`

Attention, Trap!

The operator = corresponds to the assignment operator of mathematics ($:=$), not to the comparison operator ($=$).

Input Operator >>

- left operand is L-Value (input stream)
- right operand is L-Value
- assigns to the right operand the next value read from the input stream, *removing it from the input stream* and returns the input stream as L-value

  Example `std::cin >> a` (mostly keyboard input)
- Input stream is being changed and must thus be an L-Value.

Output Operator <<

- left operand is L-Value (*output stream*)
- right operand is R-Value
- outputs the value of the right operand, appends it to the output stream and returns the output stream as L-Value

  Example: `std::cout << a` (mostly console output)
- The output stream is being changed and must thus be an L-Value.

Output Operator <<

Why returning the output stream?

- allows bundling of output

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

is parenthesized as follows

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

- `std::cout << a` is the left hand operand of the next << and is thus an L-Value that is no variable name