Informatik I

Vorlesung am D-ITET der ETH Zürich

Felix Friedrich

HS 2017

Welcome

to the Course Informatik I !

at the ITET departement of ETH Zürich.

Place and time:

Wednesday 8:15 - 10:00, ETF E1.
Pause 9:00 - 9:15, slight shift possible.

Course web page

`http://lec.inf.ethz.ch/itet/informatik1`

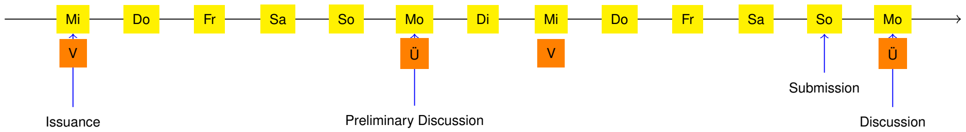
Team

chef assistant	Martin Bättig	
assistants	Ivana Unkovic	Francois Serre
	Hossein Shafagh	Marc Bitterli
	Christoph Amevor	Temmy Bounedjar
	Michael Prasthofer	Sean Bone
	Patrik Hadorn	Nathaneal Köhler
	Robin Worreby	Alexander Hedges
	Christelle Gloor	Yvan Bosshard
	Alessio Bähler	
lecturer	FF	

Recitation Session Registry

- Registration via web page `http://echo.ethz.ch`
- Works only when enrolled for this course via myStudies.
- Available rooms depend on the course of studies.

Procedure



- Exercises availabe at lectures.
- Preliminary discussion in the following recitation session
- Solution of the exercise until the day before the next recitation session.
- Dicussion of the exercise in the next recitation session.

Exercises

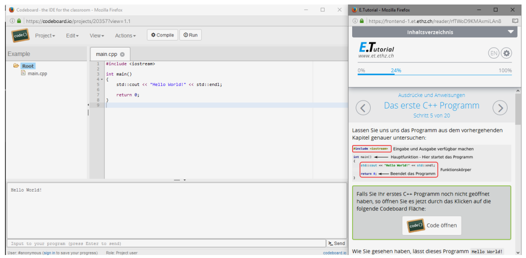
- At ETH an exercise certificate is not required in order to subscribe for the exams.
- The solution of the weekly exercises is thus voluntary but *stronly* recommended.

No lacking resources!

For the exercises we use an online development environment that requires only a browser, internet connection and your ETH login.

If you do not have access to a computer: there are a a lot of computers publicly accessible at ETH.

Online Tutorial



For a smooth course entry we provide an *online C++ tutorial*

Goal: leveling of the different programming skills.

Written mini test for your *self assessment* in the first recitation session.

Exams

The exam (in examination period 2018) will cover

- Lectures content (lectures, handouts)
- Exercise content (exercise sessions, exercises).

Written exam without any examination adds.

We will test your practical skills (programming skills [1]) and theoretical knowledge (background knowledge, systematics).

[1] as far as possible in a written exam

Offer

- During the semester we offer weekly programming exercises that are graded. Points achieved will be taken as a bonus to the exam.
- The achieved grade bonus is proportional to the achieved points of all exercise series. Achieving all points corresponds to 1/4 grade.

Academic integrity

Rule: You submit solutions that you have written yourself and that you have understood.

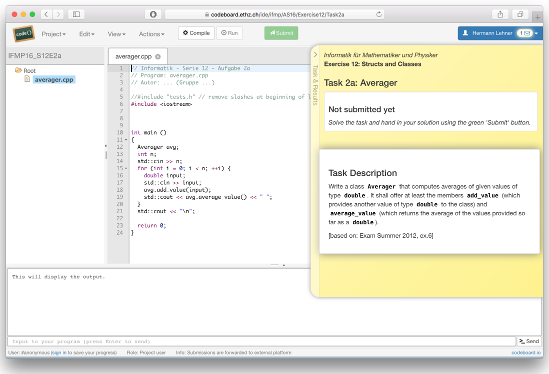
We check this (partially automatically) and reserve our rights to invite you to interviews.

Should you be invited to an interview: don't panic. Primary we presume your innocence and want to know if you understood what you have submitted.

Codeboard

Codeboard is an online IDE: programming in the browser!

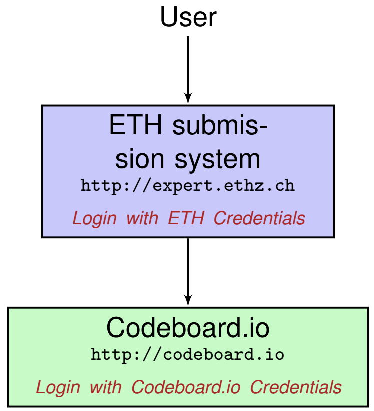
- Bring your laptop / tablet / . . . along, if available.
- You can try out examples in class without having to install any tools.



Expert

Our exercise system consists of two independent systems that communicate with each other:

- **The ETH submission system:** Allows us to evaluate your tasks.
- **The online IDE:** The programming environment



Exercise Registration

Codeboard.io Registration

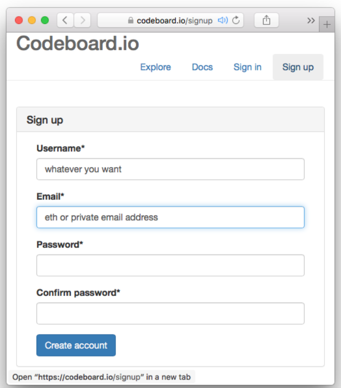
Go to `http://codeboard.io` and create an account, stay logged in.

Registration for exercises

Go to `http://expert.ethz.ch/ifee1y17e01t1` and inscribe for one of the exercise groups there.

Codeboard.io Registration

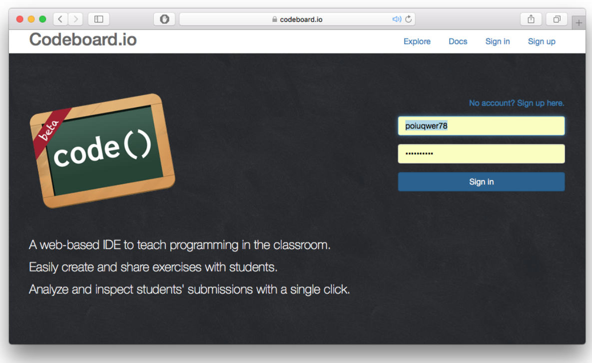
If you do not yet have an **Codeboard.io** account ...



- We use the online IDE **Codeboard.io**
- Create an account to store your progress and be able to review submissions later on
- Credentials can be chose arbitrarily *Do not use the ETH password.*

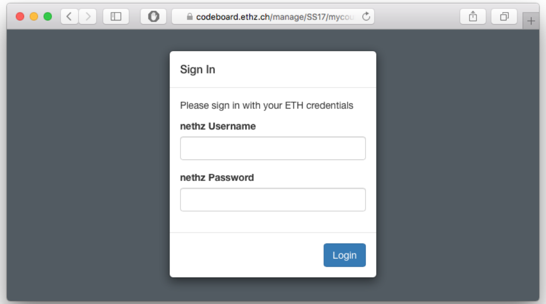
Codeboard.io Login

If you have an account, log in:



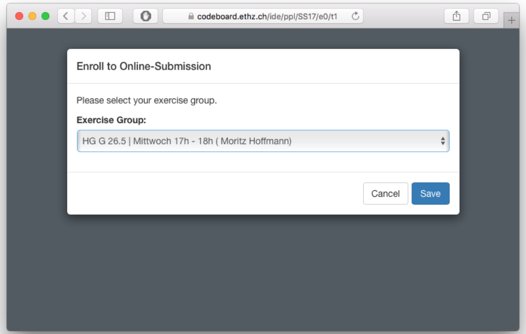
Exercise group registration I

- Visit `http://expert.ethz.ch/ifee1y17e01t1`
- Log in with your nethz account.



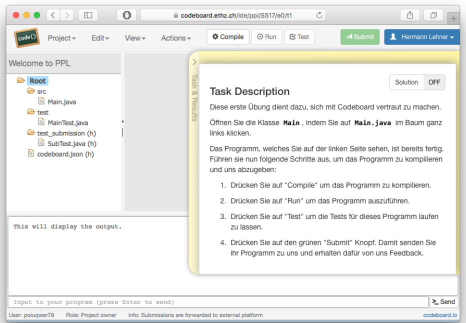
Exercise group registration II

Register with this dialog for an exercise group.



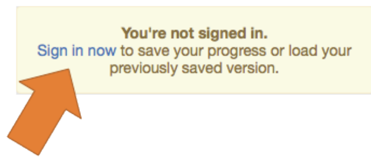
The first exercise.

You are now registered and the first exercise is loaded. Follow the instructions in the yellow box.



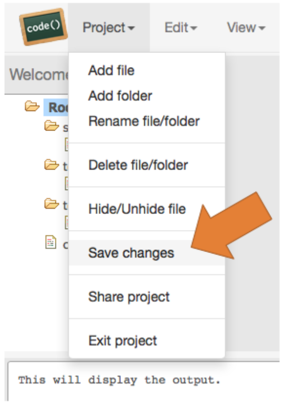
The first exercise – codeboard.io login

Attention If you see this message, click on Sign in now and register with you **codeboard.io** account.



The first exercise – store progress

Attention! Store your progress regularly. So you can continue working at any different location.



Literature

- The course is designed to be self explanatory.
- Skript together with the course Informatik at the D-MATH/D-PHYS department.
- Recommended Literature
  - B. Stroustrup. *Einführung in die Programmierung mit C++*, Pearson Studium, 2010.
  - B. Stroustrup, *The C++ Programming Language* (4th Edition) Addison-Wesley, 2013.
  - A. Koenig, B.E. Moo, *Accelerated C++*, Adddison Wesley, 2000.
  - B. Stroustrup, *The design and evolution of C++*, Addison-Wesley, 1994.

Credits

- Course structure developed together with Prof. Bernd Gärtner
- Skript from Prof. Bernd Gärtner.

Andere Quellen werden hier am Rand in dieser Form angegeben.

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)

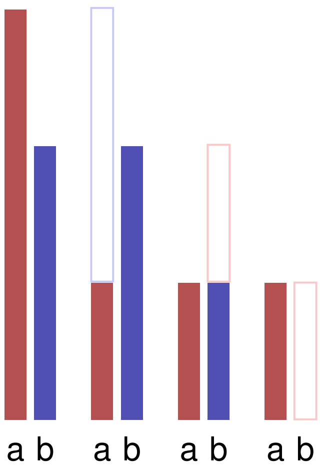
http://de.wikipedia.org/wiki/Algorithmus

Oldest Nontrivial Algorithm

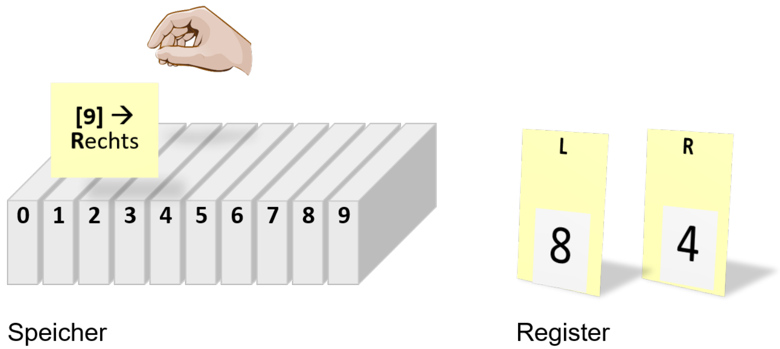
Euclidean algorithm (from the *elements* from Euklid, 3. century B.C.)

- 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



Euklid in the Box

Speicher

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

$\underbrace{\text{0–7}}_{\text{Programmcode}}$ $\underbrace{\text{8–9}}_{\text{Daten}}$

Daten

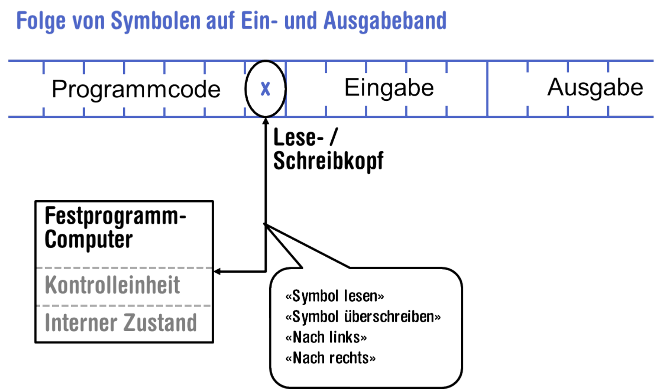
Links	**R**echts
b	a

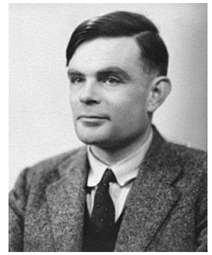
Register

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)

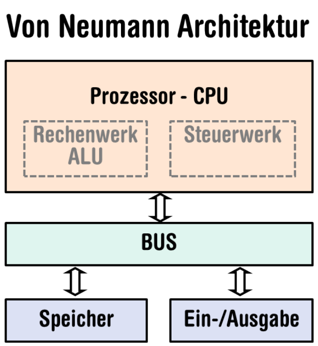



Alan Turing

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

Computer – Implementation

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




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

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

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 → 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.

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 C++

Syntax

- What *is* a C++ program?
- Is it *grammatically* correct?

Semantics

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

Syntax and semantics of C++

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

- is the “law” of C++
- defines the grammar and meaning of 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 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 $C{+}{+}$ 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;
    // 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;
}
```

Statements: Do something (read in a)!

Expressions: Compute a value (a^2)!

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>      include directive
int main() {             declaration of the main function
    // 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 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`:

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

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

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

Fundamental Types

$C{+}{+}$ 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.

Example

`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;

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

return (
```

Two times composed expression

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)

// computation          L-value (expression + address)
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

```
// 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;
```

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

- Left operand is L-value,
- Right 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

2. Integers

Evaluation of Arithmetic Expressions, Associativity and Precedence, Arithmetic Operators, Domain of Types `int`, `unsigned int`

Celsius to Fahrenheit

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

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

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


`9 * celsius / 5 + 32`

- Arithmetic expression,
- contains three literals, a variable, three operator symbols

How to put the expression in parentheses?

Precedence

Multiplication/Division before Addition/Subtraction

`9 * celsius / 5 + 32`

bedeutet

`(9 * celsius / 5) + 32`

Rule 1: precedence

Multiplicative operators (`*`, `/`, `%`) have a higher precedence ("bind more strongly") than additive operators (`+`, `-`)

Associativity

From left to right

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

bedeutet

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

Rule 2: Associativity

Arithmetic operators (*, /, %, +, −) are left associative: operators of same precedence evaluate from left to right

Arity

Rule 3: Arity

Unary operators +, − first, then binary operators +, −.

```
-3 - 4
```

means

```
(-3) - 4
```


Parentheses

Any expression can be put in parentheses by means of

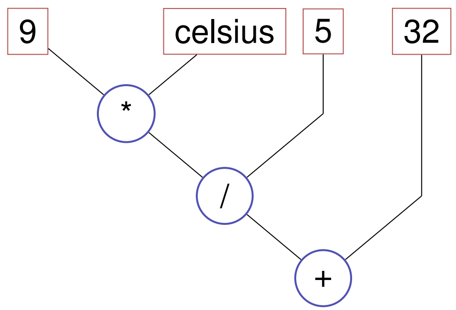
- associativities
- precedences
- arities (number of operands)

of the operands in an unambiguous way (Details in the lecture notes).

Expression Trees

Parentheses yield the expression tree

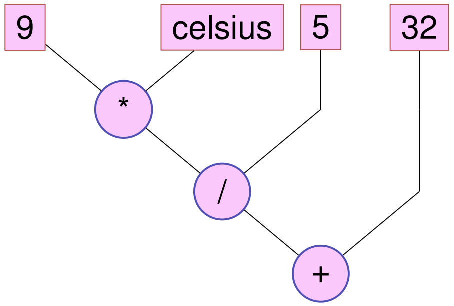
```
(((9 * celsius) / 5) + 32)
```



Evaluation Order

"From top to bottom" in the expression tree

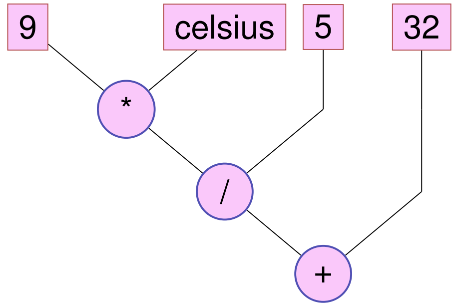
`9 * celsius / 5 + 32`



Evaluation Order

Order is not determined uniquely:

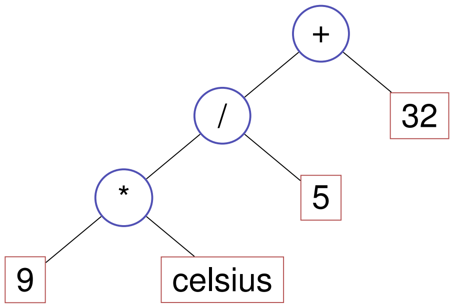
`9 * celsius / 5 + 32`



Expression Trees – Notation

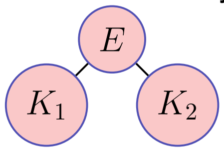
Common notation: root on top

`9 * celsius / 5 + 32`



Evaluation Order – more formally

- Valid order: any node is evaluated *after* its children



In C++, the valid order to be used is not defined.

- "Good expression": any valid evaluation order leads to the same result.
- Example for a "bad expression": `(a+b)*(a++)`

Evaluation order

Guideline

Avoid modifying variables that are used in the same expression more than once.

Arithmetic operations

	Symbol	Arity	Precedence	Associativity
Unary +	+	1	16	right
Negation	-	1	16	right
Multiplication	*	2	14	left
Division	/	2	14	left
Modulus	%	2	14	links
Addition	+	2	13	left
Subtraction	-	2	13	left

All operators: [R-value $\times$] R-value $\rightarrow$ R-value

Assignment expression – in more detail

- Already known: `a = b` means
  Assignment of `b` (R-value) to `a` (L-value).
  Returns: L-value
- What does `a = b = c` mean?
- Answer: assignment is right-associative

`a = b = c` $\iff$ `a = (b = c)`

Example multiple assignment:
`a = b = 0` $\Longrightarrow$ `b=0; a=0`

Division and Modulus

- Operator / implements integer division

`5 / 2` has value 2

- In `fahrenheit.cpp`

```
9 * celsius / 5 + 32

15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematically equivalent. . . but not in C++!

```
9 / 5 * celsius + 32

15 degrees Celsius are 47 degrees Fahrenheit
```

Division and Modulus

- Modulus-operator computes the rest of the integer division

`5 / 2` has value 2,   `5 % 2` has value 1.

- It holds that:

`(a / b) * b + a % b` has the value of `a`.

Increment and decrement

- Increment / Decrement a number by one is a frequent operation
- works like this for an L-value:

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

Disadvantages

- relatively long
- expr is evaluated twice (effects!)

In-/Decrement Operators

Post-Increment

```
expr++
```

Value of expr is increased by one, the *old* value of expr is returned (as R-value)

Pre-increment

```
++expr
```

Value of expr is increased by one, the *new* value of expr is returned (as L-value)

Post-Dekrement

```
expr--
```

Value of expr is decreased by one, the *old* value of expr is returned (as R-value)

Prä-Dekrement

```
--expr
```

Value of expr is increased by one, the *new* value of expr is returned (as L-value)

In-/decrement Operators

	use	arity	prec	assoz	L-/R-value
Post-increment	`expr++`	1	17	left	L-value → R-value
Pre-increment	`++expr`	1	16	right	L-value → L-value
Post-decrement	`expr--`	1	17	left	L-value → R-value
Pre-decrement	`--expr`	1	16	right	L-value → L-value

In-/Decrement Operators

Example

```
int a = 7;
std::cout << ++a << "\n"; // 8
std::cout << a++ << "\n"; // 8
std::cout << a << "\n"; // 9
```


In-/Decrement Operators

Is the expression

`++expr;` ← we favour this

equivalent to

`expr++;`?

Yes, but

- Pre-increment can be more efficient (old value does not need to be saved)
- Post In-/Decrement are the only left-associative unary operators (not very intuitive)

C++ vs. ++C

Strictly speaking our language should be named ++C because

- it is an advancement of the language C
- while C++ returns the old C.

Arithmetic Assignments

`a += b`

⇔

`a = a + b`

analogously for `-`, `*`, `/` and `%`

Arithmetic Assignments

	Gebrauch	Bedeutung
`+=`	`expr1 += expr2`	`expr1 = expr1 + expr2`
`-=`	`expr1 -= expr2`	`expr1 = expr1 - expr2`
`*=`	`expr1 *= expr2`	`expr1 = expr1 * expr2`
`/=`	`expr1 /= expr2`	`expr1 = expr1 / expr2`
`%=`	`expr1 %= expr2`	`expr1 = expr1 % expr2`

Arithmetic expressions evaluate expr1 only once.
Assignments have precedence 4 and are right-associative.

Binary Number Representations

Binary representation ("Bits" from $\{0, 1\}$)

$$b_n b_{n-1} \dots b_1 b_0$$

corresponds to the number $b_n \cdot 2^n + \dots + b_1 \cdot 2 + b_0$

Example: `101011` corresponds to 43.

Least Significant Bit (LSB)

Most Significant Bit (MSB)

Binary Numbers: Numbers of the Computer?

Truth: Computers calculate using binary numbers.


NEUE ZÜRCHER ZEITUNG TECHNIK Mittwoch, 30. August 1950 Blatt 5 Mittagausgabe Nr. 1796 (50)

Das programmgesteuerte Rechengerät an der Eidgenössischen Technischen Hochschule in Zürich

Die Entwicklung programmgesteuerter Rechenmaschinen in den Vereinigten Staaten von Amerika wurde in den Artikeln „Elektronische Rechenmaschinen" (vgl. Nr. 2140 der „N. Z. Z." vom 13. Oktober 1948) und „Die neueste elektronische Rechenmaschine" (vgl. Nr. 871 der „N. Z. Z." vom 26. April 1950) behandelt. Nachstehend soll von einem Gerät deutscher Herkunft — Zuse K.-G., Neukirchen — die Rede sein, welches im Juli dieses Jahres am Institut für angewandte Mathematik der Eidgenössischen Technischen Hochschule in Zürich, das unter der Leitung von Prof. Dr. E. Stiefel steht, in Betrieb genommen wurde. Damit ist dieses Institut in der Lage, dem in der Schweiz immer stärker werdenden Bedürfnis nach einer leistungsfähigen Zentralstelle für numerische Rechnungen wenigstens teilweise gerecht zu werden. Bereits sind einige mathematische Probleme behandelt worden,

Aenderung des Maßstabes durchgerechnet werden können.

Die beschriebene Darstellung bringt eine gewisse Komplikation der Rechenoperationen mit sich. So müssen vor einer Addition die beiden Summanden zunächst so verschoben werden, daß ihre Kommata untereinander zu liegen kommen, was an Hand eines Beispiels erläutert werden soll. Damit der Leser nicht durch das ungewohnte duale Zahl-

Binary Numbers: Numbers of the Computer?

Stereotype: computers are talking 0/1 gibberish

Computing Tricks

- Estimate the orders of magnitude of powers of two.[3]:

$2^{10} = 1024 = 1\text{Ki} \approx 10^3$.
$2^{20} = 1\text{Mi} \approx 10^6$,
$2^{30} = 1\text{Gi} \approx 10^9$,
$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi}$.
$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}$.

[3] Decimal vs. binary units: MB - Megabyte vs. MiB - Megabibyte (etc.)
kilo (K, Ki) – mega (M, Mi) – giga (G, Gi) – tera(T, Ti) – peta(P, Pi) – exa (E, Ei)

Hexadecimal Numbers

Numbers with base 16

$$h_n h_{n-1} \dots h_1 h_0$$

corresponds to the number

$$h_n \cdot 16^n + \dots + h_1 \cdot 16 + h_0.$$

notation in C++: prefix `0x`

Example: `0xff` corresponds to 255.

Hex Nibbles

hex	bin	dec
0	0000	0
1	**0001**	**1**
2	**0010**	**2**
3	0011	3
4	**0100**	**4**
5	0101	5
6	0110	6
7	0111	7
8	**1000**	**8**
9	1001	9
a	1010	10
b	1011	11
c	1100	12
d	1101	13
e	1110	14
f	1111	15

Why Hexadecimal Numbers?

- A Hex-Nibble requires exactly 4 bits. Numbers 1, 2, 4 and 8 represent bits 0, 1, 2 and 3.
- „compact representation of binary numbers"

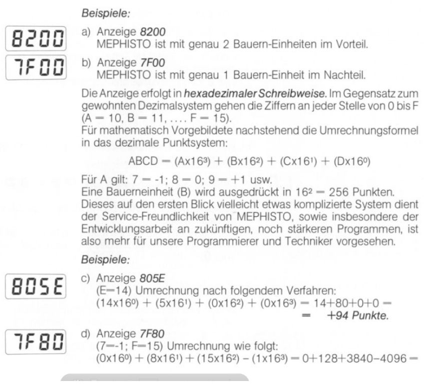
32-bit numbers consist of eight hex-nibbles: `0x00000000 -- 0xffffffff` .
`0x400` $= 1Ki = 1'024$.
`0x100000` $= 1Mi = 1'048'576$.
`0x40000000` $= 1Gi = 1'073.741,824$.
`0x80000000`: highest bit of a 32-bit number is set
`0xffffffff`: all bits of a 32-bit number are set
„`0x8a20aaf0` is an address in the upper 2G of the 32-bit address space"

Example: Hex-Colors

$$\#\underbrace{00}_{r}\underbrace{FF}_{g}\underbrace{00}_{b}$$

Why Hexadecimal Numbers?

“For programmers and technicians” (Excerpt of a user manual of the chess computers *Mephisto II*, 1981)



Beispiele:

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

7F00 b) Anzeige ***7F00***
MEPHISTO ist mit genau 1 Bauern-Einheit im Nachteil.

Die Anzeige erfolgt in ***hexadezimaler Schreibweise.*** Im Gegensatz zum gewohnten Dezimalsystem gehen die Ziffern an jeder Stelle von 0 bis F (A = 10, B = 11, ... F = 15).
Für mathematisch Vorgebildete nachstehend die Umrechnungsformel in das dezimale Punktsystem:

$$ABCD = (A\text{x}16^3) + (B\text{x}16^2) + (C\text{x}16^1) + (D\text{x}16^0)$$

Für A gilt: 7 = -1; 8 = 0; 9 = +1 usw.
Eine Bauerneinheit (B) wird ausgedrückt in $16^2 = 256$ Punkten.
Dieses auf den ersten Blick vielleicht etwas komplizierte System dient der Service-Freundlichkeit von MEPHISTO, sowie insbesondere der Entwicklungsarbeit an zukünftigen, noch stärkeren Programmen, ist also mehr für unsere Programmierer und Techniker vorgesehen.

Beispiele:

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

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

http://www.zanchetta.net/default.aspx?Categorie=ECHIQUIERS&Page=documentations

Why Hexadecimal Numbers?

The NZZ could have saved a lot of space ...

Domain of Type `int`

```
// Program: limits.cpp
// Output the smallest and the largest value of type int.

#include <iostream>
#include <limits>

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

For example

```
Minimum int value is -2147483648.
Maximum int value is 2147483647.
```

Where do these numbers come from?

Domain of the Type `int`

- Representation with B bits. Domain comprises the 2^B integers:

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

- On most platforms $B = 32$
- For the type `int` $\mathrm{C}{++}$ guarantees $B \geq 16$
- Background: Section 2.2.8 (Binary Representation) in the lecture notes.

Where does this partitioning come from?

Over- and Underflow

- Arithmetic operations (`+,-,*`) can lead to numbers outside the valid domain.
- Results can be incorrect!

`power8.cpp`: $15^8 = -1732076671$

`power20.cpp`: $3^{20} = -808182895$

- There is *no error message!*

The Type `unsigned int`

- Domain

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

- All arithmetic operations exist also for `unsigned int`.
- Literals: `1u, 17u` ...

Mixed Expressions

- Operators can have operands of different type (e.g. `int` and `unsigned int`).

```
17 + 17u
```

- Such mixed expressions are of the “more general” type `unsigned int`.
- `int`-operands are *converted* to `unsigned int`.

Conversion

`int` Value	Sign	`unsigned int` Value
x	≥ 0	x
x	< 0	$x + 2^B$

Using two complements representation, nothing happens internally

Conversion “reversed”

The declaration

```
int a = 3u;
```

converts `3u` to `int`.

The value is preserved because it is in the domain of `int`; otherwise the result depends on the implementation.

Signed Number Representation

- (Hopefully) clear by now: binary number representation without sign, e.g.

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

- Obviously required: use a bit for the sign.
- Looking for a consistent solution

The representation with sign should coincide with the unsigned solution as much as possible. Positive numbers should arithmetically be treated equal in both systems.

Computing with Binary Numbers (4 digits)

Simple Addition

$$\begin{array}{rr} 2 & 0010 \\ +3 & +0011 \\ \hline 5 & 0101 \end{array}$$

Simple Subtraction

$$\begin{array}{rr} 5 & 0101 \\ -3 & -0011 \\ \hline 2 & 0010 \end{array}$$

Computing with Binary Numbers (4 digits)

Addition with Overflow

$$\begin{array}{rr} 7 & 0111 \\ +9 & +1001 \\ \hline 16 & (1)0000 \end{array}$$

Negative Numbers?

$$\begin{array}{rr} 5 & 0101 \\ +(-5) & ???? \\ \hline 0 & (1)0000 \end{array}$$

Computing with Binary Numbers (4 digits)

Simpler -1

$$\begin{array}{rr} 1 & 0001 \\ +(-1) & 1111 \\ \hline 0 & (1)0000 \end{array}$$

Utilize this:

$$\begin{array}{rr} 3 & 0011 \\ +? & +???? \\ \hline -1 & 1111 \end{array}$$

Computing with Binary Numbers (4 digits)

Invert!

$$\begin{array}{rr} 3 & 0011 \\ +(-4) & +1100 \\ \hline -1 & 1111 \,\widehat{=}\, 2^B - 1 \end{array}$$

$$\begin{array}{rr} a & a \\ +(-a-1) & \bar{a} \\ \hline -1 & 1111 \,\widehat{=}\, 2^B - 1 \end{array}$$

Computing with Binary Numbers (4 digits)

- Negation: inversion and addition of 1

$$-a \quad \widehat{=} \quad \bar{a} + 1$$

- Wrap around semantics (calculating modulo 2^B

$$-a \quad \widehat{=} \quad 2^B - a$$

Why this works

Modulo arithmetics: Compute on a circle[4]

$11 \equiv 23 \equiv -1 \equiv \ldots \mod 12$ + $4 \equiv 16 \equiv \ldots \mod 12$ = $3 \equiv 15 \equiv \ldots \mod 12$

[4]The arithmetics also work with decimal numbers (and for multiplication).

Negative Numbers (3 Digits)

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

The most significant bit decides about the sign.

Two's Complement

- Negation by bitwise negation and addition of 1

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

  - Arithmetics of addition and subtraction *identical* to unsigned arithmetics

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

  - Intuitive "wrap-around" conversion of negative numbers.

$$-n \to 2^B - n$$

  - Domain: $-2^{B-1} \ldots 2^{B-1} - 1$

3. Logical Values

Boolean Functions; the Type `bool`; logical and relational operators; shortcut evaluation

Our Goal

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

Behavior depends on the value of a Boolean expression

Boolean Values in Mathematics

Boolean expressions can take on one of two values:

0 or *1*

- *0* corresponds to *“false”*
- *1* corresponds to *“true”*

The Type `bool` in C++

- represents *logical values*
- Literals `false` and `true`
- Domain {*false*, *true*}

```
bool b = true; // Variable with value true
```


Relational Operators

`a < b` (smaller than)
`a >= b` (greater than)
`a == b` (equals)
`a != b` (not equal)

arithmetic type × arithmetic type → `bool`

R-value × R-value → R-value

Table of Relational Operators

	Symbol	Arity	Precedence	Associativity
smaller	<	2	11	left
greater	>	2	11	left
smaller equal	<=	2	11	left
greater equal	>=	2	11	left
equal	==	2	10	left
unequal	!=	2	10	left

arithmetic type × arithmetic type → `bool`

R-value × R-value → R-value

Boolean Functions in Mathematics

- Boolean function

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

- 0 corresponds to “false”.
- 1 corresponds to “true”.

$\mathrm{AND}(x, y)$ $x \wedge y$

- “logical And”

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

- 0 corresponds to “false”.
- corresponds to “true”.

x	y	$\mathrm{AND}(x,y)$
0	0	0
0	1	0
1	0	0
1	1	1

Logical Operator &&

`a && b` (logical and)

`bool` $\times$ `bool` $\to$ `bool`

R-value $\times$ R-value $\to$ R-value

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


$\mathrm{OR}(x, y)$ $x \vee y$

- “logical Or”

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

- 0 corresponds to “false”.
- corresponds to “true”.

x	y	$\mathrm{OR}(x,y)$
0	0	0
0	1	1
1	0	1
1	1	1

Logical Operator ||

`a || b` (logical or)

`bool` $\times$ `bool` $\to$ `bool`

R-value $\times$ R-value $\to$ R-value

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


$\mathrm{NOT}(x)$ $\neg x$

- "logical Not"

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

- 0 corresponds to "false".
- corresponds to "true".

x	$\mathrm{NOT}(x)$
0	1
1	0

Logical Operator !

`!b` (logical not)

`bool` $\to$ `bool`

R-value $\to$ R-value

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


Precedences

`!b && a`

$\Updownarrow$

`(!b) && a`

`a && b || c && d`

$\Updownarrow$

`(a && b) || (c && d)`

`a || b && c || d`

$\Updownarrow$

`a || (b && c) || d`

Table of Logical Operators

	Symbol	Arity	Precedence	Associativity
Logical and (AND)	&&	2	6	left
Logical or (OR)	||	2	5	left
Logical not (NOT)	!	1	16	right

Precedences

The unary logical operator `!`

binds more strongly than

binary arithmetic operators. These

bind more strongly than

relational operators,

and these bind more strongly than

binary logical operators.

```
7 + x < y && y != 3 * z || ! b
7 + x < y && y != 3 * z || (!b)
```


Completeness

- AND, OR and NOT are the boolean functions available in C++.
- Any other *binary* boolean function can be generated from them.

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

Completeness: $\mathrm{XOR}(x, y)$ $x \oplus y$

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

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

```
(x || y) && !(x && y)
```

Completeness Proof

- Identify binary boolean functions with their characteristic vector.

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

characteristic vector: 0110

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

Completeness Proof

- Step 1: generate the *fundamental* functions $f_{0001}, f_{0010}, f_{0100}, f_{1000}$

$$\begin{aligned} f_{0001} &= \text{AND}(x,y) \\ f_{0010} &= \text{AND}(x, \text{NOT}(y)) \\ f_{0100} &= \text{AND}(y, \text{NOT}(x)) \\ f_{1000} &= \text{NOT}(\text{OR}(x,y)) \end{aligned}$$

Completeness Proof

- Step 2: generate all functions by applying logical or

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

- Step 3: generate f_{0000}

$$f_{0000} = 0.$$

`bool` vs `int`: Conversion

- `bool` can be used whenever `int` is expected – and vice versa.
- Many existing programs use `int` instead of `bool`
  This is bad style originating from the language C .

`bool`	$\rightarrow$	`int`
true	$\rightarrow$	1
false	$\rightarrow$	0

`int`	$\rightarrow$	`bool`
$\neq$0	$\rightarrow$	*true*
0	$\rightarrow$	*false*

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

DeMorgan Rules

- `!(a && b) == (!a || !b)`
- `!(a || b) == (!a && !b)`

! (rich *and* beautiful) == (poor *or* ugly)

Application: either ... or (XOR)

`(x || y)`	`&& !(x && y)`	x or y, and not both
`(x || y)`	`&& (!x || !y)`	x or y, and one of them not
`!(!x && !y)`	`&& !(x && y)`	not none and not both
`!(!x && !y || x && y)`		not: both or none

Short circuit Evaluation

- Logical operators `&&` and `||` evaluate the *left operand first*.
- If the result is then known, the right operand will *not be* evaluated.

```
x != 0 && z / x > y
```

$\Rightarrow$ No division by 0

Sources of Errors

- Errors that the compiler can find: syntactical and some semantical errors
- Errors that the compiler cannot find: runtime errors (always semantical)

Avoid Sources of Bugs

1. Exact knowledge of the wanted program behavior

   ≫ It's not a bug, it's a feature !!≪
2. Check at many places in the code if the program is still on track!
3. Question the (seemingly) obvious, there could be a typo in the code.

Against Runtime Errors: *Assertions*

`assert(expr)`

- halts the program if the boolean expression `expr` is false
- requires `#include <cassert>`
- can be switched off

DeMorgan's Rules

Question the obvious Question the seemingly obvious!

```
// Prog: assertion.cpp
// use assertions to check De Morgan's laws

#include<cassert>

int main()
{
  bool x; // whatever x and y actually are,
  bool y; // De Morgan's laws will hold:
  assert ( !(x && y) == (!x || !y) );
  assert ( !(x || y) == (!x && !y) );
  return 0;
}
```


Switch off Assertions

```
// Prog: assertion2.cpp
// use assertions to check De Morgan's laws. To tell the
// compiler to ignore them, #define NDEBUG ("no debugging")
// at the beginning of the program, before the #includes

#define NDEBUG
#include<cassert>

int main()
{
  bool x; // whatever x and y actually are,
  bool y; // De Morgan's laws will hold:
  assert ( !(x && y) == (!x || !y) );  // ignored by NDEBUG
  assert ( !(x || y) == (!x && !y) );  // ignored by NDEBUG
  return 0;
}
```

Div-Mod Identity `a/b * b + a%b == a`

Check if the program is on track...

```
std::cout << "Dividend a =? ";
int a;
std::cin >> a;

std::cout << "Divisor b =? ";
int b;
std::cin >> b;

// check input
assert (b != 0);
```

Input arguments for calculation

Precondition for the ongoing computation

Div-Mod identity `a/b * b + a%b == a`

...and question the obvious!

```
// check input
assert (b != 0);

// compute result
int div = a / b;
int mod = a % b;

// check result
assert (div * b + mod == a);
...
```

Precondition for the ongoing computation

Div-Mod identity

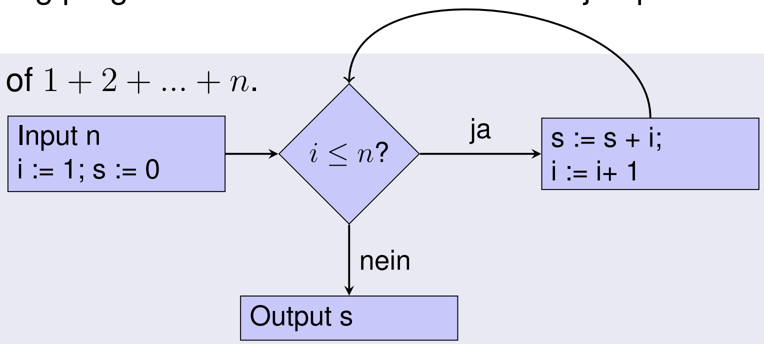
4. Control Structures I

Selection Statements, Iteration Statements, Termination, Blocks

Control Flow

- up to now *linear* (from top to bottom)
- For interesting programs we need “branches” and “jumps”

Computation of $1 + 2 + ... + n$.



Selection Statements

implement branches

- `if` statement
- `if-else` statement

`if`-Statement

`if` (*condition*)
    statement

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

If *condition* is true then *statement* is executed

- *statement*: arbitrary statement (*body* of the `if`-Statement)
- *condition*: convertible to `bool`

`if-else`-statement

`if` (*condition*)
    statement1
else
    statement2

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

If *condition* is true then *statement1* is executed, otherwise *statement2* is executed.

- *condition*: convertible to `bool`.
- *statement1*: *body* of the `if`-branch
- *statement2*: *body* of the `else`-branch

Layout!

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

Iteration Statements

implement “loops”

- `for`-statement
- `while`-statement
- `do`-statement

Compute $1 + 2 + ... + n$

```
// Program: sum_n.cpp
// Compute the sum of the first n natural numbers.

#include <iostream>

int main()
{
  // input
  std::cout << "Compute the sum 1+...+n for n =? ";
  unsigned int n;
  std::cin >> n;

  // computation of sum_{i=1}^n i
  unsigned int s = 0;
  for (unsigned int i = 1; i <= n; ++i) s += i;

  // output
  std::cout << "1+...+" << n << " = " << s << ".\n";
  return 0;
}
```


`for`-Statement Example

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

Assumptions: `n == 2`, `s == 0`

`i`		`s`
`i==1`	wahr	`s == 1`
`i==2`	wahr	`s == 3`
`i==3`	falsch	

`s == 3`

`for`-Statement: Syntax

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

- *init-statement*: expression statement, declaration statement, null statement
- *condition*: convertible to bool
- *expression*: any expression
- *statement* : any statement (*body* of the for-statement)

`for`-Statement: semantics

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

- *init-statement* is executed
- *condition* is evaluated
  - `true`: Iteration starts
    statement is executed
    expression is executed
  - false: `for`-statement is ended.

Gauß as a Child (1777 - 1855)

- Math-teacher wanted to keep the pupils busy with the following task:

Compute the sum of numbers from 1 to 100 !

- Gauß finished after one minute.

The Solution of Gauß

- The requested number is

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

- This is half of

$$\begin{array}{rrrrrrrrrr} & 1 & + & 2 & + & \cdots & + & 99 & + & 100 \\ + & 100 & + & 99 & + & \cdots & + & 2 & + & 1 \\ \hline = & 101 & + & 101 & + & \cdots & + & 101 & + & 101 \end{array}$$

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

`for`-Statement: Termination

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

Here and in most cases:

- *expression* changes its value that appears in *condition* .
- After a finite number of iterations *condition* becomes false: *Termination*

Infinite Loops

- Infinite loops are easy to generate:

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

  - Die *empty condition* is true.
  - Die *empty expression* has no effect.
  - Die *null statement* has no effect.
- ... but can in general not be automatically detected.

```
for ( e; v; e) r;
```


Halting Problem

Undecidability of the Halting Problem

There is no $\mathrm{C++}$ program that can determine for each $\mathrm{C++}$-Program P and each input I if the program P terminates with the input I.

This means that the correctness of programs can in general *not* be automatically checked.[5]

[5] Alan Turing, 1936. Theoretical quesitons of this kind were the main motivation for Alan Turing to construct a computing machine.

Example: Prime Number Test

Def.: a natural number $n \geq 2$ is a prime number, if no $d \in \{2, \ldots, n-1\}$ divides n .

A loop that can test this:

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

- Observation 1:
  After the `for`-statement it holds that $d \leq n$.
- Observation 2:
  n is a prime number if and only if finally $d = n$.

Blocks

- Blocks group a number of statements to a new statement
  `{statement1 statement2 ... statementN}`
- Example: body of the main function

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

- Example: loop body

```
for (unsigned int i = 1; i <= n; ++i) {
    s += i;
    std::cout << "partial sum is " << s << "\n";
}
```

5. Control Statements II

Visibility, Local Variables, While Statement, Do Statement, Jump Statements

Visibility

Declaration in a block is not “visible” outside of the block.

```
int main ()
{
    {
        int i = 2;
    }
    std::cout << i; // Error:  undeclared name
    return 0;
}
```

main block

block

„Blickrichtung”

Control Statement defines Block

In this respect, statements behave like blocks.

```
int main()
{
    for (unsigned int i = 0; i < 10; ++i)
        s += i;
    std::cout << i; // Error:  undeclared name
    return 0;
}
```

block

Scope of a Declaration

Potential scope: from declaration until end of the part that contains the declaration.

in the block

```
{
    int i = 2;
    ...
}
```

scope

in function body

```
int main() {
    int i = 2;
    ...
    return 0;
}
```

scope

in control statement

```
for ( int i = 0; i < 10; ++i) {s += i; ... }
```

scope

Scope of a Declaration

Real scope = potential scope minus potential scopes of declarations of symbols with the same name

```
int main()
{
  int i = 2;
  for (int i = 0; i < 5; ++i)
     // outputs 0,1,2,3,4
     std::cout << i;
   // outputs 2
   std::cout << i;
  return 0;
}
```

in main

i_2 in for

scope of i

Automatic Storage Duration

Local Variables (declaration in block)

- are (re-)created each time their declaration is reached
  - memory address is assigned (allocation)
  - potential initialization is executed
- are deallocated at the end of their declarative region (memory is released, address becomes invalid)

Local Variables

```
int main()
{
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        std::cout << ++i; // outputs 6, 7, 8, 9, 10
        int k = 2;
        std::cout << --k; // outputs 1, 1, 1, 1, 1
    }
}
```

Local variables (declaration in a block) have *automatic storage duration*.

`while` Statement

`while (` *condition* `)`
  statement

- *statement*: arbitrary statement, body of the `while` statement.
- *condition*: convertible to `bool`.

`while` Statement

```
while ( condition )
  statement
```

is equivalent to

```
for ( ; condition ; )
  statement
```


`while`-Statement: Semantics

```
while ( condition )
  statement
```

- *condition* is evaluated
  - `true`: iteration starts
    statement is executed
  - `false`: `while`-statement ends.

`while`-statement: why?

- In a `for`-statement, the expression often provides the progress ("counting loop")

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

- If the progress is not as simple, `while` can be more readable.

Example: The Collatz-Sequence $(n \in \mathbb{N})$

- $n_0 = n$
- $n_i = \begin{cases} \frac{n_{i-1}}{2} & \text{, if } n_{i-1} \text{ even} \\ 3n_{i-1} + 1 & \text{, if } n_{i-1} \text{ odd} \end{cases} , i \geq 1.$

n=5: 5, 16, 8, 4, 2, 1, 4, 2, 1, ... (repetition at 1)

The Collatz Sequence in $C++$

```
// Program: collatz.cpp
// Compute the Collatz sequence of a number n.

#include <iostream>

int main()
{
  // Input
  std::cout << "Compute the Collatz sequence for n =? ";
  unsigned int n;
  std::cin >> n;

  // Iteration
  while (n > 1) {
    if (n % 2 == 0)
      n = n / 2;
    else
      n = 3 * n + 1;
    std::cout << n << " ";
  }
  std::cout << "\n";
  return 0;
}
```


The Collatz Sequence in $C++$

```
n = 27:
82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242,
121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233,
700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336,
668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276,
638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429,
7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232,
4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488,
244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20,
10, 5, 16, 8, 4, 2, 1
```


The Collatz-Sequence

Does 1 occur for each n?

- It is conjectured, but nobody can prove it!
- If not, then the `while`-statement for computing the Collatz-sequence can theoretically be an endless loop for some n.

`do` Statement

do
  statement
while (*expression*);

- *statement*: arbitrary statement, body of the `do` statement.
- *expression*: convertible to `bool`.

`do` Statement

`do`
  statement
`while` (*expression*);

is equivalent to

statement
`while` (*expression*)
  statement

`do`-Statement: Semantics

`do`
  statement
`while` (*expression*);

- Iteration starts
  - *statement* is executed.
- *expression* is evaluated
  - `true`: iteration begins
  - `false`: `do`-statement ends.

`do`-Statement: Example Calculator

Sum up integers (if 0 then stop):

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


Conclusion

- Selection (conditional *branches*)
  - `if` and `if-else`-statement
- Iteration (conditional *jumps*)
  - `for`-statement
  - `while`-statement
  - `do`-statement
- Blocks and scope of declarations

Jump Statements

- `break;`
- `continue;`

`break`-Statement

```
break;
```

- Immediately leave the enclosing iteration statement.
- useful in order to be able to break a loop "in the middle" [6]

[6] and indispensible for switch-statements.

Calculator with `break`

Sum up integers (if 0 then stop)

```
int a;
int s = 0;
do {
    std::cout << "next number =? ";
    std::cin >> a;
    // irrelevant in last iteration:
    s += a;
    std::cout << "sum = " << s << "\n";
}  while (a != 0);
```


Calculator with `break`

Suppress irrelevant addition of 0:

```
int a;
int s = 0;
do {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a == 0) break; // stop loop in the middle
    s += a;
    std::cout << "sum = " << s << "\n";
} while (a != 0)
```

Calculator with `break`

Equivalent and yet more simple:

```
int a;
int s = 0;
for (;;) {
    std::cout << "next number =? ";
    std::cin >> a;
    if (a == 0) break; // stop loop in the middle
    s += a;
    std::cout << "sum = " << s << "\n";
}
```


Calculator with `break`

Version without break evaluates `a` twice and requires an additional block.

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


`continue`-Statement

```
continue;
```

- Jump over the rest of the body of the enclosing iteration statement
- Iteration statement is *not* left.

Calculator with `continue`

Ignore negative input:

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

Equivalence of Iteration Statements

We have seen:

- `while` and `do` can be simulated with `for`

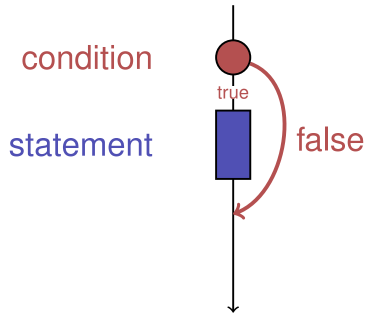
It even holds: Not so simple if a continue is used!

- The three iteration statements provide the same “expressiveness” (lecture notes)

Control Flow

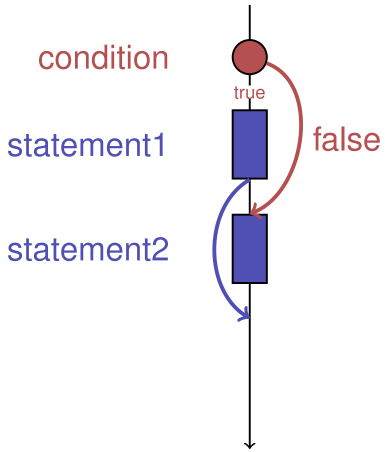
Order of the (repeated) execution of statements

- generally from top to bottom. . .
- . . . except in selection and iteration statements



`if` (*condition*)
    statement

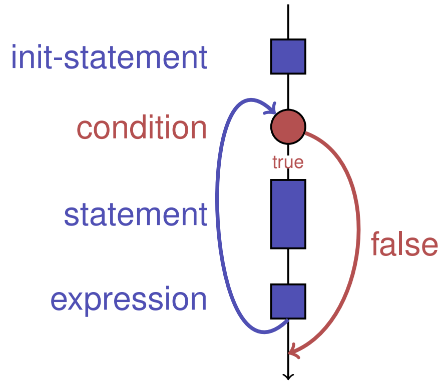
Control Flow `if else`



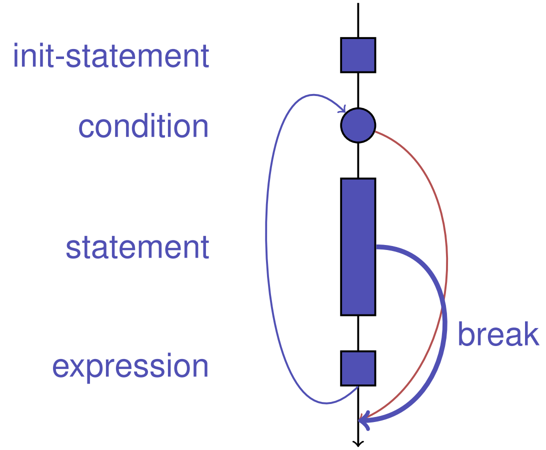
`if` (*condition*)
    statement1
else
    statement2

Control Flow `for`

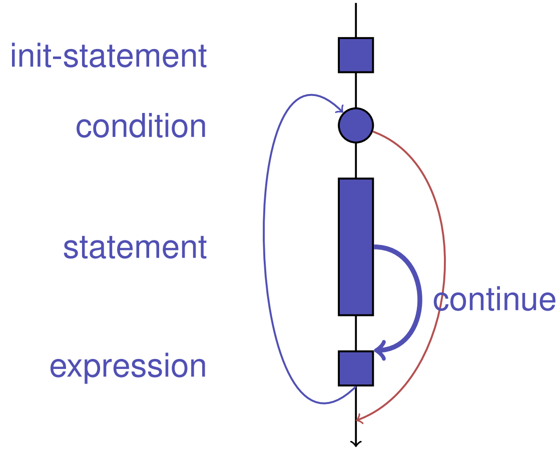
`for` (*init statement* *condition* `;` *expression*)
    statement



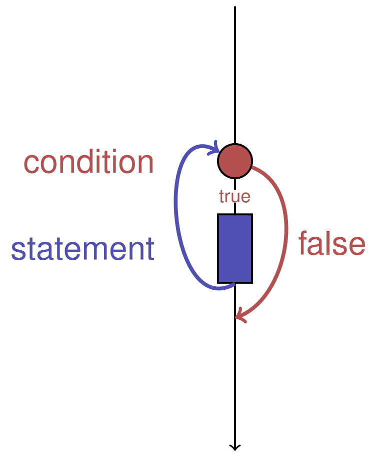
Control Flow `break` in for



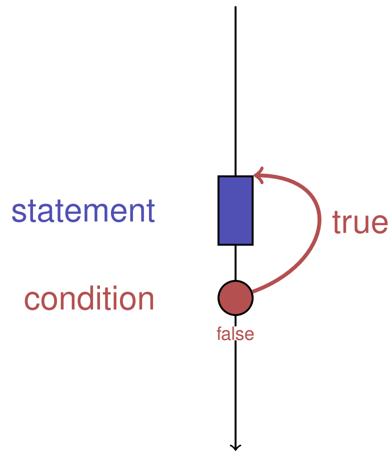
Control Flow `continue` in for



Control Flow `while`



Control Flow `do while`

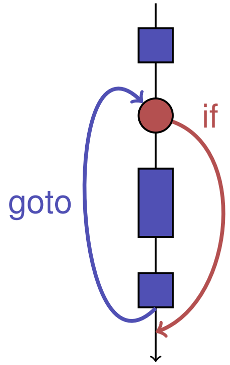


Control Flow: the Good old Times?

Observation
Actually, we only need `if` and jumps to arbitrary places in the program (`goto`).

Models:

- Machine Language
- Assembler (“higher” machine language)
- BASIC, the first prorgamming language for the general public (1964)



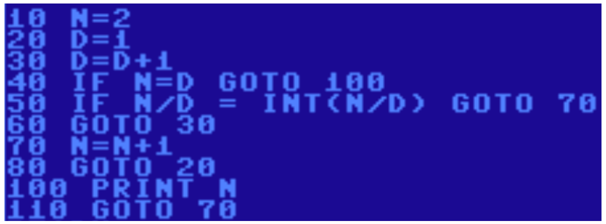
BASIC and home computers...

...allowed a whole generation of young adults to program.

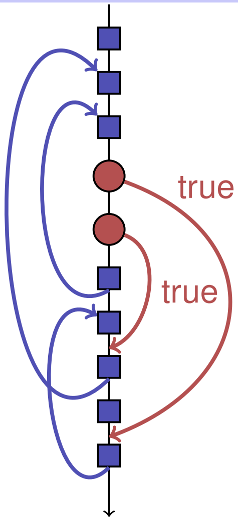
Home-Computer Commodore C64 (1982)

Spaghetti-Code with `goto`

Output of all prime numbers with BASIC



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



The “right” Iteration Statement

Goals: readability, conciseness, in particular

- few statements
- few lines of code
- simple control flow
- simple expressions

Often not all goals can be achieved simultaneously.

Odd Numbers in $\{0, \ldots, 100\}$

First (correct) attempt:

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


Odd Numbers in $\{0, \ldots, 100\}$

Less statements, *less* lines:

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


Odd Numbers in $\{0, \ldots, 100\}$

Less statements, *simpler* control flow:

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

This is the “right” iteration statement!

Jump Statements

- implement unconditional jumps.
- are useful, such as `while` and `do` but not indispensible
- should be used with care: only where the control flow is *simplified* instead of making it *more complicated*

The switch-Statement

```
switch (condition)
    statement
```

- *condition*: Expression, convertible to integral type
- *statement* : arbitrary statemet, in which `case` and `default`-lables are permitted, `break` has a special meaning.

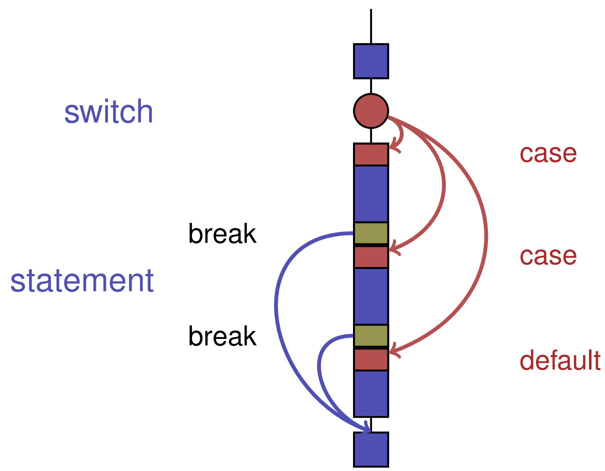
```
int Note;
...
switch (Note) {
    case 6:
        std::cout << "super!";
        break;
    case 5:
        std::cout << "cool!";
        break;
    case 4:
        std::cout << "ok.";
        break;
    default:
        std::cout << "hmm...";
}
```


Semantics of the switch-statement

```
switch (condition)
    statement
```

- `condition` is evaluated.
- If `statement` contains a `case`-label with (constant) value of `condition`, then jump there
- otherwise jump to the `default`-lable, if available. If not, jump over `statement`.
- The `break` statement ends the `switch`-statement.

Control Flow switch



Control Flow switch in general

If `break` is missing, continue with the next case.

7: ???
6: ok.
5: ok.
4: ok.
3: oops!
2: ooops!
1: oooops!
0: ???

```
switch (Note) {
    case 6:
    case 5:
    case 4:
        std::cout << "ok.";
        break;
    case 1:
        std::cout << "o";
    case 2:
        std::cout << "o";
    case 3:
        std::cout << "oops!";
        break;
    default:
        std::cout << "???";
}
```

6. Floating-point Numbers I

Types `float` and `double`; Mixed Expressions and Conversion; Holes in the Value Range

"Proper Calculation"

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

#include <iostream>

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

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


Fixed-point numbers

- fixed number of integer places (e.g. 7)
- fixed number of decimal places (e.g. 3)

`0.0824 = 0000000.082` ⟵ third place truncated

Disadvantages

- Value range is getting *even* smaller than for integers.
- Representability depends on the position of the decimal point.

Floating-point numbers

- fixed number of significant places (e.g. 10)
- plus position of the decimal point

`82.4` $= 824 \cdot 10^{-1}$

`0.0824` $= 824 \cdot 10^{-4}$

- Number is *Mantissa* $\times\ 10^{\textit{Exponent}}$

Types `float` and `double`

- are the fundamental C++ types for floating point numbers
- approximate the field of real numbers $(\mathbb{R}, +, \times)$ from mathematics
- have a big value range, sufficient for many applications (`double` provides more places than `float`)
- are fast on many computers

Arithmetic Operators

Like with `int`, but . . .

- Division operator `/` models a “proper” division (real-valued, not integer)
- No modulo operators such as `%` or `%=`

Literals

are different from integers by providing

- decimal point

1.23e-7f

integer part

exponent

fractional part

`1.0` : type `double`, value 1

`1.27f` : type `float`, value 1.27

- and / or exponent.

`1e3` : type `double`, value 1000

`1.23e-7` : type `double`, value $1.23 \cdot 10^{-7}$

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

Computing with `float`: Example

Approximating the Euler-Number

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

using the first 10 terms.

Computing with `float`: Euler Number

```
// Program: euler.cpp
// Approximate the Euler number e.

#include <iostream>

int main ()
{
  // values for term i, initialized for i = 0
  float t = 1.0f;   // 1/i!
  float e = 1.0f;   // i-th approximation of e

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

  return 0;
}
```


Computing with `float`: Euler Number

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


Mixed Expressions, Conversion

- Floating point numbers are more general than integers.
- In mixed expressions integers are converted to floating point numbers.

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


Value range

Integer Types:

- Over- and Underflow relatively frequent, but ...
- the value range is contiguous (no “holes”): $\mathbb{Z}$ is “discrete”.

Floating point types:

- Overflow and Underflow seldom, but ...
- there are holes: $\mathbb{R}$ is “continuous”.

Holes in the value range

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

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

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

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

input 1.1

input 1.0

input 0.1

output 2.23517e-8

What is going on here?

7. Floating-point Numbers II

Floating-point Number Systems; IEEE Standard; Limits of Floating-point Arithmetics; Floating-point Guidelines; Harmonic Numbers

Floating-point Number Systems

A Floating-point number system is defined by the four natural numbers:

- $\beta \geq 2$, the base,
- $p \geq 1$, the precision (number of places),
- $e_{\min}$, the smallest possible exponent,
- $e_{\max}$, the largest possible exponent.

Notation:

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

Floating-point number Systems

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

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

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

represented in base β:

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

Floating-point Number Systems

Example

- $\beta = 10$

Representations of the decimal number 0.1

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

Normalized representation

Normalized number:

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

Remark 1

The normalized representation is unique and therefore prefered.

Remark 2

The number 0 (and all numbers smaller than $\beta^{e_{\min}}$) have no normalized representation (we will deal with this later)!

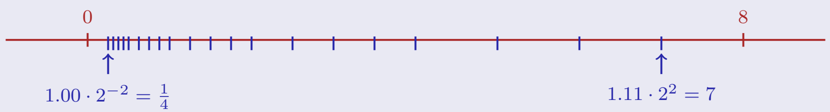
Set of Normalized Numbers

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

Normalized Representation

Example $F^*(2, 3, -2, 2)$ (only positive numbers)

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



Binary and Decimal Systems

- Internally the computer computes with $\beta = 2$ (binary system)
- Literals and inputs have $\beta = 10$ (decimal system)
- Inputs have to be converted!

Conversion Decimal $\to$ Binary

Assume, $0 < x < 2$.

Binary representation:

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

Conversion Decimal $\to$ Binary

Assume $0 < x < 2$.

- Hence: $x' = b_{-1}{\bullet}b_{-2}b_{-3}b_{-4}\ldots = 2 \cdot (x - b_0)$
- Step 1 (for x): Compute b_0:

$$
b_0 = \begin{cases} 1, & \text{if } x \geq 1 \\ 0, & \text{otherwise} \end{cases}
$$

- Step 2 (for x): Compute $b_{-1}, b_{-2}, \ldots$:

  Go to step 1 (for $x' = 2 \cdot (x - b_0)$)

Binary representation of 1.1

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_{-1} = \mathbf{0}$	0.2	0.4
0.4	$b_{-2} = \mathbf{0}$	0.4	0.8
0.8	$b_{-3} = \mathbf{0}$	0.8	1.6
1.6	$b_{-4} = \mathbf{1}$	0.6	1.2
1.2	$b_{-5} = \mathbf{1}$	0.2	0.4

$\Rightarrow 1.0\overline{0011}$, periodic, *not* finite

Binary Number Representations of 1.1 and 0.1

- are not finite, hence there are errors when converting into a (finite) binary floating-point system.
- `1.1f` and `0.1f` do not equal 1.1 and 0.1, but are slightly inaccurate approximation of these numbers.
- In `diff.cpp`: $1.1 - 1.0 \neq 0.1$

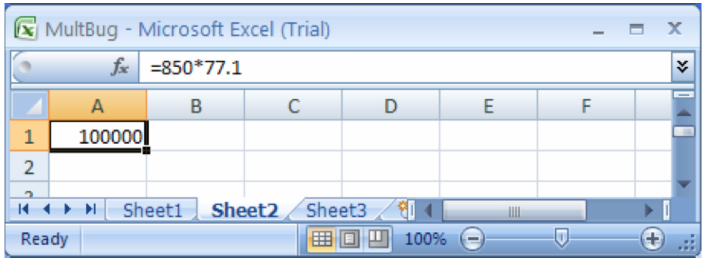
Binary Number Representations of 1.1 and 0.1

on my computer:

$$\begin{aligned} \texttt{1.1} &= \underline{1.1000000000000000}888178\ldots \\ \texttt{1.1f} &= \underline{1.1000000}238418\ldots \end{aligned}$$

The Excel-2007-Bug

```
std::cout << 850 * 77.1; // 65535
```



- 77.1 does not have a finite binary representation, we obtain $65534.9999999999927\ldots$
- For this and exactly 11 other "rare" numbers the output (and only the output) was wrong.

http://www.lomont.org/Math/Papers/2007/Excel2007/Excel2007Bug.pdf

Computing with Floating-point Numbers

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

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

1. adjust exponents by denormalizing one number 2. binary addition of the significands 3. renormalize 4. round to p significant places, if necessary

The IEEE Standard 754

- defines floating-point number systems and their rounding behavior
- is used nearly everywhere
- Single precision (`float`) numbers:

$F^*(2, 24, -126, 127)$ plus $0, \infty, \ldots$

- Double precision (`double`) numbers:

$F^*(2, 53, -1022, 1023)$ plus $0, \infty, \ldots$

- All arithmetic operations round the *exact* result to the next representable number

The IEEE Standard 754

Why

$$F^*(2, 24, -126, 127)?$$

- 1 sign bit
- 23 bit for the significand (leading bit is 1 and is not stored)
- 8 bit for the exponent (256 possible values)(254 possible exponents, 2 special values: 0, ∞,...)

$\Rightarrow$ 32 bit in total.

The IEEE Standard 754

Why

$$F^*(2, 53, -1022, 1023)?$$

- 1 sign bit
- 52 bit for the significand (leading bit is 1 and is not stored)
- 11 bit for the exponent (2046 possible exponents, 2 special values: 0, ∞,...)

$\Rightarrow$ 64 bit in total.

Floating-point Rules Rule 1

Rule 1

Do not test rounded floating-point numbers for equality.

```
for (float i = 0.1; i != 1.0; i += 0.1)
    std::cout << i << "\n";
```

endless loop because i never becomes exactly 1

Floating-point Rules — Rule 2

Rule 2

Do not add two numbers of very different orders of magnitude!

$$\begin{aligned} & 1.000 \cdot 2^5 \\ & +1.000 \cdot 2^0 \\ & = 1.00001 \cdot 2^5 \\ & \text{“=”}\ 1.000 \cdot 2^5 \quad \text{(Rounding on 4 places)} \end{aligned}$$

Addition of 1 does not have any effect!

Harmonic Numbers — Rule 2

- The n-the harmonic number is

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln n.$$

- This sum can be computed in forward or backward direction, which is mathematically clearly equivalent

Harmonic Numbers — Rule 2

```
// Program: harmonic.cpp
// Compute the n-th harmonic number in two ways.

#include <iostream>

int main()
{
  // Input
  std::cout << "Compute H_n for n =? ";
  unsigned int n;
  std::cin >> n;

  // Forward sum
  float fs = 0;
  for (unsigned int i = 1; i <= n; ++i)
    fs += 1.0f / i;

  // Backward sum
  float bs = 0;
  for (unsigned int i = n; i >= 1; --i)
    bs += 1.0f / i;

  // Output
  std::cout << "Forward sum  = " << fs << "\n"
            << "Backward sum = " << bs << "\n";
  return 0;
}
```


Harmonic Numbers — Rule 2

Results:

- ```
  Compute H_n for n =?  10000000
  Forward sum = 15.4037
  Backward sum = 16.686
  ```
- ```
  Compute H_n for n =?  100000000
  Forward sum = 15.4037
  Backward sum = 18.8079
  ```

Harmonic Numbers — Rule 2

Observation:

- The forward sum stops growing at some point and is "really" wrong.
- The backward sum approximates H_n well.

Explanation:

- For $1 + 1/2 + 1/3 + \cdots$, later terms are too small to actually contribute
- Problem similar to $2^5 + 1$ "=" 2^5

Floating-point Guidelines — Rule 3

Rule 4
Do not subtract two numbers with a very similar value.

Cancellation problems, cf. lecture notes.

Literature

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

Randy Glasbergen, 1996

8. Functions I

Defining and Calling Functions, Evaluation of Function Calls, the Type `void`, Pre- and Post-Conditions

Functions

- encapsulate functionality that is frequently used (e.g. computing powers) and make it easily accessible
- structure a program: partitioning into small sub-tasks, each of which is implemented as a function

⇒ Procedural programming; procedure: a different word for function.

Example: Computing Powers

```
double a;
int n;
std::cin >> a; // Eingabe a
std::cin >> n; // Eingabe n

double result = 1.0;
if (n < 0) { // a^n = (1/a)^(−n)
  a = 1.0/a;
  n = −n;
}
for (int i = 0; i < n; ++i)
  result ∗= a;
```

"Funktion pow"

```
std::cout << a << "^" << n << " = " << result pow(a,n) << ".\n";
```


Function to Compute Powers

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


Function to Compute Powers

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

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

int main()
{
  std::cout << pow( 2.0, −2) << "\n"; // outputs 0.25
  std::cout << pow( 1.5, 2) << "\n"; // outputs 2.25
  std::cout << pow(−2.0, 9) << "\n"; // outputs −512

  return 0;
}
```

Function Definitions

return type *argument types*

T fname (T_1 pname$_1$, T_2 pname$_2$, $\ldots$, T_N pname$_N$)
block

body

function name *formal arguments*

Defining Functions

- may not occur *locally*, i.e. not in blocks, not in other functions and not within control statements
- can be written consecutively without separator in a program

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

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


Example: Xor

```
// post: returns l XOR r
bool Xor(bool l, bool r)
{
    return l && !r || !l && r;
}
```


Example: Harmonic

```
// PRE: n >= 0
// POST: returns nth harmonic number
//       computed with backward sum
float Harmonic(int n)
{
    float res = 0;
    for (unsigned int i = n; i >= 1; --i)
        res += 1.0f / i;
    return res;
}
```

Example: min

```
// POST: returns the minimum of a and b
int min(int a, int b)
{
    if (a<b)
        return a;
    else
        return b;
}
```


Function Calls

fname (*expression*$_1$, *expression*$_2$, ..., *expression*$_N$)

- All call arguments must be convertible to the respective formal argument types.
- The function call is an expression of the return type of the function. Value and effect as given in the postcondition of the function *fname*.

Example: `pow(a,n)`: Expression of type `double`

Function Calls

For the types we know up to this point it holds that:

- Call arguments are R-values
- The function call is an R-value.

fname: R-value $\times$ R-value $\times \cdots \times$ R-value $\longrightarrow$ R-value

Evaluation of a Function Call

- Evaluation of the call arguments
- Initialization of the formal arguments with the resulting values
- Execution of the function body: formal arguments behave laike local variables
- Execution ends with
  `return` *expression*;

Return value yiels the value of the function call.

Example: Evaluation Function Call

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

...
pow (2.0, −2)
```

Call of pow

Return

Formal arguments

- Declarative region: function definition
- are *invisible* outside the function definition
- are allocated for each call of the function (automatic storage duration)
- modifications of their value do not have an effect to the values of the call arguments (call arguments are R-values)

Scope of Formal Arguments

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

```
int main(){
    double b = 2.0;
    int e = −2;
    double z = pow(b, e);

    std::cout << z; // 0.25
    std::cout << b; // 2
    std::cout << e; // −2
    return 0;
}
```

Not the formal arguments `b` and `e` of `pow` but the variables defined here locally in the body of `main`

The type `void`

- Fundamental type with empty value range
- Usage as a return type for functions that do *only* provide an effect

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

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

`void`-Functions

- do not require `return`.
- execution ends when the end of the function body is reached or if
- `return;` is reached
  or
- `return` *expression*`;` is reached.

Expression with type `void` (e.g. a call of a function with return type `void`

Pre- and Postconditions

- characterize (as complete as possible) what a function does
- document the function for users and programmers (we or other people)
- make programs more readable: we do not have to understand *how* the function works
- are ignored by the compiler
- Pre and postconditions render statements about the correctness of a program possible – provided they are correct.

Preconditions

precondition:

- what is required to hold when the function is called?
- defines the *domain* of the function

0^e is undefined for $e < 0$

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


Postconditions

postcondition:

- What is guaranteed to hold after the function call?
- Specifies *value* and *effect* of the function call.

Here only value, no effect.

```
// POST: return value is b^e
```

Pre- and Postconditions

- should be correct:
- *if* the precondition holds when the function is called *then* also the postcondition holds after the call.

Funktion **pow**: works for all numbers $b \neq 0$

Pre- and Postconditions

- We do not make a statement about what happens if the precondition does not hold.
- $\mathrm{C}{+}{+}$-standard-slang: „Undefined behavior".

Function **pow**: division by 0

Pre- and Postconditions

- pre-condition should be as *weak* as possible (largest possible domain)
- post-condition should be as *strong* as possible (most detailed information)

White Lies. . .

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

is formally incorrect:

- Overflow if e or b are too large
- b^e potentially not representable as a double (holes in the value range!)

White Lies are Allowed

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

The exact pre- and postconditions are platform-dependent and often complicated. We abstract away and provide the mathematical conditions. ⇒ compromise between formal correctness and lax practice.

Checking Preconditions...

- Preconditions are only comments.
- How can we ensure that they hold when the function is called?

...with assertions

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


Postconditions with Asserts

- The result of “complex” computations is often easy to check.
- Then the use of asserts for the postcondition is worthwhile.

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

Exceptions

- Assertions are a rough tool; if an assertions fails, the program is halted in a unrecoverable way.
- C++provides more elegant means (exceptions) in order to deal with such failures depending on the situation and potentially without halting the program
- Failsafe programs should only halt in emergency situations and therefore should work with exceptions. For this course, however, this goes too far.

9. Functions II

Stepwise Refinement, Scope, Libraries and Standard Functions

Stepwise Refinement

- A simple *technique* to solve complex problems

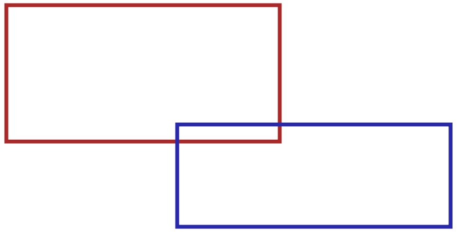
[3] Niklaus Wirth. Program development by stepwise refinement. Commun. ACM 14, 4, 1971

Stepwise Refinement

- Solve the problem step by step. Start with a coarse solution on a high level of abstraction (only comments and abstract function calls)
- At each step, comments are replaced by program text, and functions are implemented (using the same principle again)
- The refinement also refers to the development of data representation (more about this later).
- If the refinement is realized as far as possible by functions, then partial solutions emerge that might be used for other problems.
- Stepwise refinement supports (but does not replace) the structural understanding of a problem.

Example Problem

Find out if two rectangles intersect!



Coarse Solution

(include directives omitted)

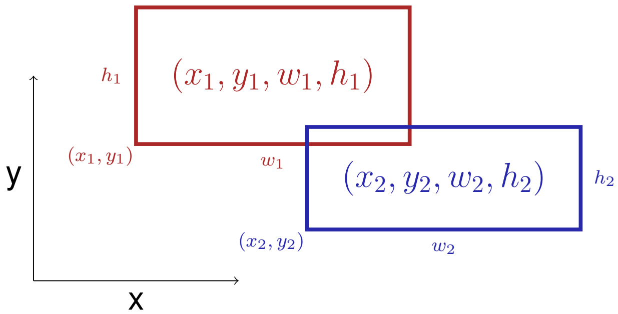
```
int main()
{
    // input rectangles

    // intersection?

    // output solution

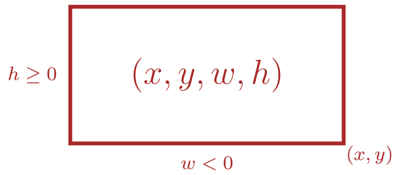
    return 0;
}
```


Refinement 1: Input Rectangles



Refinement 1: Input Rectangles

Width w and height h may be negative.



Refinement 1: Input Rectangles

```
int main()
{
    std::cout << "Enter two rectangles [x y w h each] \n";
    int x1, y1, w1, h1;
    std::cin >> x1 >> y1 >> w1 >> h1;
    int x2, y2, w2, h2;
    std::cin >> x2 >> y2 >> w2 >> h2;

    // intersection?

    // output solution

    return 0;
}
```


Refinement 2: Intersection? and Output

```
int main()
{
    input rectangles ✓

    bool clash = rectangles_intersect (x1,y1,w1,h1,x2,y2,w2,h2);

    if (clash)
        std::cout << "intersection!\n";
    else
        std::cout << "no intersection!\n";

    return 0;
}
```


Refinement 3: Intersection Function. . .

```
bool rectangles_intersect (int x1, int y1, int w1, int h1,
                           int x2, int y2, int w2, int h2)
{
    return false; // todo
}

int main() {
    input rectangles ✓

    intersection? ✓

    output solution ✓

    return 0;
}
```


Refinement 3: Intersection Function. . .

```
bool rectangles_intersect (int x1, int y1, int w1, int h1,
                           int x2, int y2, int w2, int h2)
{
    return false; // todo
}
```

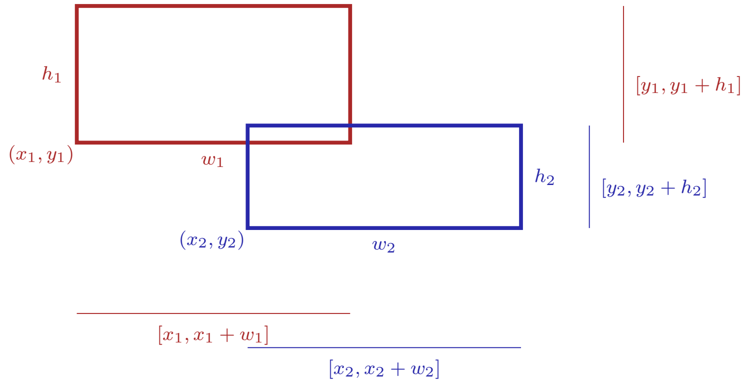
Function main ✓

Refinement 3: ...with PRE and POST

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


Refinement 4: Interval Intersection

Two rectangles intersect if and only if their x and y-intervals intersect.



Refinement 4: Interval Intersections

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


Refinement 4: Interval Intersections

```
// PRE: [a1, b1], [a2, b2] are (generalized) intervals,
//     with [a,b] := [b,a] if a>b
// POST: returns true if [a1, b1],[a2, b2] intersect
bool intervals_intersect (int a1, int b1, int a2, int b2)
{
    return false; // todo
}
```

Function rectangles_intersect ✓

Function main ✓

Refinement 5: Min and Max

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


Refinement 5: Min and Max

```
// POST: the maximum of x and y is returned
int max (int x, int y){
    if (x>y) return x; else return y;
}

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

already exists in the standard library

`Function intervals_intersect` ✓

`Function rectangles_intersect` ✓

`Function main` ✓

Back to Intervals

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


Look what we have achieved step by step!

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

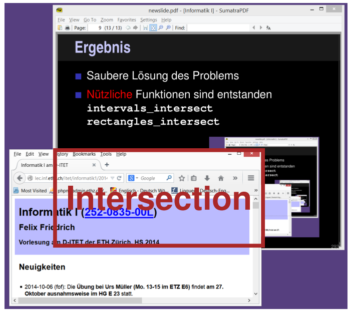
// PRE: [a1, b1], [a2, h2] are (generalized) intervals,
//     with [a,b] := [b,a] if a>b
// POST: returns true if [a1, b1],[a2, b2] intersect
bool intervals_intersect (int a1, int b1, int a2, int b2)
{
  return std::max(a1, b1) >= std::min(a2, b2)
      && std::min(a1, b1) <= std::max(a2, b2);
}

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

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

Result

- Clean solution of the problem
- Useful functions have been implemented
  `intervals_intersect`
  `rectangles_intersect`



Where can a Function be Used?

```
#include<iostream>

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

int f (int i) // Scope of f starts here
{
    return i;
}
```

Gültigkeit f

Scope of a Function

- is the part of the program where a function can be called
- is defined as the union of all scopes of its declarations (there can be more than one)

declaration of a function: like the definition but without `{...}`.

```
double pow (double b, int e);
```


This does not work...

```
#include<iostream>

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

int f (int i) // Scope of f starts here
{
    return i;
}
```

Gültigkeit f

. . . but this works!

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

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

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


Forward Declarations, why?

Functions that mutually call each other:

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

int f (...) // f valid from here
{
    g(...) // ok
}

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

Gültigkeit g

Gültigkeit f

Reusability

- Functions such as **`rectanges`** and **`pow`** are useful in many programs.
- "Solution": copy-and-paste the source code
- Main disadvantage: when the function definition needs to be adapted, we have to change *all* programs that make use of the function

Level 1: Outsource the Function

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

Level 1: Include the Function

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

#include <iostream>
#include "math.cpp"
```

file in working directory

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

  return 0;
}
```


Disadvantage of Including

- `#include` copies the file (`math.cpp`) into the main program (`callpow2.cpp`).
- The compiler has to (re)compile the function definition for each program
- This can take long for many and large functions.

Level 2: Separate Compilation

of `math.cpp` independent of the main program:

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

math.cpp

g++ -c math.cpp

```
00111010110010101 0
00010111010100011 1
00010 Funktion pow
11110000110101000 1
1111111 01000111010
01010110101101000 1
10010111110010101 0
```

math.o

Level 2: Separate Compilation

Declaration of all used symbols in so-called *header* file.

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

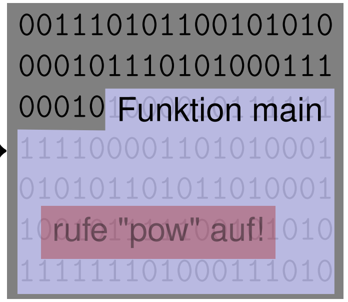
math.h

Level 2: Separate Compilation

of the main program, independent of `math.cpp`, if a *declaration* of `math` is included.

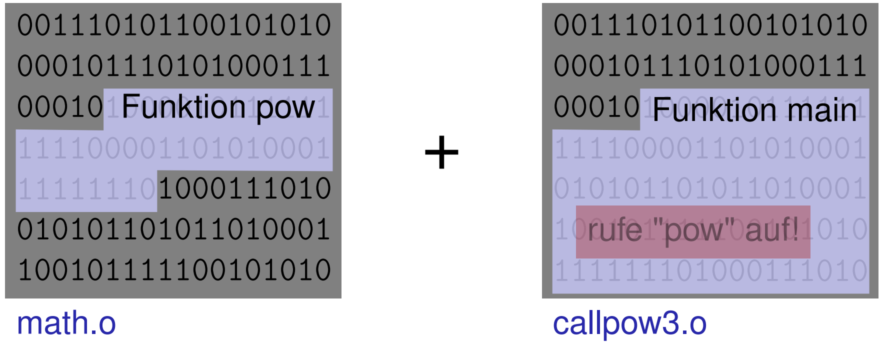
```
#include <iostream>
#include "math.h"
int main()
{
  std::cout << pow(2,-2) << "\n";
  return 0;
}
```

callpow3.cpp

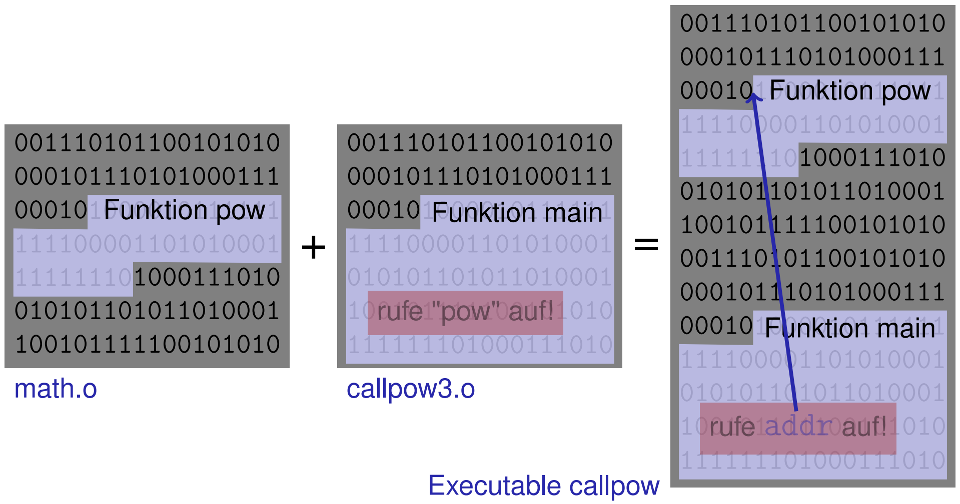


callpow3.o

The linker unites...



... what belongs together



Availability of Source Code?

Observation

`math.cpp` (source code) is not required any more when the `math.o` (object code) is available.

Many vendors of libraries do not provide source code.

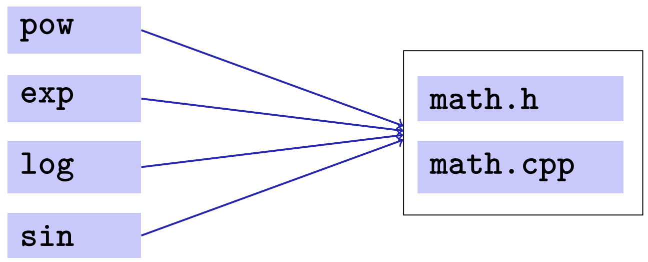
Header files then provide the *only* readable informations.

„Open Source” Software

- Source code is generally available.
- Only this allows the continued development of code by users and dedicated “hackers”.
- Even in commercial domains, “open source” gains ground.
- Certain licenses force naming sources and open development. Example GPL (GNU Genereal Public License)
- Known open source software: Linux (operating system), Firefox (browser), Thunderbird (email program)...

Libraries

- Logical grouping of similar functions



Name Spaces…

```
// ifeemath.h
// A small library of mathematical functions
namespace ifee {

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

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


…Avoid Name Conflicts

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

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

Name Spaces / Compilation Units

In C++ the concept of separate compilation is *independent* of the concept of name spaces

In some other languages,e.g. Modula / Oberon (partially also for Java) the compilation unit can define a name space.

Functions from the Standard Library

- help to avoid re-inventing the wheel (such as with `ifee::pow`);
- lead to interesting and efficient programs in a simple way;
- guarantee a quality standard that cannot easily be achieved with code written from scratch.

Prime Number Test with `sqrt`

$n \geq 2$ is a prime number if and only if there is no d in $\{2, \ldots, n-1\}$ dividing n .

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


Prime Number test with `sqrt`

$n \geq 2$ is a prime number if and only if there is no d in $\{2, \ldots, \lfloor\sqrt{n}\rfloor\}$ dividing n .

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

- This works because `std::sqrt` rounds to the next representable `double` number (IEEE Standard 754).
- Other mathematical functions (`std::pow,...`) are almost as exact in practice.

Prime Number test with `sqrt`

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

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

  // Computation: test possible divisors d up to sqrt(n)
  unsigned int bound = std::sqrt(n);
  unsigned int d;
  for (d = 2; d <= bound && n % d != 0; ++d);@

  // Output
  if (d <= bound)
    // d is a divisor of n in {2,...,[sqrt(n)]}
    std::cout << n << " = " << d << " * " << n / d << ".\n";
  else
    // no proper divisor found
    std::cout << n << " is prime.\n";

  return 0;
}
```


Functions Should be More Capable! Swap ?

```
void swap (int x, int y) {
 int t = x;
 x = y;
 y = t;
}
int main(){
    int a = 2;
    int b = 1;
    swap (a, b);
    assert (a==1 && b==2); // fail!
}
```


Functions Should be More Capable! Swap ?

```
// POST: values of x and y are exchanged
void swap (int& x, int& y) {
 int t = x;
 x = y;
 y = t;
}
int main(){
    int a = 2;
    int b = 1;
    swap (a, b);
    assert (a==1 && b==2); // ok!
}
```


Sneak Preview: Reference Types

- We can enable functions to change the value of call arguments.
- Not a new concept for functions but rather a new class of types

Re

10. Reference Types

Reference Types: Definition and Initialization, Call By Value, Call by Reference, Temporary Objects, Constants, Const-References

Swap!

```
// POST: values of x and y are exchanged
void swap (int& x, int& y) {
 int t = x;
 x = y;
 y = t;
}
int main(){
    int a = 2;
    int b = 1;
    swap (a, b);
    assert (a == 1 && b == 2); // ok!
}
```


Reference Types

- We can make functions change the values of the call arguments
- no new concept for functions, but a new class of types

Reference Types

Reference Types: Definition

T& read as „*T*-reference”

underlying type

- *T&* has the same range of values and functionality as *T*, ...
- but initialization and assignment work differently.

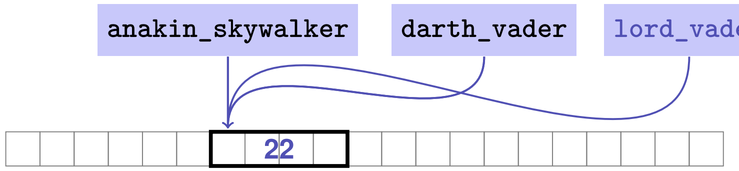
Anakin Skywalker alias Darth Vader

Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;
int& darth_vader = anakin_skywalker; //  alias
int& lord_vader = darth_vader; // another alias
darth_vader = 22;
```

assignment to the L-value behind the alias

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



Reference Types: Intialization and Assignment

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

- A variable of reference type (a *reference*) can only be initialized with an L-Value .
- The variable is becoming an *alias* of the L-value (a different name for the referenced object).
- Assignment to the reference is to the object behind the alias.

Reference Types: Implementation

Internally, a value of type *T&* is represented by the address of an object of type *T*.

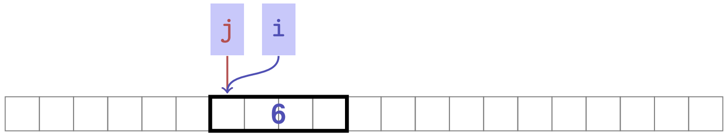
```
int& j;  // Error: j must be an alias of something

int& k = 5;  // Error: the literal 5 has no address
```

Call by Reference

Reference types make it possible that functions modify the value of the call arguments:

```
void increment (int& i)   ← initialization of the formal arguments
{ // i becomes an alias of the call argument
    ++i;
}
...
int j = 5;
increment (j);
std::cout << j << "\n"; // 6
```



Call by Reference

Formal argument has reference type:

⇒ **Call by Reference**

Formal argument is (internally) initialized with the *address* of the call argument (L-value) and thus becomes an *alias*.

Call by Value

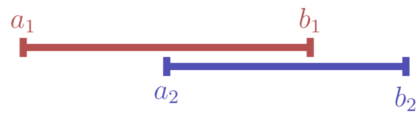
Formal argument does not have a reference type:

⇒ **Call by Value**

Formal argument is initialized with the *value* of the actual parameter (R-Value) and thus becomes a *copy*.

In Context: Assignment to References

```
// PRE:  [a1, b1], [a2, b2] are (generalized) intervals,
// POST: returns true if [a1, b1], [a2, b2] intersect, in which case
//       [l, h] contains the intersection of [a1, b1], [a2, b2]
bool intervals_intersect (int& l, int& h,
                          int a1, int b1, int a2, int b2) {
  sort (a1, b1);
  sort (a2, b2);
  l = std::max (a1, a2);
  h = std::min (b1, b2);
  return l <= h;
}
...
int lo = 0; int hi = 0;
if (intervals_intersect (lo, hi, 0, 2, 1, 3))
    std::cout << "[" << lo << "," << hi << "]" << "\n"; // [1,2]
```



In Context: Initialization of References

```
// POST: a <= b
void sort (int& a, int& b) {
   if (a > b)
     std::swap (a, b); // 'passing through' of references a,b
}

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


Return by Value / Reference

- Even the return type of a function can be a reference type (return by reference)
- In this case the function call itself is an L-value

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

exactly the semantics of the pre-increment

Temporary Objects

What is wrong here?

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

Return value of type `int&` becomes an alias of the formal argument. But the memory lifetime of i ends after the call!

```
int k = 3;
int& j = foo (k); // j is an alias of a zombie
std::cout << j << "\n";  // undefined behavior
```


The Reference Guidline

Reference Guideline

When a reference is created, the object referred to must “stay alive” at least as long as the reference.

The Compiler as Your Friend: Constants

Constants

- are variables with immutable value

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

- Usage: `const` before the definition

The Compiler as Your Friend: Constants

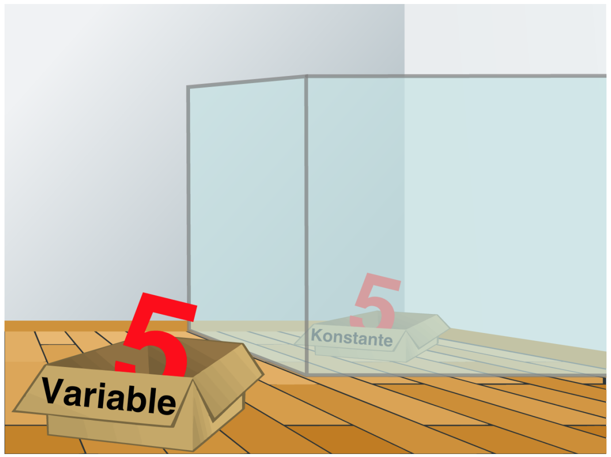
- Compiler checks that the `const`-promise is kept

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

compiler: error

- Tool to avoid errors: constants guarantee the promise :*“value does not change”*

Constants: Variables behind Glass



The `const`-guideline

`const`-guideline

For *each variable*, think about whether it will change its value in the lifetime of a program. If not, use the keyword `const` in order to make the variable a constant.

A program that adheres to this guideline is called `const`-correct.

Const-References

- have type `const` T `&` (= `const` (T `&`))
- can be initialized with R-Values (compiler generates a temporary object with sufficient lifetime)

`const T& r =` *lvalue*;

r is initialized with the address of *lvalue* (efficient)

`const T& r =` *rvalue*;

r is initialized with the address of a temporary object with the value of the *rvalue* (flexible)

What exactly does Constant Mean?

Consider an L-value with type `const` T

- Case 1: T is no reference type

Then the L-value is a constant.

```
const int n = 5;
int& i = n; // error: const-qualification is discarded
i = 6;
```

The compiler detects our attempt to cheat

What exactly does Constant Mean?

Consider L-value of type `const` T

- Case 2: T is reference type.

Then the L-value is a read-only alias which cannot be used to change the value

```
int n = 5;
const int& i = n;// i: read-only alias of n
int& j = n;      // j: read-write alias
i = 6;           // Error: i is a read-only alias
j = 6;           // ok: n takes on value 6
```


When `const T&` ?

Rule

Argument type `const` T `&` (call by *read-only* reference) is used for efficiency reasons instead of T (call by value), if the type T requires large memory. For fundamental types (`int`, `double`,...) it does not pay off.

Examples will follow later in the course

11. Arrays I

Array Types, Sieve of Erathostenes, Memory Layout, Iteration, Vectors, Characters and Texts, ASCII, UTF-8, Caesar-Code

Array: Motivation

- Now we can iterate over numbers

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

- Often we have to iterate over *data*. (Example: find a cinema in Zurich that shows "C++ Runner 2049" today)
- Arrays allow to store *homogeneous* data (example: schedules of all cinemas in Zurich)

Arrays: a first Application

The Sieve of Erathostenes

- computes all prime numbers $< n$
- method: cross out all non-prime numbers

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

at the end of the crossing out process, only prime numbers remain.

- Question: how do we cross out numbers ??
- Answer: with an *array*.

Sieve of Erathostenes: Initialization

```
const unsigned int n = 1000;
bool crossed_out[n];
for (unsigned int i = 0; i < n; ++i)
    crossed_out[i] = false;
```

constant!

`crossed_out[i]` indicates if i has been crossed out.

Sieve of Eratosthenes: Computation

```
for (unsigned int i = 2; i < n; ++i)
    if (!crossed_out[i] ){
      // i is prime
      std::cout << i << " ";
      // cross out all proper multiples of i
      for (unsigned int m = 2*i; m < n; m += i)
        crossed_out[m] = true;
    }
}
```

The sieve: go to the next non-crossed out number i (this must be a prime number), output the number and cross out all proper multiples of i

Arrays: Definition

Declaration of an array variable:

T a [expr]

base type

variable of array type

constant integer expression value provides the length of the array

value is known at compile time. e.g. literal, constant

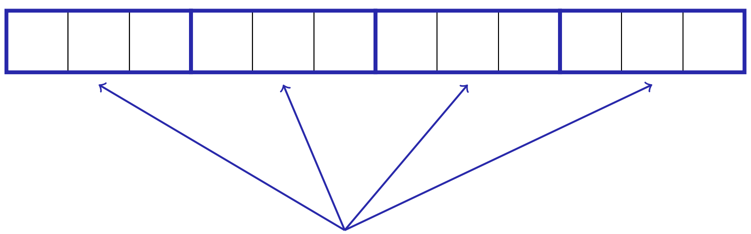
type of `a`*: „T[k]”*
values range of `a`*:* T^k

Beispiel: `bool crossed_out[n]`

Memory Layout of an Array

- An array occupies a *contiguous* memory area

example: an array with 4 elements



memory cells for a value of type `T` each

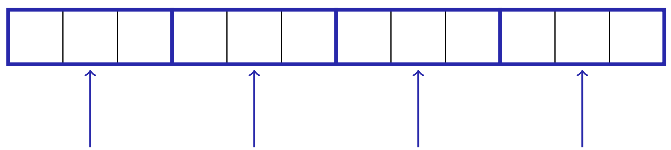
Random Access

The L-value

value i

a [expr]

has type *T* and refers to the i-th element of the array *a* (counting from 0!)



a[0] a[1] a[2] a[3]

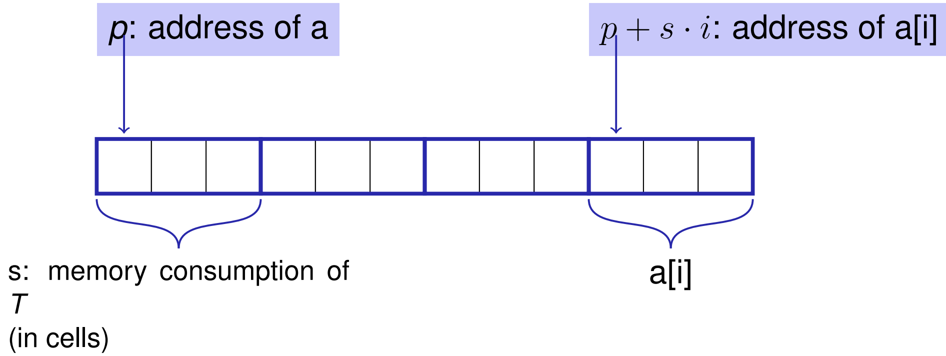
Random Access

a [*expr*]

The value i of *expr* is called *array index*.
`[]`: subscript operator

Random Access

- Random access is very efficient:



Array Initialization

- `int a[5];`

  The five elements of `a` remain uninitialized (values can be assigned later)

- `int a[5] = {4, 3, 5, 2, 1};`

  the 5 elements of `a` are initialized with an *initialization list*.

- `int a[] = {4, 3, 5, 2, 1};`

also ok: the compiler will deduce the length

Arrays are Primitive

- Accessing elements outside the valid bounds of the array leads to undefined behavior.

```
int arr[10];
for (int i=0; i<=10; ++i)
  arr[i] = 30; // runtime error: access to arr[10]!
```

Arrays are Primitive

Array Bound Checks

With no special compiler or runtime support it is the sole *responsibility of the programmer* to check the validity of element accesses.

Arrays are Primitive (II)

- Arrays cannot be initialized and assigned to like other types

```
int a[5] = {4,3,5,2,1};
int b[5];
b = a;            // Compiler error!
int c[5] = a;     // Compiler error!
```

Why?

Arrays are Primitive

- Arrays are legacy from the language C and primitive from a modern viewpoint
- In C, arrays are very low level and efficient, but do not offer any luxury such as initialization or copying.
- Missing array bound checks have far reaching consequences. Code with non-permitted but possible index accesses has been exploited (far too) often for malware.
- the standard library offers comfortable alternatives

Vectors

- Obvious disadvantage of static arrays: *constant array length*

```
const unsigned int n = 1000;
bool crossed_out[n];
```

- remedy: use the type `Vector` from the standard library

```
#include <vector>
...
std::vector<bool> crossed_out (n, false);
```

Initialization with n elements initial value `false`.

element type in triangular brackets

Sieve of Erathostenes with Vectors

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

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

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


Characters and Texts

- We have seen texts before:

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

  String-Literal

- can we really work with texts? Yes:

Character:	Value of the fundamental type `char`
Text:	Array with base type `char`

The type `char` ("character")

- represents printable characters (e.g. `'a'`) and *control characters* (e.g. `'\n'`)

`char c = 'a'`

defines variable c of type char with value 'a'

literal of type `char`

The type `char` ("character")

is formally an integer type

- values convertible to `int` / `unsigned int`
- all arithmetic operators are available (with dubious use: what is `'a'/'b'` ?)
- values typically occupy 8 Bit

domain:
$\{-128, \ldots, 127\}$ or $\{0, \ldots, 255\}$

The ASCII-Code

- defines concrete conversion rules
  `char` $\longrightarrow$ `int / unsigned int`
- is supported on nearly all platforms

Zeichen $\longrightarrow \{0, \ldots, 127\}$
`'A', 'B', ... , 'Z'` $\longrightarrow 65, 66, ..., 90$
`'a', 'b', ... , 'z'` $\longrightarrow 97, 98, ..., 122$
`'0', '1', ... , '9'` $\longrightarrow 48, 49, ..., 57$

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

Extension of ASCII: UTF-8

- Internationalization of Software $\Rightarrow$ large character sets required. Common today: unicode, 100 symbol sets, 110000 characters.
- ASCII can be encoded with 7 bits. An eighth bit can be used to indicate the appearance of further bits.

Bits	Encoding
7	`0xxxxxxx`
11	`110xxxxx 10xxxxxx`
16	`1110xxxx 10xxxxxx 10xxxxxx`
21	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`
26	`111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`
31	`1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx`

Interesting property: for each byte you can decide if a new UTF8 character begins.

Einige Zeichen in UTF-8

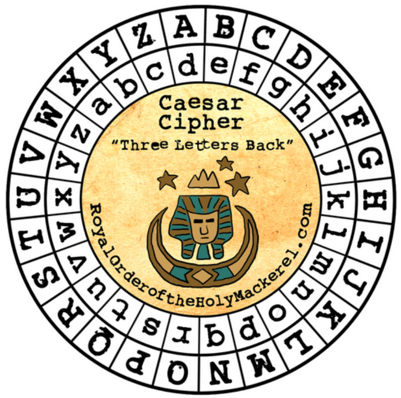
Symbol	Codierung (jeweils 16 Bit)
☠	`11100010 10011000 10100000`
☃	`11100010 10011000 10000011`
⍨	`11100010 10001101 10101000`
☙	`11100010 10011000 10011001`
〠	`11100011 10000000 10100000`
ﯹ	`11101111 10101111 10111001`

http://t-a-w.blogspot.ch/2008/12/funny-characters-in-unicode.html

Caesar-Code

Replace every printable character in a text by its pre-pre-predecessor.

' '	(32)	$\rightarrow$	'|'	(124)
'!'	(33)	$\rightarrow$	'}'	(125)
		...		
'D'	(68)	$\rightarrow$	'A'	(65)
'E'	(69)	$\rightarrow$	'B'	(66)
		...		
$\sim$	(126)	$\rightarrow$	'{'	(123)


  ```

Caesar-Code: Main Program

```
// Program: caesar_encrypt.cpp
// encrypts a text by applying a cyclic shift of −3

#include<iostream>
#include<cassert>
#include<ios> // for std::noskipws

// PRE: s < 95 && s > −95
// POST: if c is one of the 95 printable ASCII characters, c is
//       cyclically shifted s printable characters to the right
void shift (char& c, int s);
```

spaces and newline characters shall *not* be ignored

Caesar-Code: Main Program

```
int main ()
{
  std::cin >> std::noskipws; // don't skip whitespaces!

  // encryption loop
  char next;
  while (std::cin >> next) {
    shift (next, −3);
    std::cout << next;
  }
  return 0;
}
```

Conversion to `bool`: returns *false* if and only if the input is empty.

shifts only printable characters.

Caesar-Code: shift-Function

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

Call by reference!

Overflow – 95 backwards!

underflow – 95 forward!

normal shift

./caesar_encrypt < power8.cpp

```
,,|Moldo^j7|mltbo5+`mm
,,|O^fpb|^|krj_bo|ql|qeb|bfdeqe|mltbo+

 fk`irab|9flpqob^j;|

fkq|j^fk%&
x
||,,|fkmrq
||pqa77`lrq|99|~@ljmrqb|^[5|clo|^|:<|~8||
||fkq|^8
||pqa77`fk|;;|^8

||,,|`ljmrq^qflk
||fkq|_|:|^|'|^8|,,|_|:|^[/
||_|:|_|'|_8|||||,,|_|:|^[1

||,,|lrqmrq|_|'|_)|f+b+)|^[5
||pqa77`lrq|99|^|99|~[5|:|~|99|_|'|_|99|~+Yk~8
||obqrok|-8
z
```

Program = Moldo^j

Caesar-Code: Decryption

```
// decryption loop
char next;
while (std::cin >> next) {
  shift (next, 3);
  std::cout << next;
}
```

Now: shift by 3 to *right*

An interesting way to output `power8.cpp`

- `./caesar_encrypt < power8.cpp | ./caeser_decrypt`

12. Arrays II

Strings, Lindenmayer Systems, Multidimensional Arrays, Vectors of Vectors, Shortest Paths, Arrays and Vectors as Function Arguments

Strings as Arrays

- can be represented with underlying type `char`

```
char text[] = {'b','o','o','l'}
```

- can also be defined as string-literals

```
char text[] = "bool"
```

- can only be defined with constant size

Texts

- can be represented with the type `std::string` from the standard library.
-
```
std::string text = "bool";
```

defines a string with length 4

- A string is conceptually an array with base type `char`, plus additional functionality
- Requires `#include <string>`

Strings: pimped `char`-Arrays

A `std::string`...

- knows its length

```
text.length()
```

returns its length as `int` (call of a member function; will be explained later

- can be initialized with variable length

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

`text` is filled with n `'a'`s

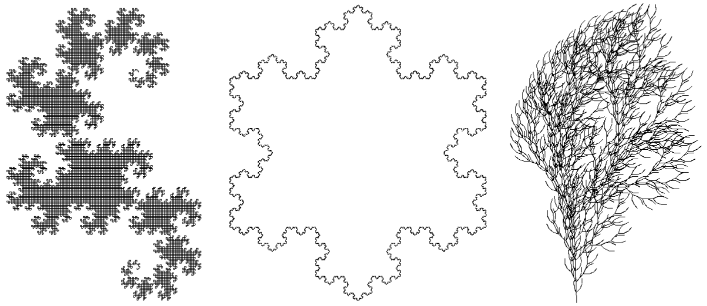
- “understands” comparisons

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

`true` if `text1` and `text2` match

Lindenmayer-Systems (L-Systems)

Fractals made from Strings and Turtles



L-Systems have been invented by the Hungarian biologist Aristid Lindenmayer (1925 – 1989) to model the growth of plants.

Definition and Example

- Alphabet Σ
- Σ^*: all finite words over Σ
- Production $P : \Sigma \rightarrow \Sigma^*$
- Initial word $s_0 \in \Sigma^*$

- $\{\mathrm{F}, +, -\}$
c	$P(c)$
F	F + F +
+	+
−	−
- F

Definition

The triple $\mathcal{L} = (\Sigma, P, s_0)$ is an L-System.

The Described Language

Words $w_0, w_1, w_2, \ldots \in \Sigma^*$: $\qquad P(\mathrm{F}) = \mathrm{F} + \mathrm{F} +$

$$\begin{aligned} w_0 &:= s_0 \\ w_1 &:= P(w_0) \\ w_2 &:= P(w_1) \\ &\vdots \end{aligned}$$

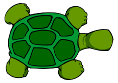
$$\begin{aligned} w_0 &:= \mathrm{F} \\ & \quad \mathrm{F} + \mathrm{F} + \\ w_1 &:= \boxed{\mathrm{F}}\boxed{+}\boxed{\mathrm{F}}\boxed{+} \\ w_2 &:= \boxed{\mathrm{F} + \mathrm{F} +}\boxed{+}\boxed{\mathrm{F} + \mathrm{F} +}\boxed{+} \\ & \quad P(\mathrm{F})P(+)P(\mathrm{F})P(+) \\ &\vdots \end{aligned}$$

Definition

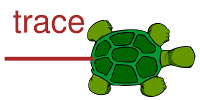
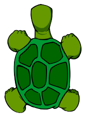
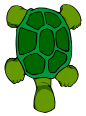
$P(c_1 c_2 \ldots c_n) := P(c_1)P(c_2) \ldots P(c_n)$

Turtle-Graphics

Turtle with position and direction.

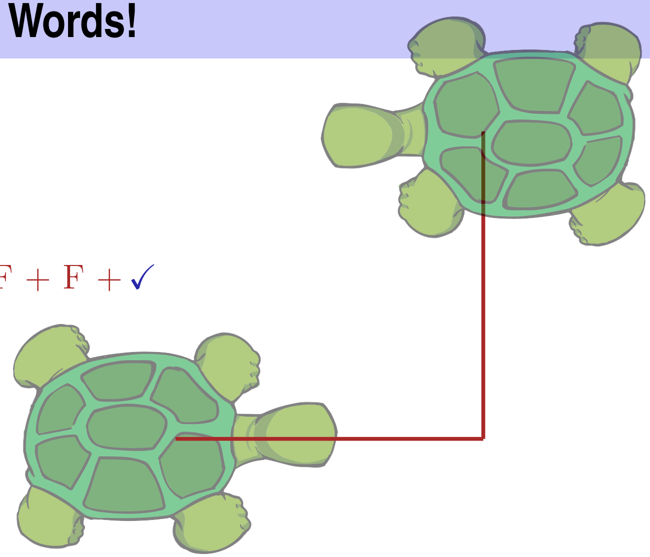


Turtle understands 3 commands:

F: one step forward ✓ 	$+$: turn by 90 degrees ✓ 	$-$: turn by -90 degrees ✓ 

Draw Words!

$w_1 = \mathrm{F} + \mathrm{F} + ✓$



`lindenmayer.cpp`: Main Program

Words $w_0, w_1, w_2, \ldots w_n \in \Sigma^*$: `std::string`

```
...
#include "turtle.h"
...
std::cout << "Number of iterations =? ";
unsigned int n;
std::cin >> n;

std::string w = "F";

for (unsigned int i = 0; i < n; ++i)
  w = next_word (w);

draw_word (w);
```

$w = w_0 = \mathrm{F}$

$w = w_i \to w = w_{i+1}$

draw $w = w_n$!

`lindenmayer.cpp`: `next_word`

```
// POST: replaces all symbols in word according to their
//       production and returns the result
std::string next_word (std::string word) {
  std::string next;
  for (unsigned int k = 0; k < word.length(); ++k)
    next += production (word[k]);
  return next;
}

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

`lindenmayer.cpp:` `draw_word`

```
// POST: draws the turtle graphic interpretation of word
void draw_word (std::string word)
{
  for (unsigned int k = 0; k < word.length(); ++k)
    switch (word[k]) {
    case 'F':
      turtle::forward();
     break;
    case '+':
      turtle::left(90);
      break;
    case '−':
      turtle::right(90);
    }
}
```

jump to the `case` that corresponds to `word[k]`.

forward! (function from our turtle library)

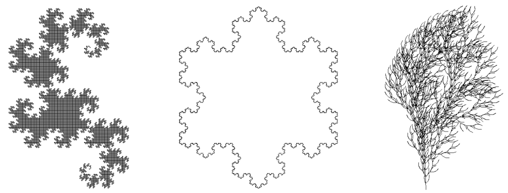
skip the remaining `cases`

turn by 90 degrees! (function from our turtle library)

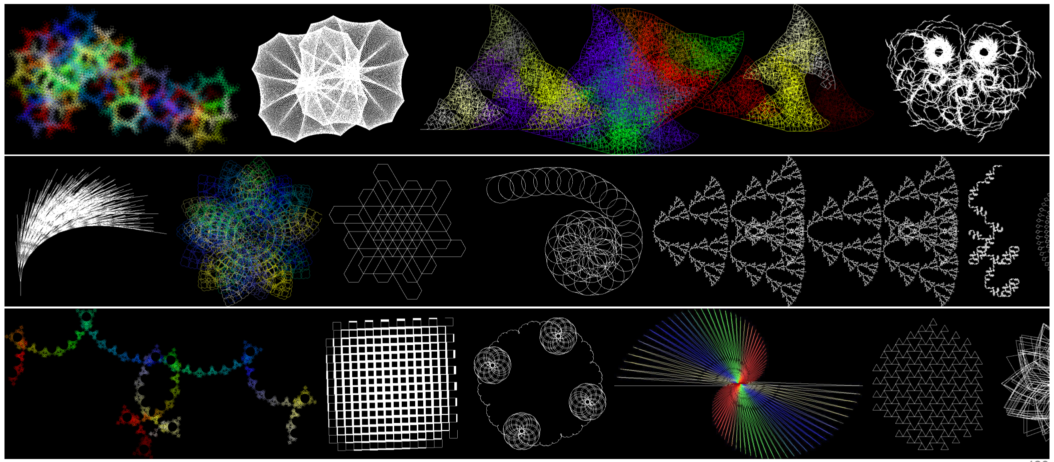
turn by -90 degrees (function from our turtle library)

L-Systems: Extensions

- Additional symbols without graphical interpretation (`dragon.cpp`)
- Arbitrary angles (`snowflake.cpp`)
- Saving and restoring the turtle state → plants (`bush.cpp`)



L-System-Challenge: `amazing.cpp!`



Multidimensional Arrays

- are arrays of arrays
- can be used to store *tables, matrices, ...*

```
int a[2][3]
```

`a` contains two elements and each of them is an array of length 3 with base type `int`

Multidimensional Arrays

In memory: flat

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]

$\underbrace{\texttt{a[0][0]}\ \texttt{a[0][1]}\ \texttt{a[0][2]}}_{\texttt{a[0]}}\ \underbrace{\texttt{a[1][0]}\ \texttt{a[1][1]}\ \texttt{a[1][2]}}_{\texttt{a[1]}}$

in our head: matrix

columns →

rows ↓

	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]

Multidimensional Arrays

- are arrays of arrays of arrays ....

T a[expr$_1$] ... [expr$_k$]

constant expressions

a has *expr*$_1$ elements and each of them is an array with *expr*$_2$ elements each of which is an array of *expr*$_3$ elements and ...

Multidimensional Arrays

Initialization

```
int a[][3] =
    {
        {2,4,6},{1,3,5}
    }
```

First dimension can be omitted

2	4	6	1	3	5

Vectors of Vectors

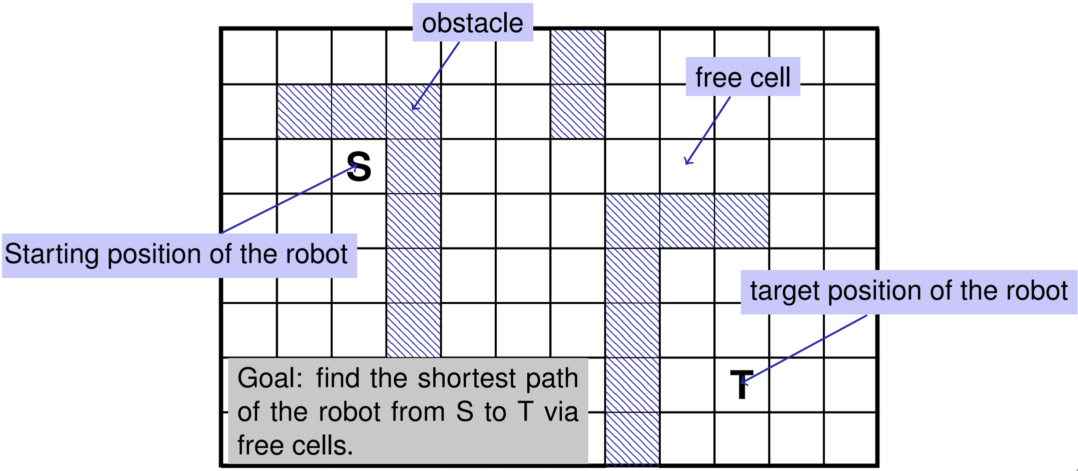
- How do we get multidimensional arrays with variable dimensions?
- Solution: vectors of vectors

Example: vector of length n of vectors with length m:

```
std::vector<std::vector<int> > a (n,
                    std::vector<int>(m));
```

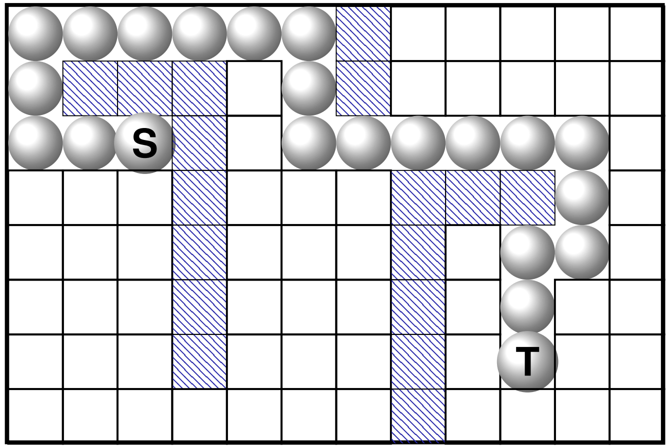
Application: Shortest Paths

Factory hall ($n \times m$ square cells)



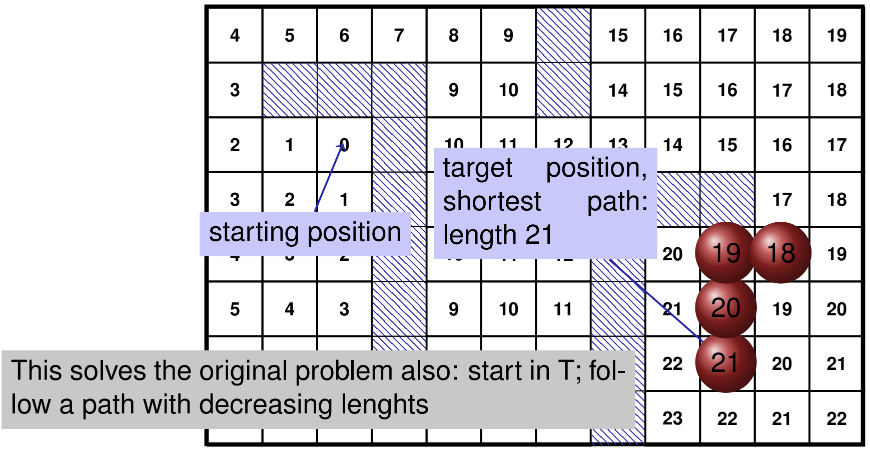
Application: shortest paths

Solution



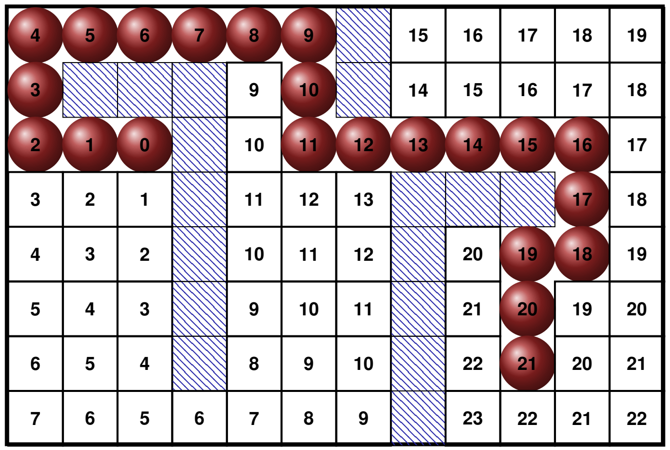
This problem appears to be different

Find the *lengths* of the shortest paths to *all* possible targets.

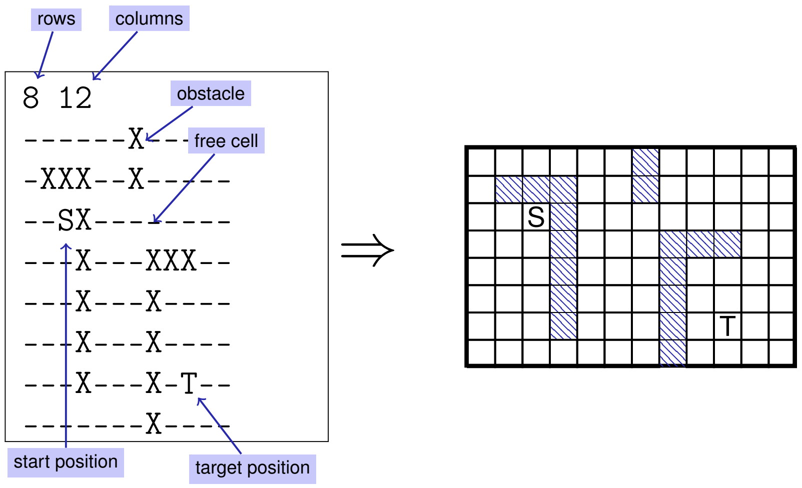


This problem appears to be different

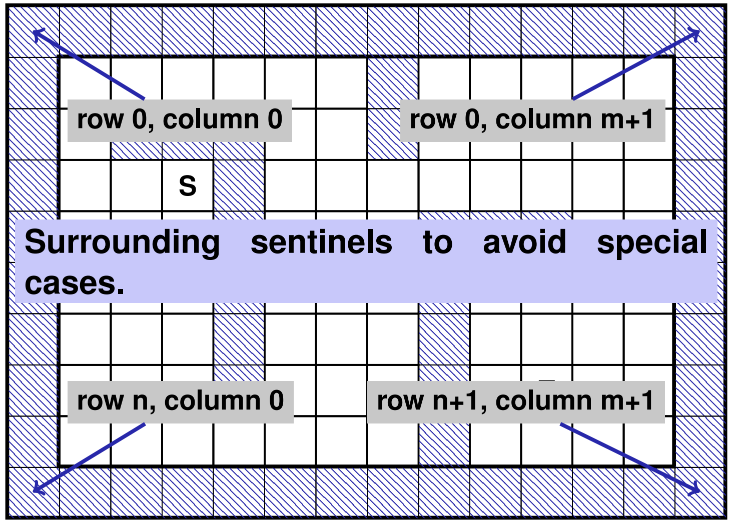
Find the *lengths* of the shortest paths to *all* possible targets.



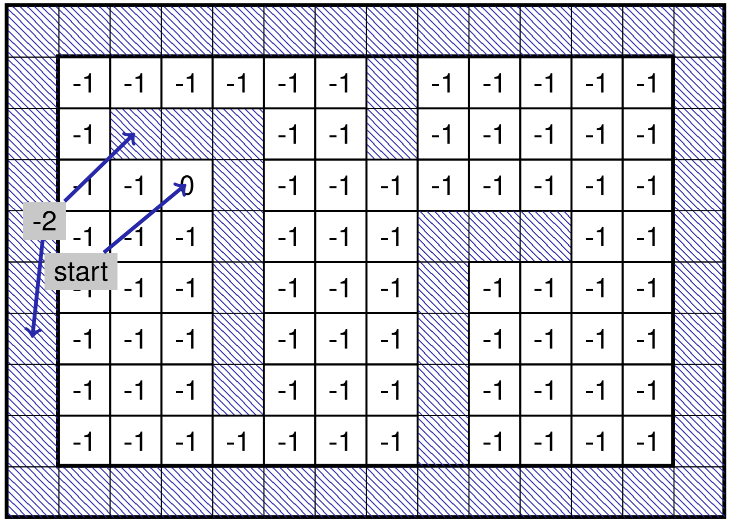
Preparation: Input Format



Preparation: Sentinels



Preparation: Initial Marking



The Shortest Path Program

- Read in dimensions and provide a two dimensional array for the path lengths

```
#include<iostream>
#include<vector>

int main()
{
  // read floor dimensions
  int n; std::cin >> n; // number of rows
  int m; std::cin >> m; // number of columns

  // define a two-dimensional
  // array of dimensions
  // (n+2) x (m+2) to hold the floor plus extra walls around
  std::vector<std::vector<int> > floor (n+2, std::vector<int>(m+2));
```

Sentinel

The Shortest Path Program

- Input the assignment of the hall and intialize the lengths

```
int tr = 0;
int tc = 0;
for (int r=1; r<n+1; ++r)
  for (int c=1; c<m+1; ++c) {
    char entry = '-';
    std::cin >> entry;
    if      (entry == 'S') floor[r][c] = 0;
    else if (entry == 'T') floor[tr = r][tc = c] = -1;
    else if (entry == 'X') floor[r][c] = -2;
    else if (entry == '-') floor[r][c] = -1;
  }
```


Das Kürzeste-Wege-Programm

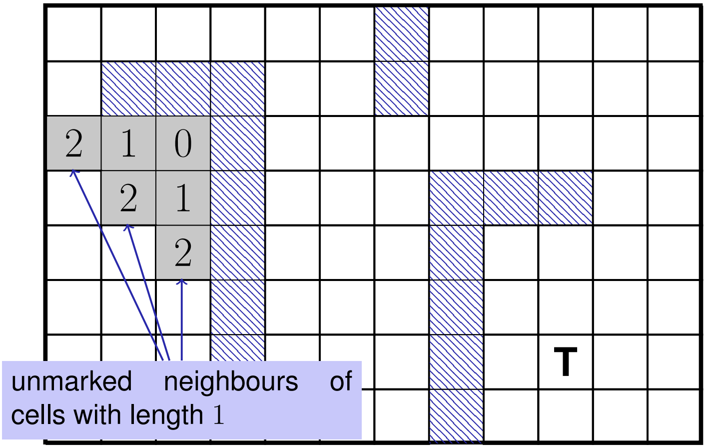
- Add the surrounding walls

```
for (int r=0; r<n+2; ++r)
  floor[r][0] = floor[r][m+1] = -2;

for (int c=0; c<m+2; ++c)
  floor[0][c] = floor[n+1][c] = -2;
```


Mark all Cells with their Path Lengths

Step 2: all cells with path length 2



Main Loop

Find and mark all cells with path lengths $i = 1, 2, 3...$

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

The Shortest Paths Program

Mark the shortest path by walking backwards from target to start.

```
int r = tr; int c = tc;
while (floor[r][c] > 0) {
  const int d = floor[r][c] − 1;
  floor[r][c] = −3;
  if      (floor[r−1][c] == d) −−r;
  else if (floor[r+1][c] == d) ++r;
  else if (floor[r][c−1] == d) −−c;
  else ++c; // (floor[r][c+1] == d)
}
```


Finish

-3	-3	-3	-3	-3	-3		15	16	17	18	19
-3				9	-3		14	15	16	17	18
-3	-3	0		10	-3	-3	-3	-3	-3	-3	17
3	2	1		11	12	13				-3	18
4	3	2		10	11	12		20	-3	-3	19
5	4	3		9	10	11		21	-3	19	20
6	5	4		8	9	10		22	-3	20	21
7	6	5	6	7	8	9		23	22	21	22

The Shortest Path Program: output

Output

```
for (int r=1; r<n+1; ++r) {
  for (int c=1; c<m+1; ++c)
    if (floor[r][c] == 0)
        std::cout << 'S';
    else if (r == tr && c == tc)
        std::cout << 'T';
    else if (floor[r][c] == -3)
        std::cout << 'o';
    else if (floor[r][c] == -2)
        std::cout << 'X';
    else
        std::cout << '-';
  std::cout << "\n";
}
```

$\Rightarrow$

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


The Shortest Paths Program

- Algorithm: *Breadth First Search*
- The program can become pretty slow because for each i all cells are traversed
- Improvement: for marking with i, traverse only the neighbours of the cells marked with $i - 1$.

Arrays as Function Arguments

Arrays can also be passed as *reference* arguments to a function. (here: const because v is read-only)

```
void print_vector(const int (&v)[3]) {
  for (int i = 0; i<3 ; ++i) {
    std::cout << v[i] << " ";
  }
}
```


Arrays as Function Argumenbts

This also works for multidimensional arrays.

```
void print_matrix(const int (&m)[3][3]) {
  for (int i = 0; i<3 ; ++i) {
    print_vector (m[i]);
    std::cout << "\n";
  }
}
```


Vectors as Function Arguments

Vectors can be passed *by value* or by *reference*

```
void print_vector(const std::vector<int>& v) {
  for (int i = 0; i<v.size() ; ++i) {
    std::cout << v[i] << " ";
  }
}
```

Here: *call by reference* is more efficient because the vector could be very long

Vectors as Function Arguments

This also works for multidimensional vectors.

```
void print_matrix(const std::vector<std::vector<int> >& m) {
  for (int i = 0; i<m.size() ; ++i) {
    print_vector (m[i]);
    std::cout << "\n";
  }
}
```

13. Pointers, Algorithms, Iterators and Containers I

Pointers, Address operator, Dereference operator, Array-to-Pointer Conversion

Strange Things. . .

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

int main(){
  int a[] = {3, 2, 1, 5, 4, 6, 7};

  // output the smallest element of a
  std::cout << *std::min_element (a, a + 7);
                ↑                      ↑
  return 0;    ???                    ???
}
```

We have to undestand *pointers* first!

References: Where is Anakin?

"Search for Vader, and Anakin find you will"

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

// anakin_skywalker = 22
```

Pointers: Where is Anakin?

"Anakins address is `0x7fff6bdd1b54`."

```
int anakin_skywalker = 9;
int* here = &anakin_skywalker;
std::cout << here; // Address
*here =  22;

// anakin_skywalker = 22
```

Swap with Pointers

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

...
int a = 2;
int b = 1;
swap(&a, &b);
std::cout << "a= " << a << "\n"; // 1
std::cout << "b = " << b << "\n"; // 2
```


Pointer Types

`T*` Pointer type to base type `T`.

An expression of type `T*` is called *pointer* (to `T`).

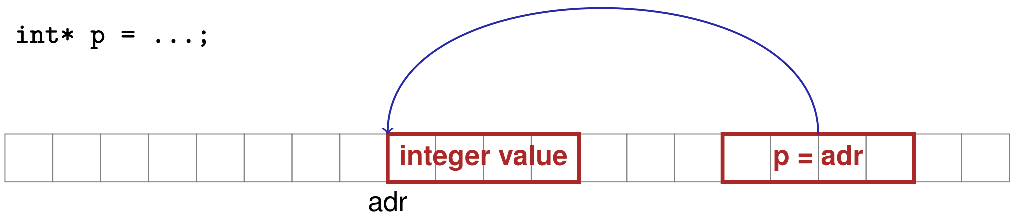
Pointer Types

Value of a pointer to `T` is the address of an object of type `T`.

Beispiele

`int* p;` Variable `p` is pointer to an `int`.
`float* q;` Variable `q` is pointer to a `float`.

```
int* p = ...;
```



Address Operator

The expression

L-value of type `T`

↓

& lval

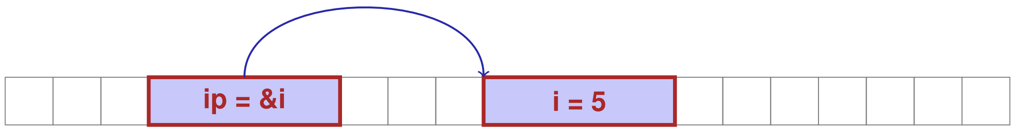
provides, as R-value, a *pointer* of type *T** to an object at the address of *lval*

The operator *&* is called Address-Operator.

Address Operator

Example

```
int i = 5;
int* ip = &i;  // ip initialized
               // with address of i.
```



Dereference Operator

The expression

R-value of type `T*`

↓

**rval*

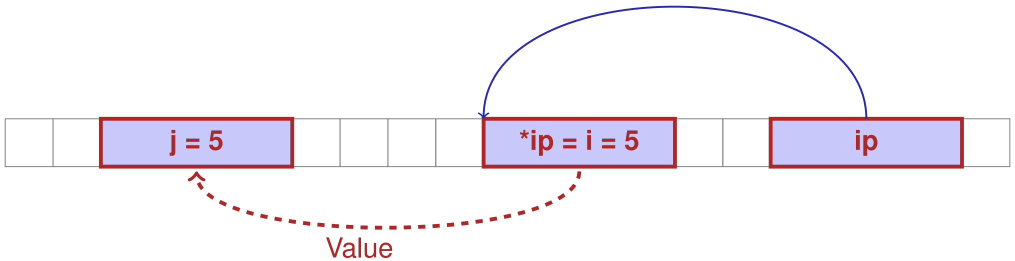
returns as L-value the *value* of the object at the address represented by *rval*.

The operator `*` is called Derecerence Operator.

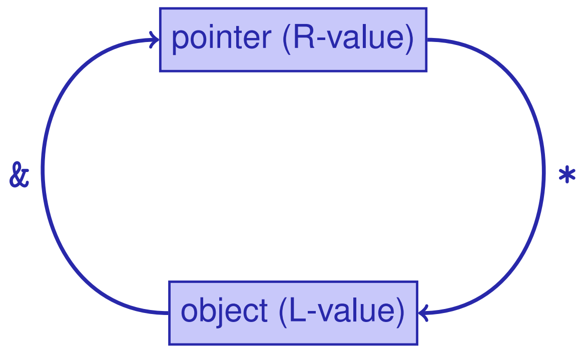
Dereference Operator

Beispiel

```
int i = 5;
int* ip = &i;  // ip initialized
               // with address of i.
int j = *ip;   // j == 5
```



Address and Dereference Operators



Pointer Types

Do not point with a `double*` to an `int`!

Examples

```
int* i = ...;     // at address i "lives" an int...
double* j = i;    //...and at j lives a double: error!
```


Mnenmonic Trick

The declaration

`T* p;` `p` is of the type "pointer to `T`"

can be read as

`T *p;` `*p` is of type `T`

Although this is legal, we do not write it like this!

Pointer Arithemtics: Pointer plus `int`

- *ptr*: Pointer to element $a[k]$ of the array a with length n
- Value of *expr*: integer i with $0 \leq k + i \leq n$

ptr + expr

is a pointer to $a[k + i]$.

For $k + i = n$ we get a *past-the-end*-pointer that must not be dereferenced.

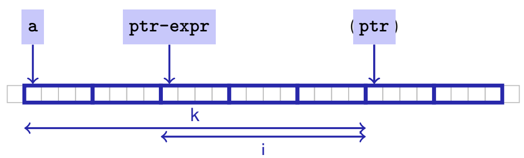
Pointer Arithemtics: Pointer minus `int`

- If *ptr* is a pointer to the element with index k in an array a with length n
- and the value of *expr* is an integer i, $0 \leq k - i \leq n$,

then the expression

ptr - expr

provides a pointer to an element of a with index $k - i$.



Conversion Array ⇒ Pointer

How do we get a pointer to the first element of an array?

- Static array of type $T[\mathrm{n}]$ is convertible to $T*$

Example

```
int a[5];
int* begin = a; // begin points to a[0]
```

- Length information is lost („arrays are primitive”)

Iteration over an Array of Pointers

Example

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

- `a+5` is a pointer behind the end of the array (past-the-end) that must not be dereferenced.
- The pointer comparison (`p < a+5`) refers to the order of the two addresses in memory.

14. Pointers, Algorithms, Iterators and Containers II

Iterations with Pointers, Arrays: Indices vs. Pointers, Arrays and Functions, Pointers and const, Algorithms, Container and Iteration, Vector-Iteration, Typdef, Sets, the Concept of Iterators

Recall: Pointers running over the Array

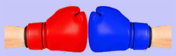
Beispiel

```
int a[5] = {3, 4, 6, 1, 2};
for (int* p = a; p < a+5; ++p)
   std::cout << *p << ' '; // 3 4 6 1 2
```

- An array can be converted into a pointer to its first element.
- Pointers “know” arithmetics and comparisons.
- Pointers can be dereferenced.

⇒ Pointers can be used to operate on arrays.

Arrays: Indices vs. Pointer



```
int a[n];

// Task: set all elements to 0

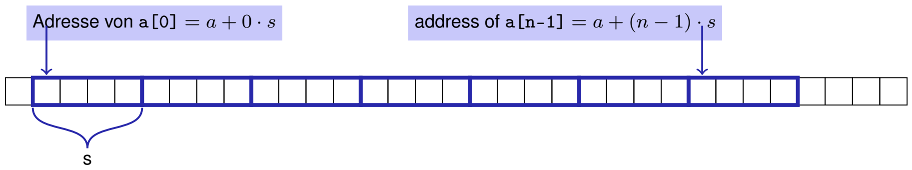
// Solution with indices is more readable
for (int i = 0; i < n; ++i)
    a[i] = 0;

// Solution with pointers is faster and more generic
int* begin = a;  // Pointer to the first element
int* end = a+n;  // Pointer past the end
for (int* p = begin; p != end; ++p)
   *p = 0;
```


Arrays and Indices

```
// Set all elements to value
for (int i = 0; i < n; ++i)
   a[i] = value;
```

Computational costs



⇒ One **addition** and one **multiplication** per element

The Truth about Random Access

The expression

`a[i]`

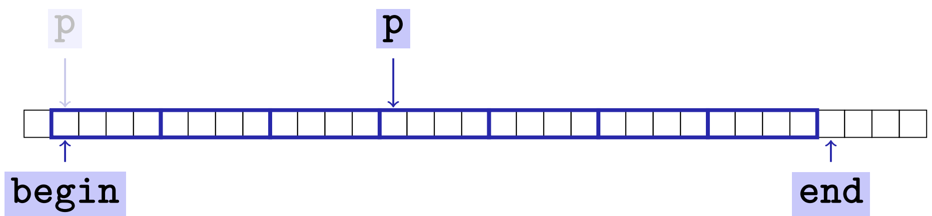
is equivalent to

$$\underset{\uparrow \atop a + i \cdot s}{*(a\ +\ i)}$$

Arrays and Pointers

```
// set all elements to value
for (int* p = begin; p != end; ++p)
   *p = value;
```

Computational cost



⇒ one **addition** per element

Reading a book . . . with indices . . . with pointers

Random Access

- open book on page 1
- close book
- open book on pages 2-3
- close book
- open book on pages 4-5
- close book
- ....

Sequential Access

- open book on page 1
- turn the page
- turn the page
- turn the page
- turn the page
- turn the page
- ...

Array Arguments: *Call by (const) reference*

```
void print_vector (const int (&v)[3]) {
  for (int i = 0; i<3 ; ++i) {
    std::cout << v[i] << " ";
  }
}
void make_null_vector (int (&v)[3]) {
  for (int i = 0; i<3 ; ++i) {
    v[i] = 0;
  }
}
```


Array Arguments: *Call by value (not really ...)*

```
void make_null_vector (int v[3]) {
  for (int i = 0; i<3 ; ++i) {
    v[i] = 0;
  }
}
...
int a[10];
make_null_vector (a); // only sets a[0], a[1], a[2]

int∗ b;
make_null_vector (b); // no array at b, crash!
```


Array Arguments: *Call by value* does not exist

- Formal argument types *T* `[n]` or *T* `[]` (array over `T`) are equivalent to *T*∗ (pointer to `T`)
- For passing an array the pointer to its first element is passed
- length information is lost
- Function cannot work on a part of an array (example: search for an element in the second half of an array)

Arrays in Functions

Covention of the standard library: pass an array (or a part of it) using two pointers

- `begin`: pointer to the first element
- `end`: pointer *behind* the last element
- `[begin, end)` designates the elements of the part of the array
- *valid* range means: there are array elements "available" here.
- `[begin, end)` is empty if `begin == end`

Arrays in Functions: `fill`

```
// PRE: [begin, end) is a valid range
// POST: every element within [begin, end) will be set to value
void fill (int* begin, int* end, int value) {
  for (int* p = begin; p != end; ++p)
     *p = value;
}
  ...

int a[5];
fill (a, a+5, 1);
for (int i=0; i<5; ++i)
  std::cout << a[i] << " ";
```

expects pointers to the first element of a range

pass the address (of the first element) of a

Pointers are not Integers!

- Addresses can be interpreted as house numbers of the memory, that is, integers
- But integer and pointer arithmetics behave differently.

`ptr + 1` is *not* the next house number but the s-next, where s is the memory requirement of an object of the type behind the pointer `ptr`.

- Integers and pointers are not compatible

```
int* ptr = 5; // error: invalid conversion from int to int*
int a = ptr;  // error: invalid conversion from int* to int
```


Null-Pointer

- special pointer value that signals that no object is pointed to
- represented b the integer number 0 (convertible to `T*`)

```
int* iptr = 0;
```

- cannot be dereferenced (checked during runtime)
- to avoid undefined behavior

```
int* iptr; // iptr points into ‘‘nirvana’’
int j = *iptr; // illegal address in *
```

Pointer Subtraction

- If *p1* and *p2* point to elements of the same array `a` with length n
- and $0 \leq k_1, k_2 \leq n$ are the indices corresponding to *p1* and *p2*, then

p1 - p2 has value k_1 - k_2

Only valid if *p1* and p2 point into the same array.

- The pointer difference describes “how far away the elements are from each other”

Pointer Operators

Description	Op	Arity	Precedence	Associativity	Assignment
Subscript	[]	2	17	left	R-value → L-value
Dereference	*	1	16	right	R-Wert → L-Wert
Address	&	1	16	rechts	L-value → R-value

Precedences and associativities of +, −, ++ (etc.) like in chapter 2

Mutating Functions

- Pointers can (like references) be used for functions with effect

Beispiel

```
int a[5];
fill(a, a+5, 1); // modifies a
```

pass address of the element past `a`

pass address of the first element of `a`

- Such functions are called *mutating*

Const Correctness

- There are also *non*-mutating functions that access elements of an array only in a read-only fashion

```
// PRE: [begin , end) is a valid and nonempty range
// POST: the smallest value in [begin, end) is returned
int min (const int* begin ,const int* end)
{
   assert (begin != end);
   int m = *begin; // current minimum candidate
   for (const int* p = ++begin; p != end; ++p)
      if (*p < m) m = *p;
   return m;
}
```

- mark with `const`: value of objects cannot be modified through such `const`-pointers.

Const and Pointers

Where does the `const`-modifier belong to?

const T is equivalent to *T const* and can be written like this

`const int a;`	$\Longleftrightarrow$	`int const a;`
`const int* a;`	$\Longleftrightarrow$	`int const *a;`

Read the declaration from right to left

`int const a;`	a is a constant inteher
`int const* a;`	a is a pointer to a constant integer
`int* const a;`	a is a constant pointer to an integer
`int const* const a;`	a is a constant pointer to a constant integer

`const` is not absolute

- The value at an address can change even if a `const`-pointer stores this address.

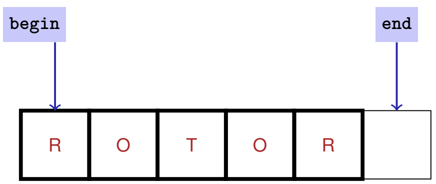
beispiel

```
int a[5];
const int* begin1 = a;
int*       begin2 = a;
*begin1 = 1;    // error *begin1 is constt
*begin2 = 1;    // ok, although *begin will be modified
```

- `const` is a promise from the point of view of the `const`-pointer, not an absolute guarantee

Wow – Palindromes!

```
// PRE: [begin end) is a valid range of characters
// POST: returns true if the range forms a palindrome
bool is_palindrome (const char* begin, const char* end) {
  while (begin < end)
    if (*(begin++) != *(--end)) return false;
  return true;
}
```



Algorithms

For many problems there are prebuilt solutions in the standard library

Example: filling an array

```
#include <algorithm> // needed for std::fill
...

int a[5];
std::fill (a, a+5, 1);

for (int i=0; i<5; ++i)
   std::cout << a[i] << " "; // 1 1 1 1 1
```

Algorithms

Advantages of using the standard library

- simple programs
- less sources of errors
- good, efficient code
- code independent from the data type
- there are also algorithms for more complicated problems such as the efficient sorting of an array

Algorithms

The same prebuilt algorithms work for many different data types.

Example: filling an array

```
#include <algorithm> // needed for std::fill
...

char c[3];
std::fill (c, c+3, '!');

for (int i=0; i<3; ++i)
    std::cout << c[i]; // !!!
```


Excursion: Templates

- Templates permit the provision of a type as argument
- The compiler finds the matching type from the call arguments

Example `fill` with templates

```
template <typename T>
void fill (T* begin , T* end, T value) {
    for (T* p = begin; p != end; ++p)
        *p = value;
}
int a[5];
fill (a, a+5, 1);   // 1 1 1 1 1

char c[3];
fill (c, c+3, '!'); // !!!
```

The triangular brackets we already know from vectors. Vectors are also implemented as templates.

`std::fill` is also implemented as template!

Containers and Traversal

- **Container:** Container (Array, Vector, ...) for elements
- **Traversal:** Going over all elements of a container
  - Initialization of all elements (`fill`)
  - Find the smallest element (`min`)
  - Check properties (`is_palindrome`)
  - ...
- There are a lot of different containers (sets, lists, ...)

Iteration Tools

- Arrays: indices (random access) or pointers (natural)
- Array algorithms (`std::`) use pointers

```
int a[5];
std::fill (a, a+5, 1);   // 1 1 1 1 1
```

- How do you traverse vectors and other containers?

```
std::vector<int> v (5, 0); // 0 0 0 0 0
std::fill (?, ?, 1);   // 1 1 1 1 1
```


Vectors: *too sexy for pointers*

- Our `fill` with templates does not work for vectors...
- ...and `std::fill` also does not work in the following way:

```
std::vector<int> v (5, 0);
std::fill (v, v+5, 1);   // Compiler error message !
```

Vectors are snobby...

- they refuse to be converted to pointers,...
- ...and cannot be traversed using pointers either.
- They consider this far too primitive.☺

Also in memory: Vector $\neq$ Array

```
bool a[8] = {true, true, true, true, true, true, true, true};
```

true	*true*	*true*	*true*	*true*	*true*	*true*	*true*

8 Byte (Speicherzelle = 1 Byte = 8 Bit)

```
std::vector<bool> v (8, true);
```

0b11111111

1 Byte

`bool*`-pointer does not fit here because it runs byte-wise and not bit-wise

Vector-Iterators

Iterator: a "pointer" that fits to the container.

Example: fill a vector using `std::fill` – this works

```
#include <vector>
#include <algorithm> // needed for std::fill

...
std::vector<int> v(5, 0);
std::fill (v.begin(), v.end(), 1);
for (int i=0; i<5; ++i)
    std::cout << v[i] << " "; // 1 1 1 1 1
```

Vector Iterators

For each vector there are two *iterator types* defined

- `std::vector<int>::const_iterator`
  - for non-mutating access
  - in analogy with `const int*` for arrays
- `std::vector<int>::iterator`
  - for mutating access
  - in analogy with `int*` for arrays
- A vector-iterator `it` is no pointer, but it behaves like a pointer:
  - it points to a vector element and can be dereferenced (`*it`)
  - it knows arithmetics and comparisons (`++it`, `it+2`, `it < end`,...)

Vector-Iterators: `begin()` and `end()`

- `v.begin()` points to the first element of `v`
- `v.end()` points past the last element of `v`
- We can traverse a vector using the iterator...

```
for (std::vector<int>::const_iterator it = v.begin();
                                      it != v.end(); ++it)
    std::cout << *it << " ";
```

- ...or fill a vector.

```
std::fill (v.begin(), v.end(), 1);
```


Type names in C++ can become looooooong

- `std::vector<int>::const_iterator`
- The declaration of a *type alias* helps with

`typedef` *Typ Name*;

existing type

Name that can now be used to access the type

Examples

```
typedef std::vector<int> int_vec;
typedef int_vec::const_iterator Cvit;
```


Vector Iterators work like Pointers

```
typedef std::vector<int>::const_iterator Cvit;

std::vector<int> v(5, 0); // 0 0 0 0 0

// output all elements of a, using iteration
for (Cvit it = v.begin(); it != v.end(); ++it)
    std::cout << *it << " ";
```

Vector element pointed to by `it`

Vector Iterators work like Pointers

```
typedef std::vector<int>::iterator Vit;

// manually set all elements to 1
for (Vit it = v.begin(); it != v.end(); ++it)
    *it = 1;

// output all elements again, using random access
for (int i=0; i<5; ++i)
    std::cout << v[i] << " ";
```

increment the iterator

short term for
`*(v.begin()+i)`

Other Containers: Sets

- A set is an unordered collection of elements, where each element is contained only once.

$$\{1,2,1\} = \{1,2\} = \{2,1\}$$

- C++: `std::set<T>` for a set with elements of type `T`

Sets: Example Application

- Determine if a given text contains a question mark and output all *pairwise different* characters!

Letter Salad (1)

Consider a text as a set of characters.

```
#include<set>
...
typedef std::set<char>::const_iterator Csit;
...
std::string text =
"What are the distinct characters in this string?";

std::set<char> s (text.begin(),text.end());
```

Set is initialized with String iterator range `[text.begin(), text.end())`

Letter Salad (2)

Determine if the text contains a question mark and output all characters

Search algorithm, can be called with arbitrary iterator range

```
// check whether text contains a question mark
if (std::find (s.begin(), s.end(), '?') != s.end())
    std::cout << "Good question!\n";

// output all distinct characters
for (Csit it = s.begin(); it != s.end(); ++it)
    std::cout << *it;
```

Ausgabe:

```
Good question!
?Wacdeghinrst
```


Sets and Indices?

- Can you traverse a set using random access? No.

```
for (int i=0; i<s.size(); ++i)
    std::cout << s[i];
```

error message: no subscript operator

- Sets are unordered.
  - There is no "ith element".
  - Iterator comparison `it != s.end()` works, but not `it < s.end()`!

The Concept of Iterators

C++knows different iterator types

- Each container provides an associated iterator type.
- All iterators can dereference (`*it`) and traverse (`++it`)
- Some can do more, e.g. random access (`it[k]`, or, equivalently `*(it + k)`), traverse backwards (`--it`),...

The Concept of Iterators

Every container algorithm is generic, that means:

- The container is passed as an iterator-range
- The algorithm works for all containers that fulfil the requirements of the algorihm
- `std::find` only requires `*` and `++` , for instance
- The implementation details of a container are irrelevant.

Why Pointers and Iterators?

Would you not prefer the code

```
for (int i=0; i<n; ++i)
    a[i] = 0;
```

over the following code?

```
for (int* ptr=a; ptr<a+n; ++ptr)
    *ptr = 0;
```

Maybe, but in order to use the generic `std::fill(a, a+n, 0);`, we *have to* work with pointers.

Why Pointers and Iterators?

In order to use the standard library, we have to know that:

- a static array `a` is a the same time a pointer to the first element of `a`
- `a+i` is a pointer to the element with index i

Using the standard library with different containers: Pointers $\Rightarrow$ Iterators

Why Pointers and Iterators?

Example: To search the smallest element of a container in the range $[\text{begin}, \text{end})$ use the function call

```
std::min_element(begin, end)
```

- returns an *iterator* to the smallest element
- To read the smallest element, we need to dereference:

```
*std::min_element(begin, end)
```


That is Why: Pointers and Iterators

- Even for non-programmers and “dumb” users of the standard library: expressions of the form
  `*std::min_element(begin, end)`
  cannot be understood without knowing pointers and iterators.
- Behind the scenes of the standard library: working with dynamic memory based on pointers is indispensible. More about this later in this course.

15. Recursion 1

Mathematical Recursion, Termination, Call Stack, Examples, Recursion vs. Iteration

Mathematical Recursion

- Many mathematical functions can be naturally defined recursively.
- This means, the function appears in its own definition

$$n! = \begin{cases} 1, & \text{if } n \leq 1 \\ n \cdot (n-1)!, & \text{otherwise} \end{cases}$$

Recursion in $\mathrm{C}{+}{+}$: In the same Way!

$$n! = \begin{cases} 1, & \text{if } n \leq 1 \\ n \cdot (n-1)!, & \text{otherwise} \end{cases}$$

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{
  if (n <= 1)
    return 1;
  else
    return n * fac (n-1);
}
```


Infinite Recursion

- is as bad as an infinite loop. . .
- . . . but even worse: it burns time and memory

```
void f()
{
  f(); // f() -> f() -> ... stack overflow
}
```

Recursive Functions: Termination

As with loops we need

- progress towards termination

fac(n):
terminates immediately for $n \leq 1$, otherwise the function is called recusively with < n .

„n is getting smaller for each call."

Recursive Functions: Evaluation

Example: `fac(4)`

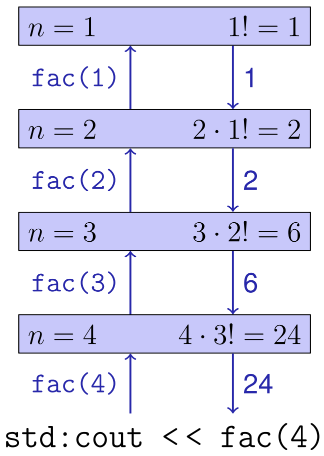
```
// POST: return value is n!
unsigned int fac (unsigned int n)
{
  if (n <= 1) return 1;
  return n * fac(n-1); // n > 1
}
```

Initialization of the formal argument: $n = 4$
recursive call with argument $n - 1 == 3$

The Call Stack

For each function call:

- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



Euclidean Algorithm

- finds the greatest common divisor $\gcd(a, b)$ of two natural numbers a and b
- is based on the following mathematical recursion (proof in the lecture notes):

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

Euclidean Algorithm in C++

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

```
unsigned int gcd
 (unsigned int a, unsigned int b)
{
  if (b == 0)
    return a;
  else
    return gcd (b, a % b);
}
```

Termination: $a \bmod b < b$, thus b gets smaller in each recursive call.

Fibonacci Numbers

$$F_n := \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2}, & \text{if } n > 1 \end{cases}$$

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 \ldots$

Fibonacci Numbers in C++

Laufzeit

`fib(50)` takes "forever" because it computes F_{48} two times, F_{47} 3 times, F_{46} 5 times, F_{45} 8 times, F_{44} 13 times, F_{43} 21 times ... F_1 ca. 10^9 times (!)

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

Correctness and termination are clear.

Fast Fibonacci Numbers

Idea:

- Compute each Fibonacci number only once, in the order $F_0, F_1, F_2, \ldots, F_n$!
- Memorize the most recent two numbers (variables `a` and `b`)!
- Compute the next number as a sum of `a` and `b`!

Fast Fibonacci Numbers in $\mathrm{C}{++}$

```
unsigned int fib (unsigned int n){
  if (n == 0) return 0;
  if (n <= 2) return 1;
  unsigned int a = 1;  // F_1
  unsigned int b = 1;  // F_2
  for (unsigned int i = 3; i <= n; ++i){
    unsigned int a_old = a;  // F_i-2
    a = b;                   // F_i-1
    b += a_old;              // F_i-1 += F_i-2 -> F_i
  }
  return b;
}
```

very fast, also for `fib(50)`

$$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$$

a b

Recursion and Iteration

Recursion can *always* be simulated by

- Iteration (loops)
- explicit "call stack" (e.g. array)

Often recursive formulations are simpler, but sometimes also less efficient.

16. Recursion 2

Building a Calculator, Streams, Formal Grammars, Extended Backus Naur Form (EBNF), Parsing Expressions

Motivation: Calculator

Goal: we build a command line calculator

Example

Input: `3 + 5`
Output: `8`
Input: `3 / 5`
Output: `0.6`
Input: `3 + 5 * 20`
Output: `103`
Input: `(3 + 5) * 20`
Output: `160`
Input: `-(3 + 5) + 20`
Output: `12`

- binary Operators `+`, `-`, `*`, `/` and numbers
- floating point arithmetic
- precedences and associativities like in $\mathrm{C}{++}$
- parentheses
- unary operator `-`

Naive Attempt (without Parentheses)

```
double lval;
std::cin >> lval;

char op;
while (std::cin >> op && op != '=') {
    double rval;
    std::cin >> rval;

    if (op == '+')
        lval += rval;
    else if (op == '*')
        lval *= rval;
    else ...
}
std::cout << "Ergebnis " << lval << "\n";
```

```
Input 2 + 3 * 3 =
Result 15
```


Analyzing the Problem

Example

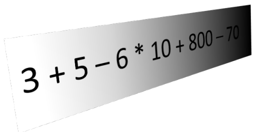
Input:

$$\underbrace{13 + 4 * (15 - 7 * 3)}_{\text{Needs to be stored such that evaluation can be performed}} =$$

"Understanding" expressions requires a lookahead to upcoming symbols!

Example

As Preparation: Streams



A program takes inputs from a conceptually infinite input stream.

So far: command line input stream `std::cin`

```
while (std::cin >> op && op != '=') { ... }
```

↑ Consume op from `std::cin`, reading position advances.

In the future we also want to be able to read from files.

Example: BSD 16-bit Checksum

```
#include <iostream>

int main () {

  char c;
  int checksum = 0;
  while (std::cin >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  std::cout << "checksum = " << std::hex << checksum << "\n";
}
```

Input: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Requires a manual termination of the input at the console[7]

Output: 67fd

[7] Ctrl-D(Unix) / Ctrl-Z(Windows) at the beginning of a line that is concluded with ENTER

Example: BSD 16-bit Checksum with a File

output: 67fd

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

int main () {
  std::ifstream fileStream ("loremispum.txt");
  char c;
  int checksum = 0;
  while (fileStream >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  std::cout << "checksum = " << std::hex << checksum << "\n";
}
```

returns `false` when file end is reached.

Example: BSD 16-bit Checksum

Reuse of common functionality?

Correct: with a function. But how?

Example: BSD 16-bit Checksum Generic!

Reference required: we modify the stream.

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

int checksum (std::istream& is)
{
  char c;
  int checksum = 0;
  while (is >> c) {
    checksum = checksum / 2 + checksum % 2 * 0x8000 + c;
    checksum %= 0x10000;
  }
  return checksum;
}
```


Equal Rights for All!

input: Lorem Yps with Gimmick
output: checksums differ

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

int checksum (std::istream& is) { ... }

int main () {
  std::ifstream fileStream("loremipsum.txt");

  if (checksum (fileStream) == checksum (std::cin))
        std::cout << "checksums match.\n";
  else
        std::cout << "checksums differ.\n";
}
```

Why does that work?

- `std::cin` is a variable of type `std::istream`. It represents an input stream.
- Our variable `fileStream` is of type `std::ifstream`. It represents an input stream on a file.
- A `std::ifstream` *is also a* `std::istream`, with more features.
- Therefore `fileStream` can be used wherever a `std::istream` is required.

Again: Equal Rights for All!

input from `stringStream`
output: checksums differ

```
#include <iostream>
#include <fstream>
#include <sstream>

int checksum (std::istream& is) { ... }

int main () {
  std::ifstream fileStream ("loremipsum.txt");
  std::stringstream stringStream ("Lorem Yps mit Gimmick");

  if (checksum (fileStream) == checksum (stringStream))
      std::cout << "checksums match.\n";
  else
      std::cout << "checksums differ.\n";
}
```


Back to Expressions

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

"Understanding an expression requires lookahead to upcoming symbols!

We will store symbols elegantly using recursion.

We need a new formal tool (that is independent of $\mathrm{C}++$).

Formal Grammars

- Alphabet: finite set of symbols Σ
- Strings: finite sequences of symbols Σ^*

A formal grammar defines which strings are valid.

Mountains

- Alphabet: $\{/, \backslash\}$
- Mountains $\mathcal{M} \subset \{/, \backslash\}^*$ (valid strings)

$m' =$ `//\\///\\\`

```
           /\
      /\  /  \
     /  \/    \
```


Forbidden Mountains

- Alphabet: $\{/, \backslash\}$
- Mountains: $\mathcal{M} \subset \{/, \backslash\}^*$ (valid strings)

$m''' =$ `/\\//\` $\notin \mathcal{M}$

```
 /\  /\
   \/
```

Both sides should have the same height. A mountain cannot fall below its starting height.

Mountains in Backus-Naur-Form (BNF)

mountain = "/\" | "/" mountain "\" | mountain mountain. **Rules**

alternatives

nonterminal

terminal

Possible Mountains

1.
```
/\
```

2.
```
                /\/\
  /\/\         /    \
 /    \   ⇒   /      \
```

3.
```
                   /\                     /\
  /\/\     /\     /  \       /\/\   /\   /  \
 /    \   /  \/    \    ⇒   /    \/  \/    \
```

It is possible to prove that this BNF describes “our” mountains, which is not completely clear a priori.

Expressions

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

What do we need in the BNF?

- Number , (Expression)
  -Number, -(Expression) — **Factor**
- Factor $*$ Factor, Factor
  Factor $*$ Factor / Factor , ... — **Term**
- Term + Term, Term
  Term - Term, ... — **Expression**

The BNF for Expressions

A factor is

- a number,
- an expression in parentheses or
- a negated factor.

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


The BNF for Expressions

A term is

- factor,
- factor ∗ factor, factor / factor,
- factor ∗ factor ∗ factor, factor / factor ∗ factor, ...
- ...

We need repetition!

EBNF

Extended Backus Naur Form: extends the BNF by

- option [] and
- optional repetition {}

```
term = factor { "∗" factor | "/" factor }.
```

Remark: the EBNF is not more powerful than the BNF. But it allows a more compact representation. The construct from above can be written as follows:

```
term = factor | factor T.
T = "∗" term | "+" term.
```


The EBNF for Expressions

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

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

expression = term { "+" term | "−" term }.
```

Parsing

- **Parsing:** Check if a string is valid according to the (E)BNF.
- **Parser:** A program for parsing.
- **Useful:** From the (E)BNF we can (nearly) automatically generate a parser:
  - Rules become functions
  - Alternatives and options become `if`–statements.
  - Nonterminial symbols on the right hand side become function calls
  - Optional repetitions become `while`–statements

Functions (Parser with Evaluation)

Expression is read from an input stream.

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

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

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


One Character Lookahead...

...to find the right alternative.

```
// POST: leading whitespace characters are extracted
//       from is, and the first non-whitespace character
//       is returned (0 if there is no such character)
char lookahead (std::istream& is)
{
  if (is.eof())
    return 0;
  is >> std::ws;          // skip whitespaces
  if (is.eof())
    return 0;             // end of stream
  return is.peek();     // next character in is
}
```


Cherry-Picking

...to extract the desired character.

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

Evaluating Factors

```
double factor (std::istream& is)
{
  double v;
  if (consume(is, '(')){
    v = expression (is);
    consume(is, ')');
  } else if (consume(is, '-'))
    v = -factor (is);
  else
    is >> v;
  return v;
}
```

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

Evaluating Terms

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

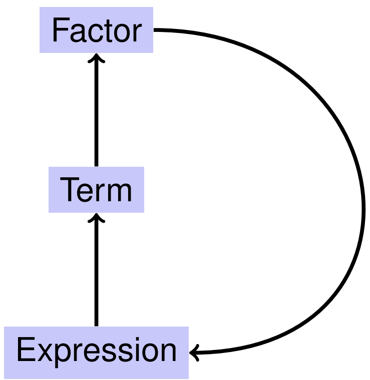
term = factor { "*" factor | "/" factor }

Evaluating Expressions

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

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

Recursion!



EBNF — and it works!

EBNF (`calculator.cpp`, Evaluation from left to right):

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

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

expression = term { "+" term | "−" term }.
```

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


BNF — and it does not work!

BNF (`calculator_r.cpp`, Evaluation from right to left):

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

term       = factor | factor "∗" term | factor "/" term.

expression = term | term "+" expression | term "−" expression.
```

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


Analysis: Repetition vs. Recursion

Simplification: sum and difference of numbers

Examples

$$3, \quad 3-5, \quad 3-7-1$$

EBNF:

```
sum = value {"−" value | "+" value}.
```

BNF:

```
sum = value | value "−" sum | value "+" sum.
```

Both grammars permit the same kind of expressions.

`value`

```
double value (std::istream& is){
    double val;
    is >> val;
    return val;
}
```

EBNF Variant

```
// sum = value {"-" value | "+" value}.
double sum(std::istream& is) {
    double v = value(is);
    while(true){
        if (consume(is, '-'))
            v -= value(is);
        else if (consume(is, '+'))
            v += value(is);
        else
            return v;
    }
}
```


We test: EBNF Variant

- input: 1-2
  output: -1 ✓
- input: 1-2-3
  output: -4 ✓

BNF Variant

```
// sum = value | value "-" sum | value "+" sum.
double sum(std::istream& is){
    double v = value(is);
    if (consume(is, '-'))
        return v - sum(is);
    else if(consume(is, '+'))
        return v + sum(is);
    return v;
}
```


We test: BNF Variant

- input: 1-2
  output: -1 ✓
- input: 1-2-3
  output: 2

We Test

Is the BNF wrong ?

```
sum = value
      | value "−" sum
      | value "+" sum.
```

- No, it does only determine the validity of expressions, not their values!
- The evaluation we have put on top naively.

Getting to the Bottom of Things

```
double sum (std::istream& is){
   double v = value (is);
   if (consume (is,'−'))
      v −= sum (is);
   else if (consume (is,'+'))
     v += sum(is);
   return v;
}
...
std::stringstream input ("1−2−3");
std::cout << sum (input) << "\n"; // 4
```

3	3
2 - "3"	-1
1 - "2 - 3"	2
"1 - 2 - 3"	2

What has gone wrong?

The BNF

- does officially not talk about values
- but it still suggests the wrong kind of evaluation order.

```
sum = value | value "−" sum | value "+" sum.
```

naturally leads to

$1 - 2 - 3 = 1 - (2 - 3)$

A Solution: Left-Recursion

```
sum  = value | sum "−" value | sum "+" value.
```

Implementation pattern from before does not work any more.
Left-recursion must be resolved to right-recursion.

This is what it looks like:

```
sum  = value | value s.
s = "−" sum | "+" sum.
```

Cf. `calculator_l.cpp`

17. Structs and Classes I

Rational Numbers, Struct Definition, Overlading Functions and Operators, Const-References, Encapsulation

Calculating with Rational Numbers

- Rational numbers ($\mathbb{Q}$) are of the form $\frac{n}{d}$ with n and d in $\mathbb{Z}$
- $\mathrm{C}{+}{+}$does not provide a built-in type for rational numbers

Goal

We build a $\mathrm{C}{+}{+}$-type for rational numbers ourselves! ☺

Vision

How it could (will) look like

```
// input
std::cout << "Rational number r =? ";
rational r;
std::cin >> r;
std::cout << "Rational number s =? ";
rational s;
std::cin >> s;

// computation and output
std::cout << "Sum is " << r + s << ".\n";
```


A First Struct

Invariant: specifies valid value combinations (informal).

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

member variable (numerator)

member variable (denominator)

- `struct` defines a new *type*
- formal range of values: *cartesian product* of the value ranges of existing types
- real range of values: `rational` $\subsetneq$ `int` $\times$ `int`.

Accessing Member Variables

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

rational add (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
```

$$\frac{r_n}{r_d} := \frac{a_n}{a_d} + \frac{b_n}{b_d} = \frac{a_n \cdot b_d + a_d \cdot b_n}{a_d \cdot b_d}$$

A First Struct: Functionality

A `struct` defines a new *type*, not a *variable*!

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

Meaning: every object of the new type is represented by two objects of type `int` the objects are called `n` and `d` .

```
// POST: return value is the sum of a and b
rational add (const rational a, const rational b)
{
  rational result;
  result.n =  a.n * b.d +  a.d * b.n;
  result.d = a.d * b.d;
  return result;
}
```

member access to the `int` objects of `a`.

Input

```
// Input r
rational r;
std::cout << "Rational number r:\n";
std::cout << " numerator =? ";
std::cin >> r.n;
std::cout << " denominator =? ";
std::cin >> r.d;

// Input s the same way
rational s;
...
```


Vision comes within Reach ...

```
// computation
const rational t = add (r, s);

// output
std::cout << "Sum is " << t.n << "/" << t.d << ".\n";
```

Struct Definitions

name of the new type (identifier)

names of the underlying types

names of the *member variables*

$$
\begin{array}{l}
\texttt{struct}\ T\ \{\\
\quad T_1\ \mathit{name}_1;\\
\quad T_2\ \mathit{name}_2;\\
\quad\ \vdots \qquad \vdots\\
\quad T_n\ \mathit{name}_n;\\
\};
\end{array}
$$

Range of Values of T: $T_1 \times T_2 \times ... \times T_n$

Struct Defintions: Examples

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

underlying types can be fundamental or user defined

Struct Definitions: Examples

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

the underlying types can be different

Structs: Accessing Members

expression of struct-type T

name of a member-variable of type T_k.

$expr.name_k$

expression of type T_k; value is the value of the object designated by $name_k$

member access operator .

Structs: Initialization and Assignment

Default Initialization:

```
rational t;
```

- Member variables of `t` are default-initialized
- for member variables of fundamental types nothing happens (values remain undefined)

Structs: Initialization and Assignment

Initialization:

```
rational t = {5, 1};
```

- Member variables of `t` are initialized with the values of the list, according to the declaration order.

Structs: Initialization and Assignment

Assignment:

```
rational s;
...
rational t = s;
```

- The values of the member variables of `s` are assigned to the member variables of `t`.

Structs: Initialization and Assignment

$\begin{matrix}\texttt{t.n}\\ \texttt{t.d}\end{matrix} \;\texttt{= add (r, s)}\begin{matrix}\texttt{.n}\\ \texttt{.d}\end{matrix}\;\texttt{;}$

Initialization:

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

- `t` is initialized with the values of `add(r, s)`

Structs: Initialization and Assignment

Assignment:

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

- `t` is default-initialized
- The value of `add (r, s)` is assigned to `t`

Structs: Initialization and Assignment

```
rational s;              ⟵ member variables are uninitialized
rational t = {1,5};      ⟵ member-wise initialization:
                            t.n = 1, t.d = 5
rational u = t;          ⟵ member-wise copy
t = u;                   ⟵ member-wise copy
rational v = add (u,t);  ⟵ member-wise copy
```


Comparing Structs?

For each fundamental type (`int, double,...`) there are comparison operators `==` and `!=` , not so for structs! Why?

- member-wise comparison does not make sense in general...
- ...otherwise we had, for example, $\frac{2}{3} \neq \frac{4}{6}$

Structs as Function Arguments

```
void increment(rational dest, const rational src)
{
    dest = add (dest, src); // modifies local copy only
}
```

Call by Value !

```
rational a;
rational b;
a.d = 1; a.n = 2;
b = a;
increment (b, a); // no effect!
std::cout << b.n << "/" << b.d; // 1 / 2
```

Structs as Function Arguments

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

Call by Reference

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


User Defined Operators

Instead of

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

we would rather like to write

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

This can be done with *Operator Overloading*.

Overloading Functions

- Functions can be addressed by name in a scope
- It is even possible to declare and to defined several functions with the same name
- the “correct” version is chosen according to the *signature* of the function.

Function Overloading

- A function is defined by name, types, number and order of arguments

```
double sq (double x) { ... }          // f1
int sq (int x) { ... }                // f2
int pow (int b, int e) { ... }        // f3
int pow (int e) { return pow (2,e); } // f4
```

- the compiler automatically chooses the function that fits “best” for a function call (we do not go into details)

```
std::cout << sq (3);     // compiler chooses f2
std::cout << sq (1.414); // compiler chooses f1
std::cout << pow (2);    // compiler chooses f4
std::cout << pow (3,3);  // compiler chooses f3
```

Operator Overloading

- Operators are special functions and can be overloaded
- Name of the operator *op*:

```
operatorop
```

- we already know that, for example, `operator+` exists for different types

Adding `rational` Numbers – Before

```
// POST: return value is the sum of a and b
rational add (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
const rational t = add (r, s);
```


Adding `rational` Numbers – After

```
// POST: return value is the sum of a and b
rational operator+ (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
const rational t = r + s;
```

infix notation

Other Binary Operators for Rational Numbers

```
// POST: return value is difference of a and b
rational operator− (rational a, rational b);

// POST: return value is the product of a and b
rational operator∗ (rational a, rational b);

// POST: return value is the quotient of a and b
// PRE: b != 0
rational operator/ (rational a, rational b);
```

Unary Minus

has the same symbol as the binary minus but only one argument:

```
// POST: return value is -a
rational operator- (rational a)
{
    a.n = -a.n;
    return a;
}
```


Comparison Operators

are not built in for structs, but can be defined

```
// POST: returns true iff a == b
bool operator== (rational a, rational b)
{
    return a.n * b.d == a.d * b.n;
}
```

$$\frac{2}{3} = \frac{4}{6} \quad \checkmark$$

Arithmetic Assignment

We want to write

```
rational r;
r.n = 1; r.d = 2;                    // 1/2

rational s;
s.n = 1; s.d = 3;                    // 1/3

r += s;
std::cout << r.n << "/" << r.d;      // 5/6
```


Operator+= First Trial

```
rational operator+= (rational a, rational b)
{
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

does not work. Why?

- The expression `r += s` has the desired value, but because the arguments are R-values (call by value!) it does not have the desired effect of modifying `r`.
- The result of `r += s` is, against the convention of C++ no L-value.

Operator +=

```
rational& operator+= (rational& a, rational b)
{
    a.n = a.n * b.d + a.d * b.n;
    a.d *= b.d;
    return a;
}
```

this works

- The L-value `a` is increased by the value of `b` and returned as L-value

  `r += s;` now has the desired effect.

In/Output Operators

can also be overloaded.

- Before:

```
std::cout << "Sum is "
          << t.n << "/" << t.d << "\n";
```

- After (desired):

```
std::cout << "Sum is "
          << t << "\n";
```


In/Output Operators

can be overloaded as well:

```
// POST: r has been written to out
std::ostream& operator<< (std::ostream& out,
                          rational r)
{
    return out << r.n << "/" << r.d;
}
```

writes `r` to the output stream
and returns the stream as L-value.

Input

```
// PRE: in starts with a rational number
// of the form "n/d"
// POST: r has been read from in
std::istream& operator>> (std::istream& in,
                          rational& r)
{
    char c; // separating character '/'
    return in >> r.n >> c >> r.d;
}
```

reads `r` from the input stream
and returns the stream as L-value.

Goal Attained!

```
// input
std::cout << "Rational number r =? ";
rational r;
std::cin >> r;

std::cout << "Rational number s =? ";
rational s;
std::cin >> s;

// computation and output
std::cout << "Sum is " << r + s << ".\n";
```

operator >>

operator +

operator<<

Recall: Large Objects ...

```
struct SimulatedCPU {
    unsigned int pc;
    int stack[16];
    unsigned int stackPosition;
    unsigned int memory[65536];
};

void outputState (SimulatedCPU p) {
    std::cout << "pc=" << p.pc;
    std::cout << ", stack: ";
    for (unsigned int i = p.stackPosition; i != 0; --i)
        std::cout << p.stack[i-1];
}
```

call by value: more than 256k get copied!

... are Better Passed as Const-Reference

```
struct SimulatedCPU {
    unsigned int pc;
    int stack[16];
    unsigned int stackPosition;
    unsigned int memory[65536];
};

void outputState (const SimulatedCPU& p) {
    std::cout << "pc=" << p.pc;
    std::cout << ", stack: ";
    for (int i = p.stackPosition; i != 0; --i)
        std::cout << p.stack[i-1];
}
```

call by reference: only the address gets copied.

A new Type with Functionality. . .

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

// POST: return value is the sum of a and b
rational operator+ (rational a, rational b)
{
    rational result;
    result.n = a.n * b.d + a.d * b.n;
    result.d = a.d * b.d;
    return result;
}
...
```

…should be in a Library!

`rational.h:`

- Definition of a struct `rational`
- Function declarations

`rational.cpp:`

- arithmetic operators (`operator+`, `operator+=`, ...)
- relational operators (`operator==`, `operator>`, ...)
- in/output (`operator >>`, `operator <<`, ...)

Thought Experiment

The three core missions of ETH:

- research
- education
- technology transfer

We found a startup: RAT PACK®!

- Selling the `rational` library to customers
- ongoing development according to customer's demands

The Customer is Happy

…and programs busily using `rational`.

- output as `double`-value $(\frac{3}{5} \rightarrow 0.6)$

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


The Customer Wants More

"Can we have rational numbers with an extended value range?"

- Sure, no problem, e.g.:

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

$\Rightarrow$

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

New Version of RAT PACK®

It sucks, nothing works any more!

- What is the problem?

$-\frac{3}{5}$ is sometimes 0.6, this cannot be true!

- That is your fault. Your conversion to `double` is the problem, our library is correct.

Up to now it worked, therefore the new version is to blame!

Liability Discussion

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

`r.is_positive` and `result.is_positive` do not appear.

correct using...

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

... not correct using

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


We are to Blame!!

- Customer sees and uses our representation of rational numbers (initially `r.n, r.d`)
- When we change it (`r.n, r.d, r.is_positive`), the customer's programs do not work anymore.
- No customer is willing to adapt the programs when the version of the library changes.

⇒ RAT PACK® is history...

Idea of Encapsulation (Information Hiding)

- A type is uniquely defined by its *value range* and its *functionality*
- The representation should not be visible.
- ⇒ The customer is not provided with representation but with functionality!

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

Classes

- provide the concept for encapsulation in C++
- are a variant of structs
- are provided in many object oriented programming languages

Encapsulation: `public` / `private`

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

is used instead of `struct` if anything at all shall be "hidden"

only difference

- `struct`: by default *nothing* is hidden
- `class` : by default *everything* is hidden

18. Classes

Classes, Member Functions, Constructors, Stack, Linked List, Dynamic Memory, Copy-Constructor, Assignment Operator, Concept Dynamic Datatype

Encapsulation: `public` / `private`

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

Good news: `r.d = 0` cannot happen any more by accident.

Bad news: the customer cannot do anything any more ...

... and we can't, either. (no `operator+`, ...)

Application Code

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

Member Functions: Declaration

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

public area

member function

member functions have access to private data

the scope of members in a class is the whole class, independent of the declaration order

Member Functions: Call

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

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

member access

Member Functions: Definition

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

- A member function is called for an expression of the class. in the function, `*this` is the name of this implicit argument. `this` itself is a pointer to it.
- const refers to `*this`, i.e., it promises that the value associated with the implicit argument cannot be changed
- `n` is the shortcut in the member function for `(*this).n`

Comparison

It would look like this...

```
class rational {
    int n;
    ...

    int numerator () const
    {
        return (*this).n;
    }
};

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

... without member functions

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

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

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

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

```
class rational {
    int n;
    ...

    int numerator () const
    {
        return n;
    }
    ....
};
```

- No separation between declaration and definition (bad for libraries)

```
class rational {
    int n;
    ...

    int numerator () const;
    ...
};

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

- This also works.

Constructors

- are special member functions of a class that are named like the class
- can be overloaded like functions, i.e. can occur multiple times with varying *signature*
- are called like a function when a variable is declared. The compiler chooses the “closest” matching function.
- if there is no matching constructor, the compiler emits an *error message*.

Initialisation? Constructors!

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

Initialization of the member variables

function body.

Constructors: Call

- directly

```
rational r (1,2); // initialisiert r mit 1/2
```

- indirectly (copy)

```
rational r = rational (1,2);
```

Initialisation “rational = int”?

```
class rational
{
public:
    rational (int num)
      : n (num), d (1)
    {}   ←—— empty function body
 ...
};
 ...
rational r (2);    // explicit initialization with 2
rational s = 2;    // implicit conversion
```


User Defined Conversions

are defined via constructors with exactly *one* argument

```
rational (int num)  ←——
    : n (num), d (1)
  {}
```

User defined conversion from `int` to `rational`. values of type `int` can now be converted to `rational`.

```
rational r = 2; // implizite Konversion
```


The Default Constructor

```
class rational
{
public:
     ...
    rational ()     ← empty list of arguments
      : n (0), d (1)
    {}
 ...
};
 ...
rational r;    // r = 0
```

⇒ There are no uninitiatlized variables of type rational any more!

The Default Constructor

- is automatically called for declarations of the form `rational r;`
- is the unique constructor with empty argmument list (if existing)
- must exist, if `rational r;` is meant to compile
- if in a struct there are no constructors at all, the default constructor is automatically generated

RAT PACK® Reloaded ...

Customer's program now looks like this:

```
// POST: double approximation of r
double to_double (const rational r)
{
  double result = r.numerator();
  return result / r.denominator();
}
```

- We can adapt the member functions together with the representation ✓

RAT PACK® Reloaded ...

before

```
class rational {
...
private:
  int n;
  int d;
};
```

```
int numerator () const
{
  return n;
}
```

after

```
class rational {
...
private:
  unsigned int n;
  unsigned int d;
  bool is_positive;
};
```

```
int numerator () const{
  if (is_positive)
    return n;
  else {
    int result = n;
    return −result;
  }
}
```


RAT PACK® Reloaded ?

```
class rational {
...
private:
  unsigned int n;
  unsigned int d;
  bool is_positive;
};
```

```
int numerator () const
{
  if (is_positive)
    return n;
  else {
    int result = n;
    return −result;
  }
}
```

- value range of nominator and denominator like before
- possible overflow in addition

Encapsulation still Incompleete

Customer's point of view (`rational.h`):

```
class rational {
public:
   // POST: returns numerator of *this
   int numerator () const;
   ...
private:
  // none of my business
};
```

- We determined denominator and nominator type to be `int`
- Solution: encapsulate not only data but alsoe types.

Fix: “our” type `rational::integer`

Customer’s point of view (`rational.h`):

```
public:
   typedef int integer; // might change
   // POST: returns numerator of *this
   integer numerator () const;
```

- We provide an additional type!
- Determine only Functionality, e.g:
  - implicit conversion `int` → `rational::integer`
  - function `double to_double (rational::integer)`

RAT PACK® Revolutions

Finally, a customer program that remains stable

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


Separate Declaration and Definition

```
class rational {
public:
   rational (int num, int denum);
   typedef int integer;
   integer numerator () const;
   ...
private:
 ...
};
```

rational.h

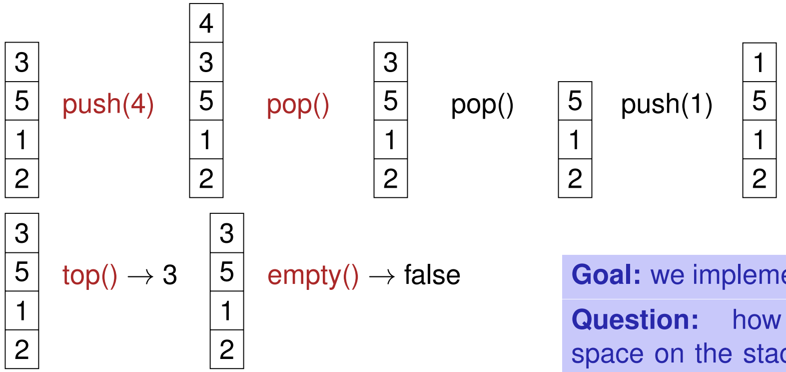
```
rational::rational (int num, int den):
    n (num), d (den) {}
rational::integer rational::numerator () const
{
    return n;
}
```

rational.cpp

class name :: member name

Motivation: Stack

Motivation: Stack (`push`, `pop`, `top`, `empty`)



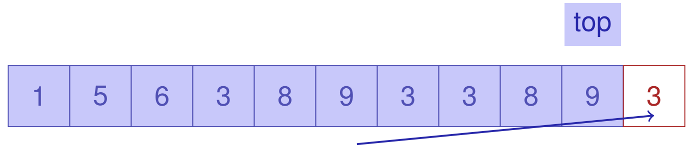
Goal: we implement a stack class

Question: how do we create space on the stack when push is called?

We Need a new Kind of Container

Our main container: Array (`T[]`)

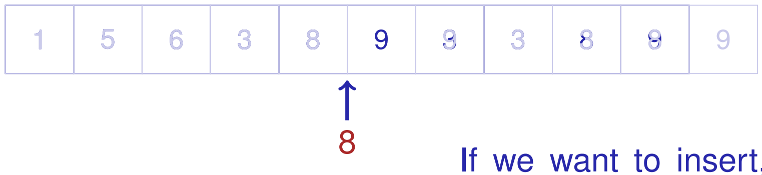
- Contiguous area of memory, random access (to ith element)
- Simulation of a stack with an array?
- No, at some point the array will become "full".



not possible to execute `push(3)` here!

Arrays are no all-rounders…

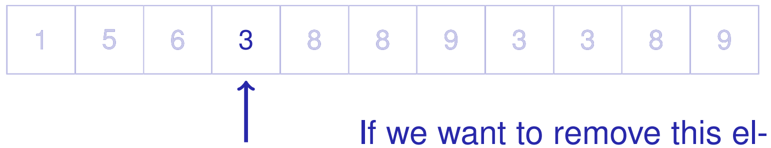
- It is expensive to insert or delete elements "in the middle ".



If we want to insert, we have to move everything to the right (if there is space at all!)

Arrays are no all-rounders…

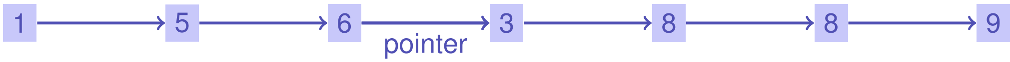
- It is expensive to insert or delete elements "in the middle ".



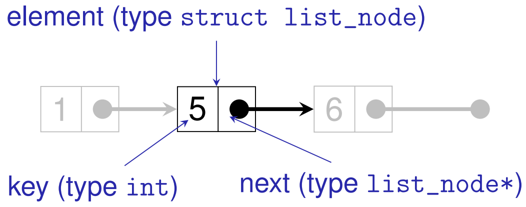
If we want to remove this element, we have to move everything to the right of it.

The new Container: Linked List

- *No* contiguous area of memory and *no* random access
- Each element “knows” its successor
- Insertion and deletion of arbitrary elements is simple, *even at the beginning of the list*
- ⇒ A stack can be implemented as linked list

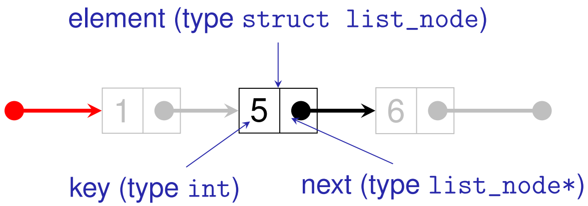


Linked List: Zoom




```
struct list_node {
  int          key;
  list_node*   next;
  // constructor
  list_node (int k, list_node* n)
    : key (k), next (n) {}
};
```

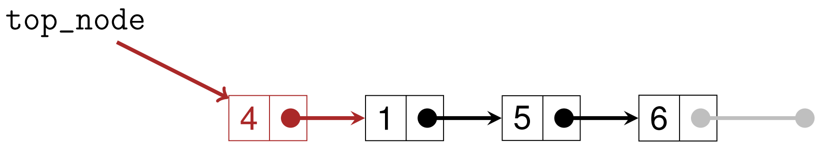

Stack = Pointer to the Top Element




```
class stack {
public:
   void push (int value) {...}
   ...
private:
   list_node* top_node;
};
```


Sneak Preview: `push(4)`

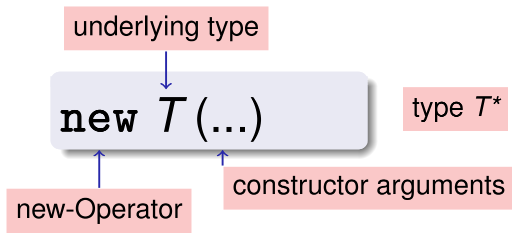
```
void push (int value)
{
  top_node = new list_node (value, top_node);
}
```



Dynamic Memory

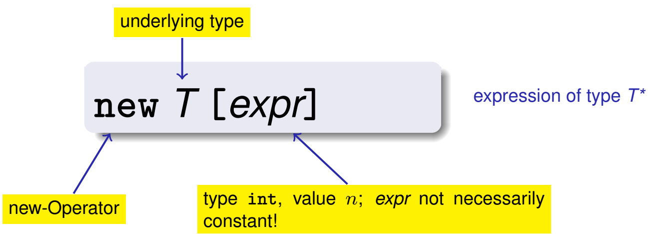
- For dynamic data structures like lists we need *dynamic memory*
- Up to now we had to fix the memory sizes of variable at *compile time*
- Pointers allow to request memory at *runtime*
- Dynamic memory management in C++ with operators `new` and `delete`

The `new` Expression



- **Effect:** new object of type *T* is allocated in memory ...
- ... and initialized by means of the matching constructor.
- **Value:** address of the new object

`new` for Arrays

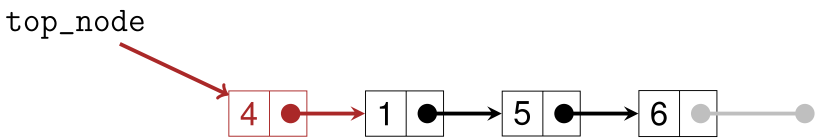


- memory for an array with length n and underlying type *T* is allocated
- Value of the expression is the address of the first element of the array

The `new` Expression `push(4)`

- **Effect:** new object of type *T* is allocated in memory ...
- ... and intialized by means of the matching constructor
- **Value:** address of the new object

```
top_node = new list_node (value, top_node);
```



The `delete` Expression

Objects generated with **`new`** have *dynamic storage duration:* they "live" until they are explicitly *deleted*

`delete` *expr*

type **`void`**

delete-Operator

pointer of type *T**, pointing to an object that had been created with **`new`**.

- **Effect:** object is deleted and memory is released

`delete` for Arrays

`delete[]` *expr*

type **`void`**

delete-Operator

pointer of type *T**, that points to an array that previously had been allocated using **`new`**

- **Effect:** array is deleted and memory is released

Carefult with `new` and `delete`!

```
rational* t = new rational;      ← memory for t is allocated
rational* s = t;                 ← other pointers may also point to the same object
delete s;                        ← ... and used for releaseing the object
int nominator = (*t).denominator();  ← error: memory already released!
```

Dereferencing of „dangling pointers"

- Pointer to released objects: *dangling pointers*
- Releasing an object more than once using **`delete`** is a similar severe error
- **`delete`** can be easily forgotten: consequence are *memory leaks*. Can lead to memory overflow in the long run.

Who is born must die...

Guideline "Dynamic Memory"

For each **`new`** there is a matching **`delete`**!

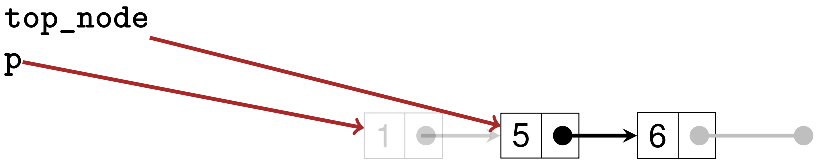
Non-compliance leads to memory leaks

- old objects that occupy memory...
- ...until it is full (heap overflow)

Stack Continued: `pop()`

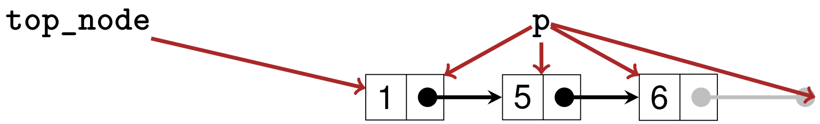
```
void pop()
{
    assert (!empty());
    list_node* p = top_node;
    top_node = top_node->next;
    delete p;
}
```

shortcut for `(*top_node).next`



Traverse the Stack `print()`

```
void print (std::ostream& o) const
{
  const list_node* p = top_node;
  while (p != 0) {
    o << p->key << " "; // 1 5 6
    p = p->next;
  }
}
```



Output Stack: `operator<<`

```
class stack {
public:
   void push (int value) {...}
   ...
   void print (std::ostream& o) const {...}
private:
   list_node* top_node;
};

// POST: s is written to o
std::ostream& operator<< (std::ostream& o, const stack& s)
{
   s.print (o);
   return o;
}
```


Empty Stack , `empty()`, `top()`

```
stack()       // default constructor
    : top_node (0)
{}

bool empty () const
{
  return top_node == 0;
}

int top () const
{
  assert (!empty());
  return top_node->key;
}
```

Stack Done? Obviously not...

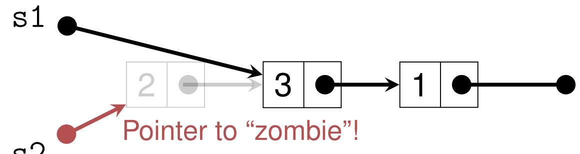
```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n"; // 2 3 1

stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop (); // Oops, crash!
```


What has gone wrong?



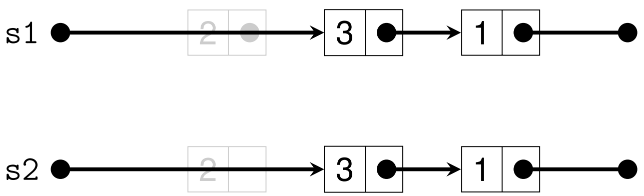
member-wise initialization: copies the `top_node` pointer only.

```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();  // Oops, crash!
```


We need a real copy




```
...
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1

s1.pop ();
std::cout << s1 << "\n"; // 3 1

s2.pop ();  // ok
```


The Copy Constructor

- The copy constructor of a class *T* is the unique constructor with declaration

  T (`const` *T*& x);

- is automatically called when values of type *T* are initialized with values of type `T`

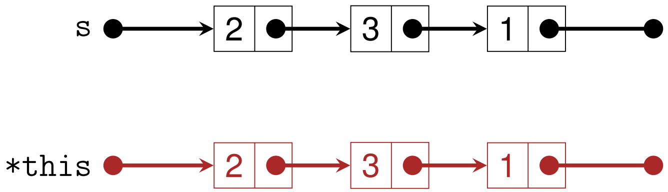
  `T x = t;`   (t of type T)
  `T x (t);`

- If there is no copy-constructor declared then it is generated automatically (and initializes member-wise – reason for the problem above

It works with a Copy Constructor

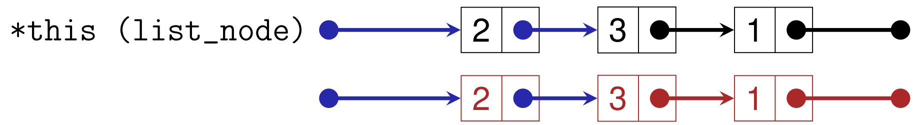
We use a copy function of the `list_node`:

```
// POST: *this is initialized with a copy of s
stack (const stack& s)
  : top_node (0)
{
  if (s.top_node != 0)
    top_node = s.top_node->copy();
}
```



The (Recursive) Copy Function of `list_node`

```
// POST: pointer to a copy of the list starting
//       at *this is returned
list_node* copy () const
{
  if (next != 0)
    return new list_node (key, next->copy());
  else
    return new list_node (key, 0);
}
```



Initialization ≠ Assignment!

```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n"; // 2 3 1

stack s2;
s2 = s1; // Zuweisung

s1.pop ();
std::cout << s1 << "\n"; // 3 1
s2.pop (); // Oops, Crash!
```


The Assignment Operator

- Overloading `operator=` as a member function
- Like the copy-constructor without initializer, but additionally
  - Releasing memory for the “old” value
  - Check for self-assignment (`s1=s1`) that should not have an effect
- If there is no assignment operator declared it is automatically generated (and assigns member-wise – reason for the problem above

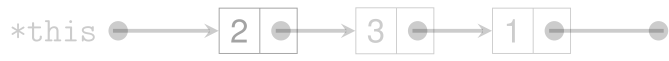
It works with an Assignment Operator!

Here a release function of the `list_node` is used:

```
// POST: *this (left operand) is getting a copy of s (right operand)
stack& operator= (const stack& s)
{
  if (top_node != s.top_node) { // keine Selbtszuweisung!
    if (top_node != 0) {
      top_node->clear(); // loesche Knoten in *this
      top_node = 0;
  }
    if (s.top_node != 0)
      top_node = s.top_node->copy(); // kopiere s nach *this
   }
  return *this; // Rueckgabe als L-Wert (Konvention)
}
```


The (recursive) release function of `list_node`

```
// POST: the list starting at *this is deleted
void clear ()
{
  if (next != 0)
    next->clear();
  delete this;
}
```



Zombie Elements

```
{
  stack s1; // local variable
  s1.push (1);
  s1.push (3);
  s1.push (2);
  std::cout << s1 << "\n"; // 2 3 1
}
// s1 has died (become invalid)...
```

- . . . but the three elements of the stack `s1` continue to live (memory leak)!
- They should be released together with `s1`.

The Destructor

- The Destructor of class T is the unique member function with declaration

$$\sim T\,(\,);$$

- is automatically called when the memory duration of a class object ends
- If no destructor is declared, it is automatically generated and calls the destructors for the member variables (pointers `top_node`, no effect – reason for zombie elements

Using a Destructor, it Works

```
// POST: the dynamic memory of *this is deleted
~stack()
{
  if (top_node != 0)
    top_node->clear();
}
```

- automatically deletes all stack elements when the stack is being released
- Now our stack class follows the guideline “dynamic memory”

Dynamic Datatype

- Type that manages dynamic memory (e.g. our class for a stack)
- Other Applications:
  - Lists (with insertion and deletion “in the middle”)
  - Trees (next week)
  - waiting queues
  - graphs
- Minimal Functionality:
  - Constructors
  - Destructor
  - Copy Constructor
  - Assignment Operator

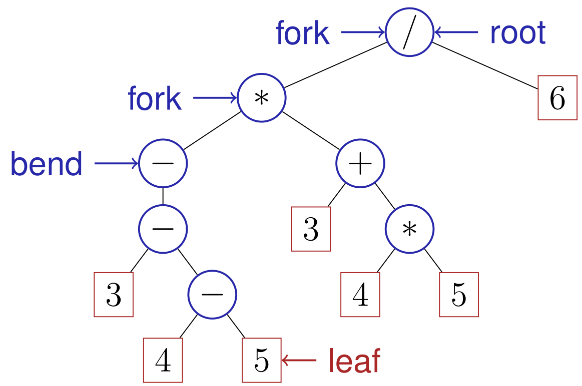
Rule of Three: if a class defines at least one of them, it must define all three

19. Inheritance and Polymorphism

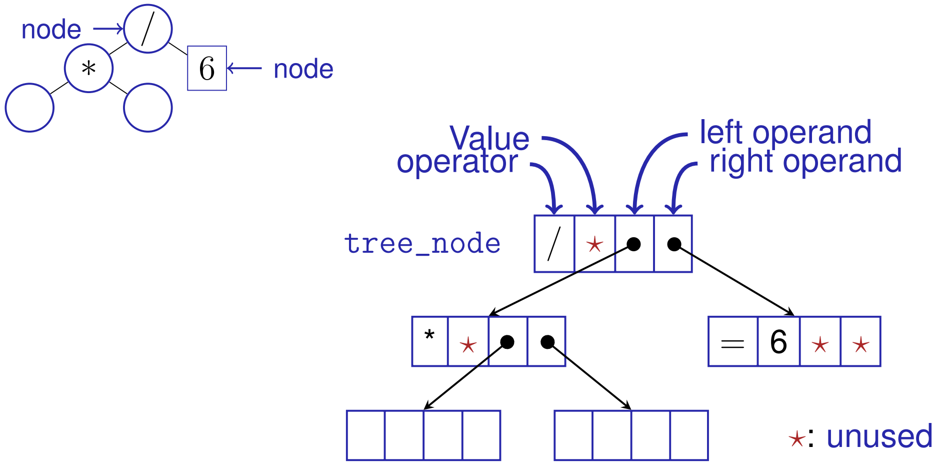
Expression Trees, Inheritance, Code-Reuse, Virtual Functions, Polymorphism, Concepts of Object Oriented Programming

(Expression) Trees

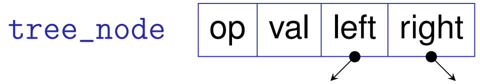
`-(3-(4-5))*(3+4*5)/6`



Nodes: Forks, Bends or Leaves

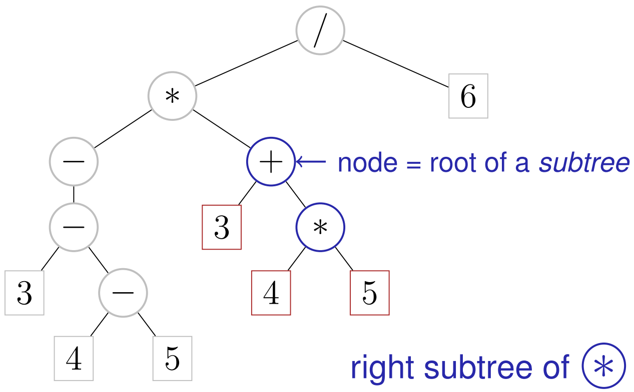


Nodes (`struct tree_node`)

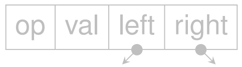



```
struct tree_node {
  char op;
  // leaf node (op: '=')
  double val;
  // internal node ( op: '+', '−', '∗', '/')
  tree_node∗ left; // == 0 for unary minus
  tree_node∗ right;
  // constructor
  tree_node (char o, double v, tree_node∗ l, tree_node∗ r)
    : op (o), val (v), left (l), right (r)
  {}
};
```


Nodes and Subtrees

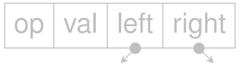


Count Nodes in Subtrees



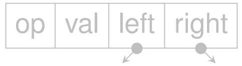

```
struct tree_node {
  ...
    // POST: returns the size (number of nodes) of
    //       the subtree with root ∗this
    int size () const
    {
        int s=1;
        if (left) // kurz für left != 0
            s += left−>size();
        if (right)
            s += right−>size();
        return s;
    }
};
```

Evaluate Subtrees



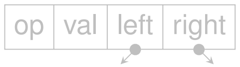

```
struct tree_node {
  ...
  // POST: evaluates the subtree with root *this
  double eval () const {
    if (op == '=') return val;←— leaf...
    double l = 0;                  ...or fork:
    if (left) l = left->eval();←— op unary, or left branch
    double r = right->eval();←— right branch
    if (op == '+') return l + r;
    if (op == '-') return l - r;
    if (op == '*') return l * r;
    if (op == '/') return l / r;
    return 0;
  }
};
```


Cloning Subtrees




```
struct tree_node {
  ...
  // POST: a copy of the subtree with root *this is
  //       made, and a pointer to its root node is
  //       returned
  tree_node* copy () const {
    tree_node* to = new tree_node (op, val, 0, 0);
    if (left)
      to->left = left->copy();
    if (right)
      to->right = right->copy();
    return to;
  }
};
```

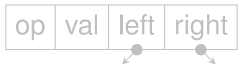

Cloning Subtrees – more Compact Notation



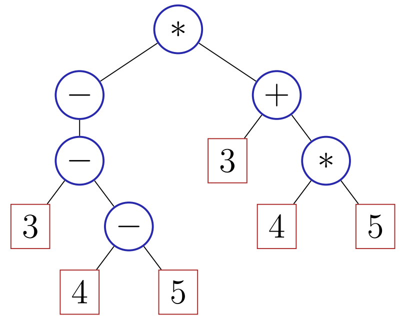

```
struct tree_node {
  ...
  // POST: a copy of the subtree with root *this is
  //       made, and a pointer to its root node is
  //       returned
  tree_node* copy () const {
     return new tree_node (op, val,
            left ? left->copy() : 0,
            right ? right->copy() : 0);
  }
};
```

cond ? expr1 : expr 2 has value *expr1*, if *cond* holds, *expr2* otherwise

Felling Subtrees




```
struct tree_node {
  ...
  // POST: all nodes in the subtree with root
  // *this are deleted
  void clear() {
    if (left) {
      left->clear();
    }
    if (right) {
      right->clear();
    }
    delete this;
  }
};
```



Powerful Subtrees!

```
struct tree_node {
  ...
  // constructor
  tree_node (char o, tree_node* l,
             tree_node* r, double v)

  // functionality
  double eval () const;
  void print (std::ostream& o) const;
  int size () const;
  tree_node* copy () const;
  void clear ();
};
```

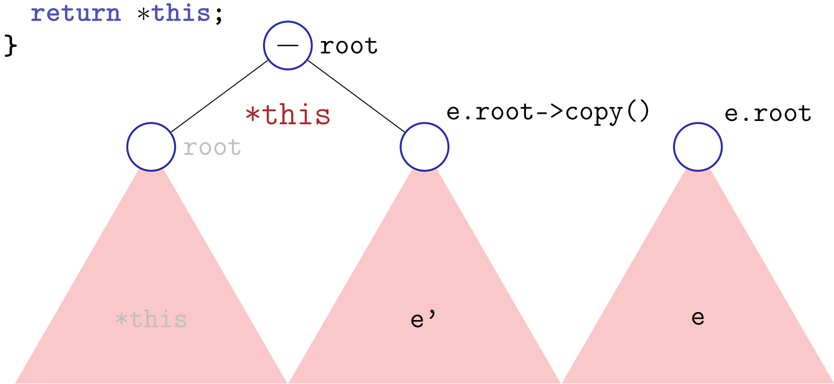

Planting Trees

```
class texpression {
private:
    tree_node* root;
public:
    ...
    texpression (double d)
        : root (new tree_node ('=', d, 0, 0)) {}
    ...
};
```

creates a tree with one leaf

Letting Trees Grow

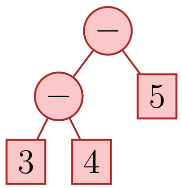
```
texpression& operator-= (const texpression& e)
{
  assert (e.root);
  root = new tree_node ('-', 0, root, e.root->copy());
  return *this;
}
```



Raising Trees

```
texpression operator- (const texpression& l,
                       const texpression& r)
{
    texpression result = l;
    return result -= r;
}

texpression a = 3;
texpression b = 4;
texpression c = 5;
texpression d = a-b-c;
```



Raising Trees

For `texpression` we also provide

- default constructor, copy constructor, assignment operator, destructor
- arithmetic assignments `+=, *=, /=`
- binary operators `+, *, /`
- the unary `-`

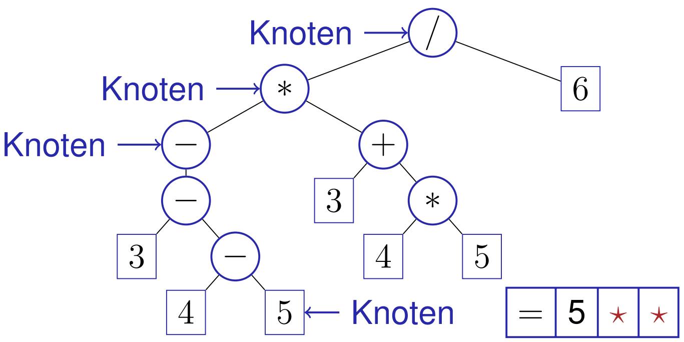
From Values to Trees!

```
typedef texpression result_type;   // Typ-Alias
```

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

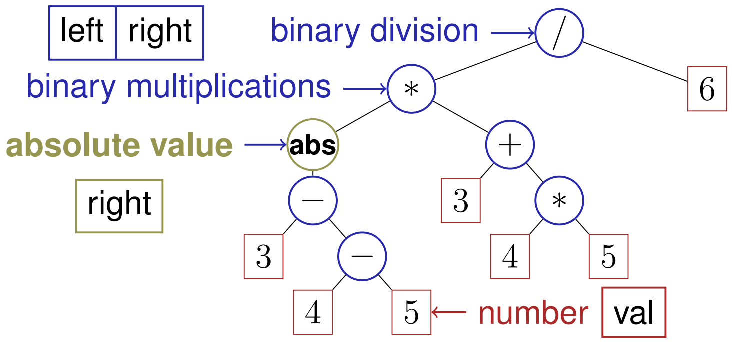
`double_calculator.cpp`
(expression value)
→
`texpression_calculator_l.cpp`
(expression tree)

Motivation Inheritance: Previously



- Nodes: Forks, Leafs and Bends
- ⇒ unused member variables ⋆

Motivation Inheritance: The Idea



- Everywhere only the necessary member variables
- Extension of “operator zoo” with new species!

Inheritance – The Hack, First. . .

Scenario: extension of the expression tree by mathematical functions `abs`, `sin`, `cos`:

- extension of the class `tree_node` by even more member variables

```
struct tree_node{
    char op; // neu: op = 'f' —> Funktion
    ...
    std::string name; // function name;
}
```

Disadvantages:

- Modification of the original code (undesirable)
- even more member variables. . .

Inheritance – The Hack, Second. . .

Scenario: extension of the expression tree by mathematical functions `abs`, `sin`, `cos`:

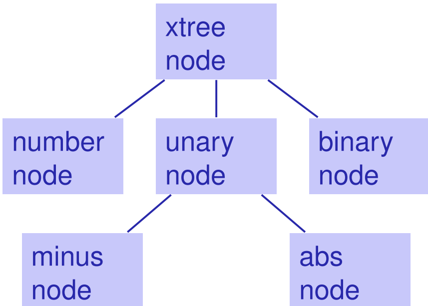
- Adaption of every single member function

```
double eval () const
{   ...
    else if (op == 'f')
        if (name == "abs")
            return std::abs(right—>eval());
    ...
}
```

Disadvantages:

- Loss of clarity
- hard to work in a team of developers

Inheritance – the Clean Solution



- “Split-up” of `tree_node`
- Common properties stay in the *base class* `xtree_node` (will be explained)

Inheritance

classes can *inherit* properties

```
struct xtree_node{
  virtual int size() const;
  virtual double eval () const;
};
```

erbt von

inheritance visible

```
struct number_node : public xtree_node {
  double val;
```

only for `number_node`

```
  int size () const;
  double eval () const;
};
```

members of `xtree_node` are *overwritten*

Inheritance – Notation

```
class A {            Base/Super Class
    ...
}

class B: public A{   Subclass
    ...
}

class C: public B{   "B and C inherit from A"
                     "C inherits from B"
    ...
}
```


Separation of Concerns: The Number Node

```
struct number_node: public xtree_node{
  double val;

  number_node (double v) : val (v) {}

  double eval () const {
    return val;
  }

  int size () const {
      return 1;
  }
};
```


A Number Node is a Tree Node...

- A (pointer to) an inheriting object can be used where (a pointer to) a base object is required , but not vice versa.

```
number_node* num = new number_node (5);

xtree_node* tn = num; // ok, number_node is
                      // just a special xtree_node

xtree_node* bn = new add_node (tn, num); // ok

number_node* nn = tn; //error:invalid conversion
```


Application

```
class xexpression {
private:                          static type
   xtree_node* root;
public:                                  dynamic type
    xexpression (double d)
      : root (new number_node (d)) {}

    xexpression& operator-= (const xexpression& t)
    {
      assert(t.root);
      root = new sub_node (root, t.root->copy());
      return *this;
    }
     ...
}
```

Polymorphism

```
struct xtree_node {
  virtual double eval();
...
};
```

- Without `Virtual` the *static type* determines which function is executed

We do not go into further details.

Separation of Concerns: Binary Nodes

```
struct binary_node : public xtree_node {
  xtree_node* left; // INV != 0
  xtree_node* right; // INV != 0

  binary_node (xtree_node* l, xtree_node* r) :
    left (l), right (r)
  {
    assert (left);
    assert (right);
  }

  int size () const {
    return 1 + left->size() + right->size();
  }
};
```

`size` works for *all* binary nodes. Derived classes (`add_node`,`sub_node`...) *inherit* this function!

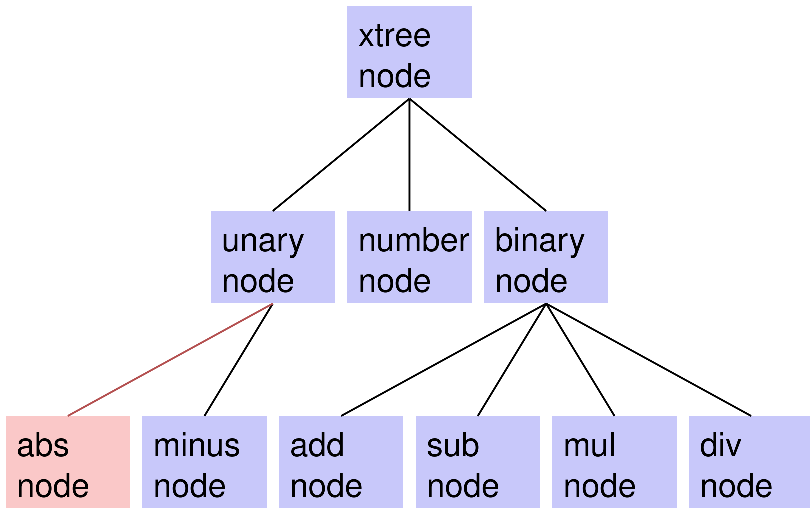
Separation of Concerns: +, −, ∗ ...

```
struct sub_node : public binary_node {
  sub_node (xtree_node* l, xtree_node* r)
    : binary_node (l, r) {}

  double eval () const {
    return left->eval() - right->eval();
  }
};
```

`eval` specific for +, −, ∗, /

Extension by `abs` Function



Extension by `abs` Function

```
struct unary_node: public xtree_node
{
  xtree_node∗ right; // INV != 0
  unary_node (xtree_node∗ r);
  int size () const;
};

struct abs_node: public unary_node
{
  abs_node (xtree_node∗ arg) : unary_node (arg) {}

  double eval () const {
    return std::abs (right−>eval());
  }

};
```


Do not forget. . . Memory Management

```
struct xtree_node {
  ...
  // POST: a copy of the subtree with root
  //       ∗this is made, and a pointer to
  //       its root node is returned
  virtual xtree_node∗ copy () const;

  // POST: all nodes in the subtree with
  //        root ∗this are deleted
  virtual void clear () {};
};
```


Do not forget. . . Memory Management

```
struct unary_node: public xtree_node {
  ...
  virtual void clear () {
    right−>clear();
    delete this;
  }
};

struct minus_node: public unary_node {
  ...
  xtree_node∗ copy () const
  {
    return new minus_node (right−>copy());
  }
};
```


`xtree_node` is no dynamic data type ??

- We do not have any variables of type `xtree_node` with automatic memory lifetime
- copy constructor, assignment operator and destructor are unnecessary
- memory management in the *container class*

```
class xexpression {
  // Copy−Konstruktor
  xexpression (const xexpression& v);
  // Zuweisungsoperator
  xexpression& operator=(const xexpression& v);
  // Destruktor
  ~xexpression ();
};
```

`xtree_node::copy`

`xtree_node::clear`

Mission: Monolithic → Modular ✓

```
struct tree_node {
  char op;
  double val;
  tree_node* left;
  tree_node* right;
  ...

  double eval () const
  {
    if (op == '=') return val;
    else {
    double l = 0;
    if (left != 0) l = left->eval();
    double r = right->eval();
    if (op == '+') return l + r;
    if (op == '-') return l - r;
    if (op == '*') return l * r;
    if (op == '/') return l / r;
    assert (false); // unknown operator
    return 0;
  }

  int size () const { ... }

  void clear() { ... }

  tree_node* copy () const { ... }

};
```

```
struct number_node: public xtree_node {
  double val;
  ...
  double eval () const {
  return val;
  }
};

struct minus_node: public unary_node {
  ...
  double eval () const {
    return -right->eval();
  }
};

struct minus_node : public binary_node {
  ...
  double eval () const {
    return left->eval() - right->eval();
  }
}
```

+

```
struct abs_node : public unary_node {
  ...
  double eval () const {
    return left->eval() - right->eval();
  }
}
```


Summary of the Concepts

.. of Object Oriented Programming

Encapsulation

- hide the implementation details of types
- definition of an interface for access to values and functionality (public area)
- make possible to ensure invariants and the modification of the implementation

Summary of Concepts

.. of Object Oriented Programming

Inheritance

- types can inherit properties of types
- inheriting types can provide new properties and overwrite existing ones
- allows to reuse code and data

Summary of Concepts

.. of Object Oriented Programming

Polymorphism

- A pointer may, depending on its use, have different underlying types
- the different underlying types can react differently on the same access to their common interface
- makes it possible to extend libraries “non invasively”

20. Conclusion

Purpose and Format

Name the most important key words to each chapter. Checklist: “does every notion make some sense for me?”

- Ⓜ motivating example for each chapter
- Ⓒ concepts that do not depend from the implementation (language)
- Ⓛ language (C++): all that depends on the chosen language
- Ⓔ examples from the lectures

1. Introduction

Ⓜ
- Euclidean algorithm

Ⓒ
- algorithm, Turing machine, programming languages, compilation, syntax and semantics
- values and effects, fundamental types, literals, variables

Ⓛ
- include directive `#include <iostream>`
- main function `int main(){...}`
- comments, layout `// Kommentar`
- types, variables, L-value `a` , R-value `a+b`
- expression statement `b=b*b;` , declaration statement `int a;`, return statement `return 0;`

2. Integers

Ⓜ
- Celsius to Fahrenheit

Ⓒ
- associativity and precedence, arity
- expression trees, evaluation order
- arithmetic operators
- binary representation, hexadecimal numbers
- signed numbers, twos complement

Ⓛ
- arithmetic operators `9 * celsius / 5 + 32`
- increment / decrement `expr++`
- arithmetic assignment `expr1 += expr2`
- conversion `int` ↔ `unsigned int`

Ⓔ
- Celsius to Fahrenheit, equivalent resistance

3. Booleans

C

- Boolean functions, completeness
- DeMorgan rules

L

- the type `bool`
- logical operators `a && !b`
- relational operators `x < y`
- precedences `7 + x < y && y != 3 * z`
- short circuit evaluation `x != 0 && z / x > y`
- the `assert`-statement, `#include <cassert>`

E

- Div-Mod identity.

4./5. Control Statements

M

- linear control flow vs. interesting programs

C

- selection statements, iteration statements
- (avoiding) endless loops, halting problem
- Visibility and scopes, automatic memory
- equivalence of iteration statement

L

- if statements `if (a % 2 == 0) {..}`
- for statements `for (unsigned int i = 1; i <= n; ++i) ...`
- while and do-statements `while (n > 1) {...}`
- blocks and branches `if (a < 0) continue;`

E

- sum computation (Gauss), prime number tests, Collatz sequence, Fibonacci numbers, calculator

6./7. Floating Point Numbers

M

- correct computation: Celsius / Fahrenheit

C

- fixpoint vs. floating point
- holes in the value range
- compute using floating point numbers
- floating point number systems, normalisation, IEEE standard 754
- *guidelines for computing with floating point numbers*

L

- types `float`, `double`
- floating point literals `1.23e-7f`

E

- Celsius/Fahrenheit, Euler, Harmonic Numbers

8./9. Functions

M

- Computation of Powers

C

- Encapsulation of Functionality
- functions, formal arguments, arguments
- scope, forward declarations
- procedural programming, modularization, separate compilation
- *Stepwise Refinement*

L

- declaration and definition of functions `double pow(double b, int e){ ... }`
- function call `pow (2.0, -2)`
- the type `void`

E

- powers, perfect numbers, minimum, calendar

10. Reference Types

Ⓜ
- Swap

Ⓒ
- value- / reference- semantics, call by value, call by reference
- lifetime of objects / temporary objects
- constants

Ⓛ
- reference type `int& a`
- call by reference, return by reference `int& increment (int& i)`
- const guideline, const references, reference guideline

Ⓔ
- swap, increment

11./12. Arrays

Ⓜ
- Iterate over data: array of erathosthenes

Ⓒ
- arrays, memory layout, random access
- (missing) bound checks
- vectors
- characters: ASCII, UTF8, texts, strings

Ⓛ
- array types `int a[5] = {4,3,5,2,1};`
- characters and texts, the type char `char c = 'a';`, Konversion nach **int**
- multi-dimensional arrays, vectors of vectors

Ⓔ
- sieve of Erathosthenes, Caesar-code, shortest paths, Lindenmayer systems

13./14. Pointers, Iterators and Containers

Ⓜ
- arrays as function arguments

Ⓒ
- pointers, chances and dangers of indirection
- random access vs. iteration, pointer arithmetics
- containers and iterators

Ⓛ
- pointer `int* x;`, conversion array → pointer, null-pointer
- address and derference operator `int *ip = &i; int j = *ip;`
- pointer and const `const int *a;`
- algorithms and iterators `std::fill (a, a+5, 1);`
- type definitions `typedef std::set<char>::const_iterator Sit;`

Ⓔ
- filling an array, character salad

15./16. Recursion

Ⓜ
- recursive mathe. functions

Ⓒ
- recursion
- call stack, memory of recursion
- correctness, termination,
- recursion vs. iteration
- EBNF, formal grammars, streams, parsing
- evaluation, associativity

Ⓔ
- factorial, GCD, Fibonacci, mountains

17. Structs and Classes I

Ⓜ
- build your own rational number

Ⓒ
- heterogeneous data types
- function and operator overloading
- encapsulation of data

Ⓛ
- struct definition `struct rational {int n; int d;};`
- member access `result.n = a.n * b.d + a.d * b.n;`
- initialization and assignment,
- function overloading `pow(2)` vs. `pow(3,3);`, operator overloading

Ⓔ
- rational numbers, complex numbers

18. Classes, Dynamic Data Types

Ⓜ
- rational numbers with encapsulation, stack

Ⓒ
- linked list, allocation, deallocation, dynamic data type

Ⓛ
- classes `class rational { ... };`
- access control `public:` / `private:`
- member functions `int rational::denominator () const`
- copy constructor, destructor, rule of three
- constructors `rational (int den, int nm): d(den), n(no) {}`
- **`new`** and **`delete`**
- copy constructor, assignment operator, destructor

Ⓔ
- linked list, stack

19. Tree Structures, Inheritance and Polymorphism

Ⓜ
- expression trees,
- extension of expression trees
- inheritance

Ⓒ
- trees
- inheritance
- polymorphism

Ⓛ
- inheritance `class tree_node: public number_node`
- virtual functions `virtual void size() const;`

Ⓔ
- expression tree, expression parsing, extension by abs-node

The End

End of the Course