

Computer Science

Course for Computational Sciences and Engineering
at D-MATH of ETH Zurich

Felix Friedrich, Malte Schwerhoff

AS 2018

Welcome

to the Course Informatik

for CSE at the MAVT department of ETH Zürich.

Place and time:

Monday 08:15 - 10:00, CHN C 14.

Pause 9:00 - 9:15, slight shift possible.

Course web page

http://lec.inf.ethz.ch/math/informatik_cse

Team

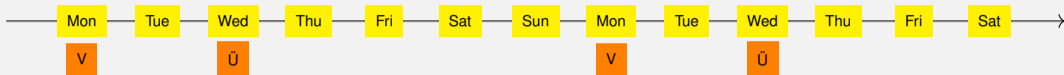
chef assistant Francois Serre

back office Lucca Hirschi

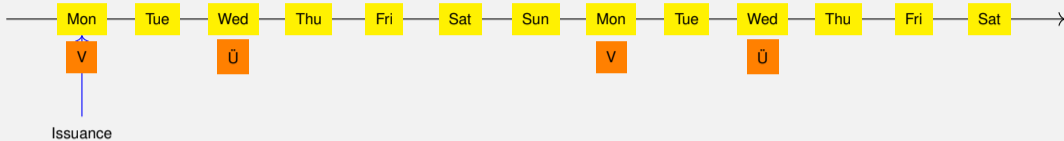
assistants Manuel Kaufmann
 Robin Worreby
 Roger Barton
 Sebastian Balzer

lecturers Dr. Felix Friedrich / Dr. Malte Schwerhoff

Procedure

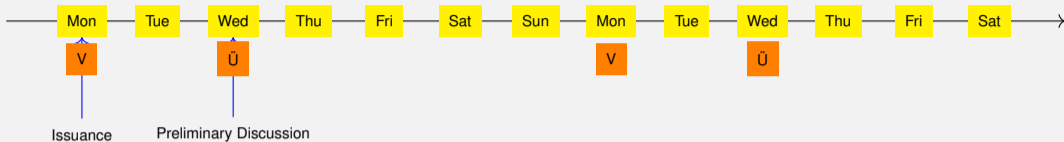


Procedure



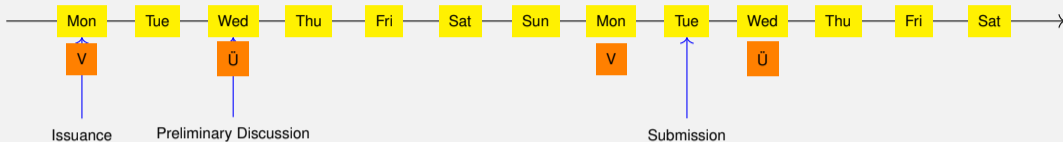
- Exercises available at lectures.
- Preliminary discussion in the following recitation session
- Solution of the exercise until the day before the next recitation session.
- Discussion of the exercise in the next recitation session.

Procedure



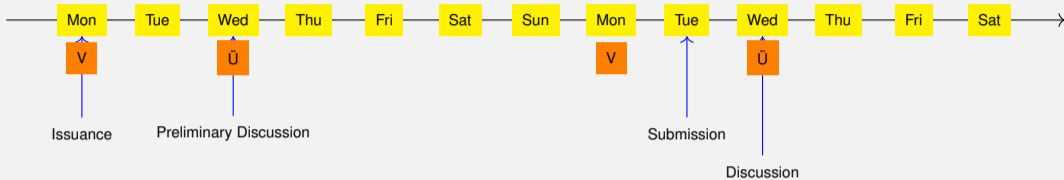
- Exercises available at lectures.
- Preliminary discussion in the following recitation session
- Solution of the exercise until the day before the next recitation session.
- Discussion of the exercise in the next recitation session.

Procedure



- Exercises available at lectures.
- Preliminary discussion in the following recitation session
- **Solution of the exercise until the day before the next recitation session.**
- Discussion of the exercise in the next recitation session.

Procedure



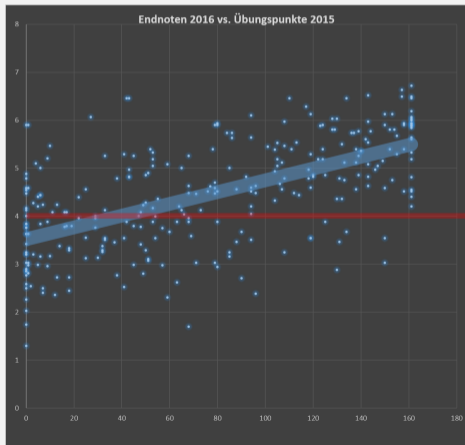
- Exercises available at lectures.
- Preliminary discussion in the following recitation session
- Solution of the exercise until the day before the next recitation session.
- Discussion of the exercise in the next recitation session.

Exercises

- The solution of the weekly exercises is thus voluntary but *strongly* recommended.

Exercises

- The solution of the weekly exercises is thus voluntary but *strongly* recommended.



Exams

The exam (in examination period 2019) will cover

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

Exams

Written exam that most probably takes place at a computer (for the CSE students).

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

Offer

- During the semester we offer weekly programming exercises that are graded. Points achieved will be taken as a bonus to the exam.
- The bonus is proportional to the score achieved in specially marked bonus tasks, where a full score equals a bonus of 0.25. The admission to specially marked bonus depends on the successful completion of other exercises. The achieved mark bonus expires as soon as the lecture is given anew.

Offer (Concretely)

- 3 bonus exercises in total; 2/3 of the points suffice for the exam bonus of 0.25 marks
- You can, e.g. fully solve 2 bonus exercises, or solve 3 bonus exercises to 66% each, or ...
- Bonus exercises must be unlocked (\rightarrow experience points) by successfully completing the weekly exercises
- It is again not necessary to solve all weekly exercises completely in order to unlock a bonus exercise
- Details: course website, exercise sessions, online exercise system (Code Expert)

Academic integrity

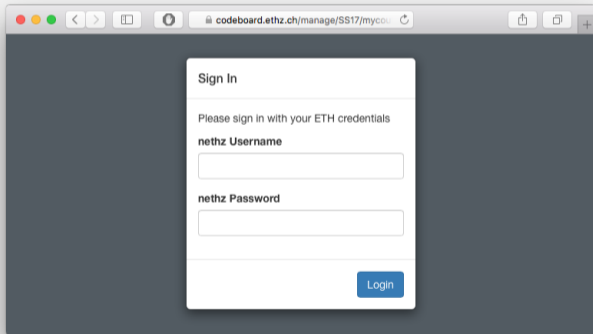
Rule: You submit solutions that you have written yourself and that you have understood.

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.

Exercise group registration I

- Visit `http://expert.ethz.ch/enroll/AS18/infocse`
- Log in with your nethz account.

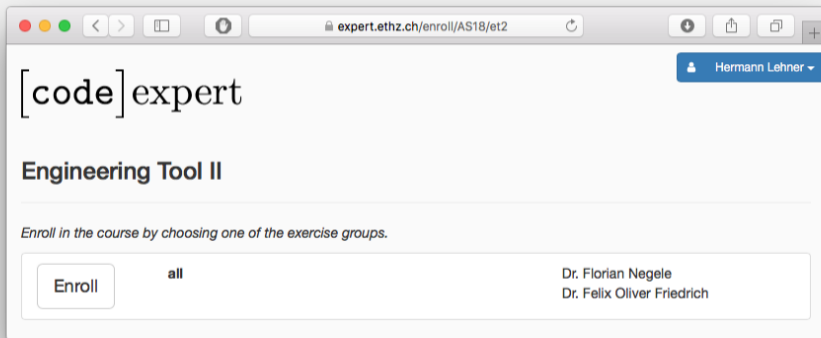


The image shows a browser window with the address bar containing `codeboard.ethz.ch/manage/SS17/mycode`. The main content area displays a 'Sign In' form with the following elements:

- Sign In** (Section Header)
- Please sign in with your ETH credentials
- nethz Username** (Label) followed by an empty text input field.
- nethz Password** (Label) followed by an empty password input field.
- Login** (Submit Button)

Exercise group registration II

Register with the subsequent dialog for an exercise group.



The screenshot shows a web browser window with the URL `expert.ethz.ch/enroll/AS18/et2`. The page content includes the logo `[code]expert`, the course title **Engineering Tool II**, and a user profile for Hermann Lehner. Below the course title, there is a message: *Enroll in the course by choosing one of the exercise groups.* At the bottom, there is a registration dialog with an **Enroll** button, the text **all**, and the names of the lecturers: **Dr. Florian Negele** and **Dr. Felix Oliver Friedrich**.

Overview

[code]expert

Felix Oliver Friedrich ▾

Autum 2017 ▾

Enrolled Courses

My Exercise Groups

My Courses

Demo Course Demo Group - Dr. Hermann Lehner [change...](#)

Coding Demo Exercise

Earned XP

Submissions

Handout Date

Due Date

Tasks

Solutions

1,000 / 1,000

9. Sep. 2017 00:00

31. Dez. 2027 00:00

Quadratic Equations In C++

1,000 ✓

100%

Hand In now

Markdown Editor Manual

Submissions

Handout Date

Due Date

Tasks

Solutions

1. Aug. 2017 00:00

1. Aug. 2017 00:01

Basic Markdown Syntax

Code Blocks and Inline Code

Programming Exercise

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min;
6     max = min;
7     for (int i = 0; i < 8; ++i){ // (there is a bug here)
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << max << std::endl;
14 }
```

A

B

C

>_ Console

Felix Oliver Friedrich

Status Not submitted yet

Create new Submission

Minimax

Write a program that outputs the minimum and maximum of a series of ten integers.

- Input format: 10 consecutive integers
number:int, example:

```
0 100250 45 0 0 1 -1000001 45 -25065 1
```

- Expected output format: minimum:int
"/" maximum:int, example:

```
-1000001/100251
```

Task History

Programming Exercise

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min;
6     max = min;
7     for (int i = 0; i < 8; ++i){ // (there is a bug here)
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << max << std::endl;
14 }
```

A
B
C

A: compile
B: run
C: test

>_ Console

Felix Oliver Friedrich

Status Not submitted yet

Create new Submission

Minimax

Write a program that outputs the minimum and maximum of a series of ten integers.

- Input format: 10 consecutive integers
number:int, example:
0 100250 45 0 0 1 -1000001 45 -25065 1
- Expected output format: minimum:int
"/" maximum:int, example:
-1000001/100251

Programming Exercise

The image shows a programming IDE interface with a code editor on the left and a task description on the right. The code editor displays a C++ program for finding the minimum and maximum of ten integers. The task description on the right includes the user's name (Felix Oliver Friedrich), the status (Not submitted yet), and the problem description. A red box highlights the task description and history buttons. A semi-transparent pink box contains the text 'D: description' and 'E: History'. At the bottom left, three icons (gear, play, flask) are labeled A, B, and C respectively.

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min;
6     max = min;
7     for (int i = 0; i < 8; ++i){ // (there is
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << max << std::endl;
14 }
```

D: description
E: History

Felix Oliver Friedrich
Status Not submitted yet

maximum and
maximum of a series of ten integers.

- Input format: 10 consecutive integers
number: int, example:
0 100250 45 0 0 1 -1000001 45 -25065 1
- Expected output format: minimum: int
"/" maximum: int, example:
-1000001/100251

A
B
C

>_ Console

Test and Submit

The screenshot displays a development environment with three main sections:

- Project Files:** Shows a file named `main.cpp`.
- Code Editor:** Contains the following C++ code:

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min; std::cin >> max;
6     max = min-1;
7     for (int i = 0; i < 8; ++i){
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << max << std::endl;
14 }
```
- Console:** Shows test results:

```
Running tests.....
min_first passed
min_last passed
min_middle passed
max_first failed
input:
100251 -25065 45 -1000001 1 0 0 45 100250 0
expected output:
-1000001/100251
actual output:
-1000001/100250
-----
max_last passed
max_middle passed
unique passed

Tests result: passed 6 of 7 / score: 86% [ ]
```
- Submission Panel:** Shows the user `Felix Oliver Friedrich` with a status of `Not submitted yet`. It includes a `Create new Submission` button and options to filter snapshots, view the first working version (2 minutes ago), and view the initial snapshot (13 minutes ago).

Test and Submit

The screenshot displays a development environment with three main sections:

- Project Files:** Shows a file named `main.cpp`.
- Minimax - Student Attempt:** Contains the following C++ code:

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min; std::cin >> max;
6     max = min-1;
7     for (int i = 0; i < 8; ++i){
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << max << std::endl;
14 }
```
- Console:** Shows test results:

```
Running tests.....
min_first passed
min_last passed
min_middle passed
max_first failed
input:
100251 -25065 45 -1000001 1 0 0 45 100250 0
expected output:
-1000001/100251
actual output:
-1000001/100250
-----
max_last passed
max_middle passed
unique passed

Tests result: passed 6 of 7 / score: 86% [ ]
```
- User Interface:** Shows the user `Felix Oliver Friedrich` with a status of `Status Not submitted yet`. It includes a `Create new Submission` button and a `Create Snapshot` button. Below these are sections for `First Working Version` (2 minutes ago) and `Initial Snapshot` (13 minutes ago), each with a refresh and download icon.

Test and Submit

The screenshot displays a code editor interface with the following components:

- Project Files:** A sidebar on the left showing a file named `main.cpp`.
- Code Editor:** The main area contains C++ code for finding the minimum and maximum values in an array. The code is as follows:

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min; std::cin >> max;
6     max = min-1;
7     for (int i = 0; i < 8; ++i){
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << max << std::endl;
14 }
```
- Test Results:** Below the code, a console window shows the output of running tests. A red arrow points to the word "Test" in a pink box. The test results are:

```
Running tests.....
min_first passed
min_last passed
min_middle passed
max_first failed
input:
100251 -25065 45 -1000001 1 0 0 45 100250 0
expected output:
-1000001/100251
actual output:
-1000001/100250
-----
max_last passed
max_middle passed
unique passed

Tests result: passed 6 of 7 / score: 86% [ ]
```
- Submission Panel:** On the right side, there is a user profile for "Felix Oliver Friedrich" with a status of "Not submitted yet". A pink box labeled "Submission" has a red arrow pointing to a green button that says "Create new Submission". Below this are sections for "Filter Snapshots" (with a "Create Snapshot" button), "First Working Version" (2 minutes ago), and "Initial Snapshot" (13 minutes ago).

Where is the Save Button?

- The file system is transaction based and is saved permanently (“autosave”). When opening a project it is found in the most recent observed state.
- The current state can be saved as (named) *snapshot*. It is always possible to return to saved snapshot.
- The current state can be submitted (as snapshot). Additionally, each saved named snapshot can be submitted.

Snapshots

The screenshot displays a code editor interface with a dark theme. On the left, a file explorer shows 'Project Files' containing 'main.cpp'. The main editor area shows a C++ program named 'Minimax - Student Attempt' with the following code:

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min; std::cin >> max;
6     max = min;
7     for (int i = 0; i < 8; ++i){ // (there is a bug here)
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << max << std::endl;
14 }
```

Below the code, a console window shows the output of running tests:

```
Running tests.....
min_first passed
min_last passed
min_middle passed
max_first passed
max_last passed
max_middle passed
unique passed

Tests result: passed 7 of 7 / score: 100% [██████████]
[]
```

On the right side, a 'History' panel is visible. It shows the user 'Felix Oliver Friedrich' and the status 'Already submitted'. A green button 'Create new Submission' is present. Below this, a 'Filter Snapshots' section includes a 'Create Snapshot' button and three snapshot entries:

- Really Working Version (2 minutes ago)
- First Working Version (6 minutes ago)
- Initial Snapshot (16 minutes ago)

Each entry has a magnifying glass icon and a download icon. At the top right of the history panel, there is a 'History Go back to current version' link with a checkmark icon.

Snapshots

The screenshot displays a code editor interface with the following components:

- Project Files:** A sidebar on the left showing a folder named "Project Files" containing a file named "main.cpp".
- Code Editor:** The main area shows a C++ program titled "Minimax - Student Attempt". The code is as follows:

```
1 #include <iostream>
2
3 int main () {
4     int min; int max;
5     std::cin >> min; std::cin >> max;
6     max = min;
7     for (int i = 0; i < 8; ++i){ // (there is a bug here)
8         int v;
9         std::cin >> v;
10        if (v<min) min = v;
11        if (v>max) max = v;
12    }
13    std::cout << min << "/" << m
14 }
```
- Test Results:** Below the code, a console window shows the output of running tests:

```
Running tests.....
min_first passed
min_last passed
min_middle passed
max_first passed
max_last passed
max_middle passed
unique passed

Tests result: passed 7 of 7 / score: 100% [ ]
[]
```
- History Panel:** On the right side, a panel titled "History" shows the user "Felix Oliver Friedrich" and the status "Already submitted". It includes a "Create new Submission" button and a "Filter Snapshots" section with a "Create Snapshot" button. Below this, three snapshots are listed:
 - Really Working Version (2 minutes ago)
 - First Working Version (6 minutes ago)
 - Initial Snapshot (16 minutes ago)
- Callout:** A pink rectangular box with the text "Look at snapshot" is overlaid on the code editor. A red arrow points from this box to the "First Working Version" entry in the history panel.

Snapshots

The screenshot displays a code editor interface with a dark theme. On the left, a file explorer shows 'Project Files' containing 'main.cpp'. The main editor area shows C++ code for a 'Minimax - Student Attempt' with line numbers 1-14. The code includes `<iostream>`, defines `int main()`, and uses `std::cin` and `std::cout`. A comment on line 7 says '// (there is a bug here)'. Below the code, a console window shows test results: 'Running tests.....', followed by 'min_first passed', 'min_last passed', 'min_middle passed', 'max_first passed', 'max_last passed', 'max_middle passed', and 'unique passed'. A summary line reads 'Tests result: passed 7 of 7 / score: 100%' with a green progress bar.

On the right side, a sidebar contains a 'History' section. At the top, it shows the user 'Felix Oliver Friedrich' and the status 'Already submitted'. A green button says 'Create new Submission'. Below this is a 'Filter Snapshots' section with a 'Create Snapshot' button. A list of snapshots is shown: 'Really Working Version' (2 minutes ago), 'First Working Version' (6 minutes ago), and 'Initial Snapshot' (16 minutes ago). Each snapshot entry has a 'Go Back' icon (a left-pointing arrow) and a 'Submission' icon (a download arrow).

Three red annotations with arrows point to these icons:

- A box labeled 'Look at snapshot' has an arrow pointing to the 'Go Back' icon of the 'First Working Version'.
- A box labeled 'Submission' has an arrow pointing to the 'Submission' icon of the 'Initial Snapshot'.
- A box labeled 'Go Back' has an arrow pointing to the 'Go Back' icon of the 'Initial Snapshot'.

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?

What is Computer Science?

- The science of **systematic processing of informations**,...

What is Computer Science?

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

(Wikipedia, according to “Duden Informatik”)

Computer Science vs. Computers

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

Mike Fellows, US Computer Scientist (1991)

Computer Science vs. Computers

- Computer science is also concerned with the development of fast computers and networks. . .

Computer Science vs. Computers

- 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 \neq Computer Literacy

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 C++.
- Hence:
not only
but also programming course.

Algorithm: Fundamental Notion of Computer Science

Algorithm:

- Instructions to solve a problem step by step

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)

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)

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 .



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 $b \neq 0$

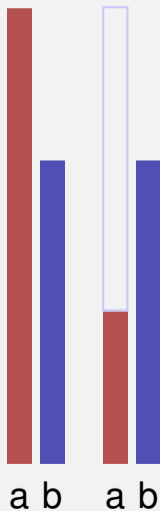
 If $a > b$ then

$$a \leftarrow a - b$$

 else:

$$b \leftarrow b - a$$

Result: a .



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 $b \neq 0$

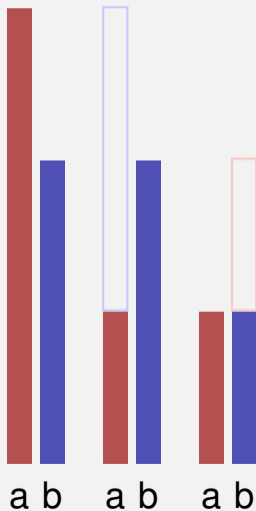
 If $a > b$ then

$$a \leftarrow a - b$$

 else:

$$b \leftarrow b - a$$

Result: a .



Oldest Nontrivial Algorithm

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

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

While $b \neq 0$

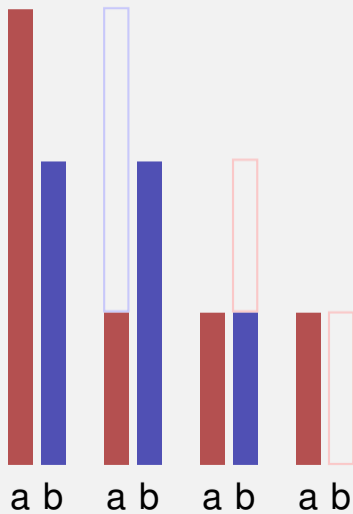
 If $a > b$ then

$$a \leftarrow a - b$$

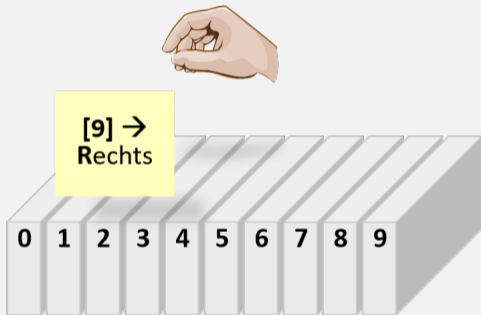
 else:

$$b \leftarrow b - a$$

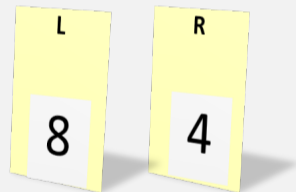
Result: a .



Live Demo: Turing Machine



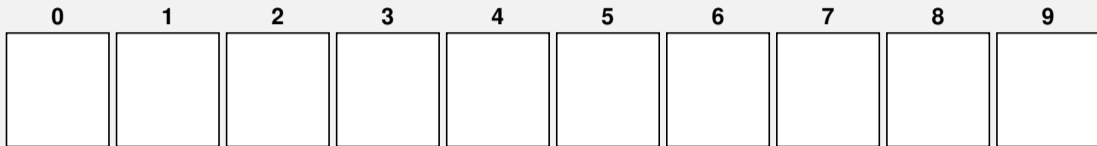
Speicher



Register

Euklid in the Box

Speicher



Links

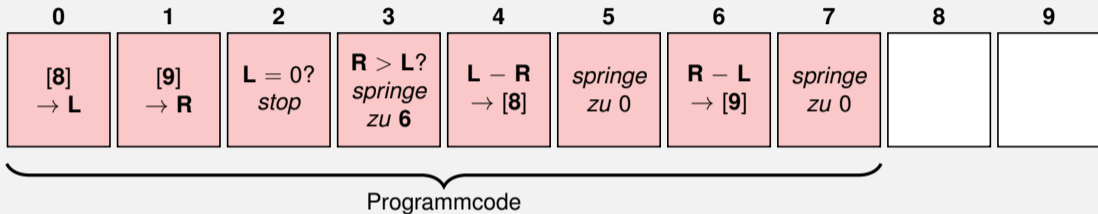
Rechts



Register

Euklid in the Box

Speicher



Links

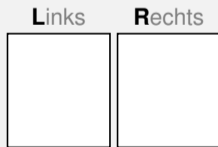
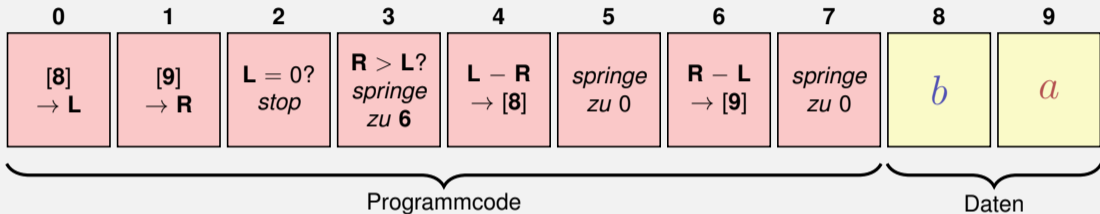
Rechts



Register

Euklid in the Box

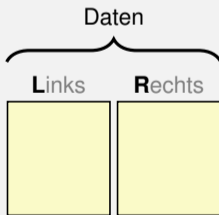
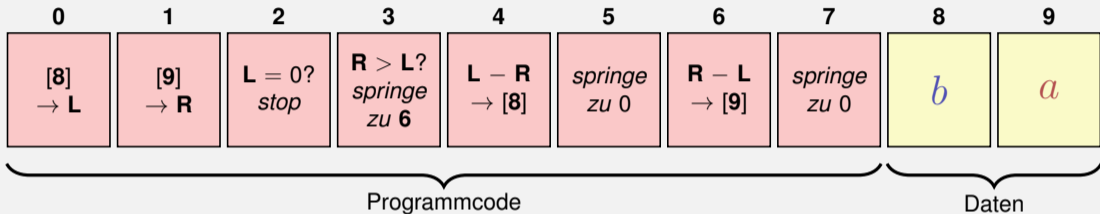
Speicher



Register

Euklid in the Box

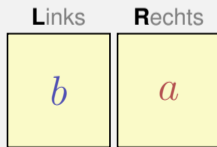
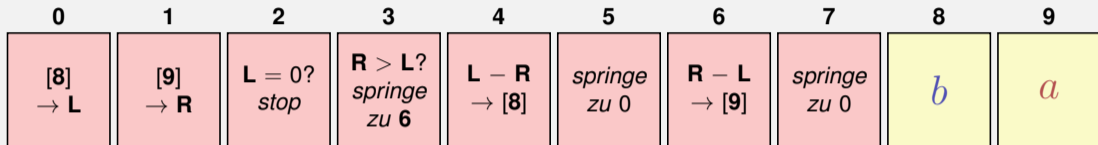
Speicher



Register

Euklid in the Box

Speicher



Register

While $b \neq 0$

 If $a > b$ then

$a \leftarrow a - b$

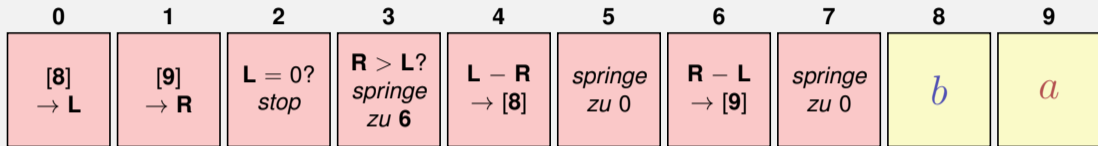
 else:

$b \leftarrow b - a$

Ergebnis: a .

Euklid in the Box

Speicher



While $b \neq 0$

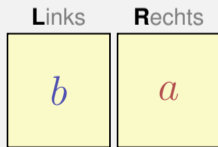
If $a > b$ then

$a \leftarrow a - b$

else:

$b \leftarrow b - a$

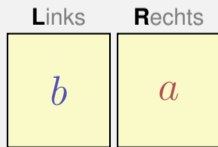
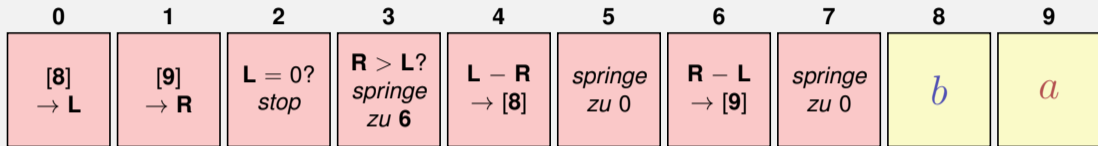
Ergebnis: a .



Register

Euklid in the Box

Speicher



Register

While $b \neq 0$

If $a > b$ then

$a \leftarrow a - b$

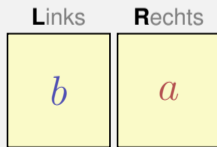
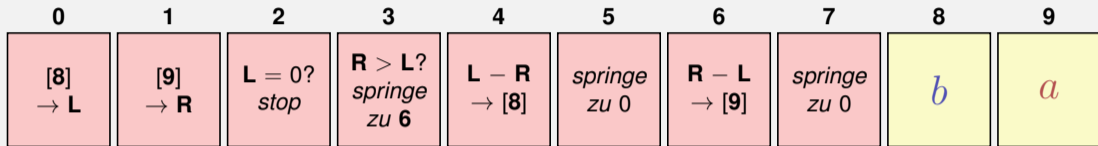
else:

$b \leftarrow b - a$

Ergebnis: a .

Euklid in the Box

Speicher



Register

While $b \neq 0$

If $a > b$ then

$a \leftarrow a - b$

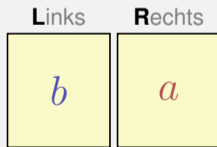
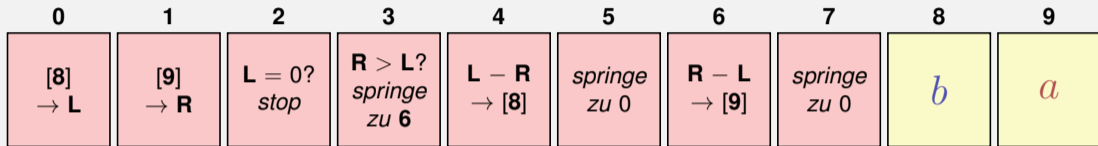
else:

$b \leftarrow b - a$

Ergebnis: a .

Euklid in the Box

Speicher

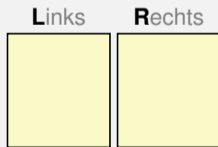
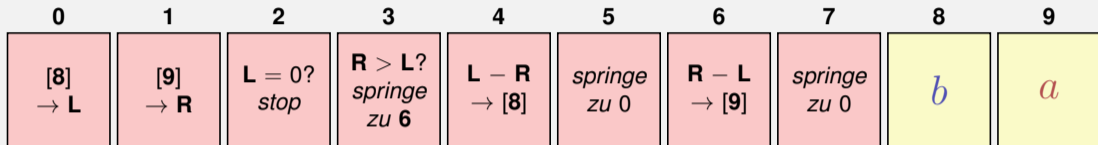


Register

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

Euklid in the Box

Speicher



Register

While $b \neq 0$

 If $a > b$ then

$a \leftarrow a - b$

 else:

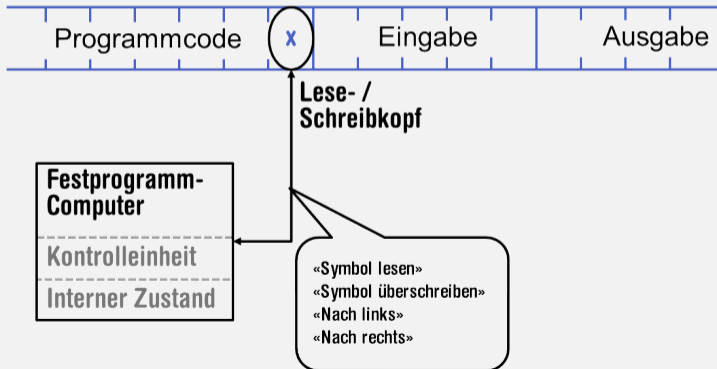
$b \leftarrow b - a$

Ergebnis: a .

Computers – Concept

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

Folge von Symbolen auf Ein- und Ausgabeband

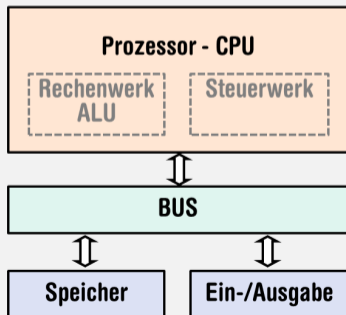


Alan Turing

Computer – Implementation

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

Von Neumann Architektur



Konrad Zuse



John von Neumann

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)

Memory for data *and* program

- Every memory cell has an address.
- Random access: access time to the memory cell is (nearly) independent of its address.



Computing speed

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

¹Uniprocessor computer at 1 GHz.

Computing speed

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



a contemporary desktop PC can process more than 100

¹Uniprocessor computer at 1 GHz.

Computing speed

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



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

a contemporary desktop PC can process more than 100 millions instructions ¹

¹Uniprocessor 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

Why programming?

- Do I study computer science or what ...

Why programming?

- Do I study computer science or what ...
- There are programs for everything ...

Why programming?

- Do I study computer science or what ...
- There are programs for everything ...
- I am not interested in programming ...

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...

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

((BTW: Lino Guzzella is not a computer scientist, he is a mechanical engineer and prof. for thermotronics 😊))

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)

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.

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 (and is useful)!

Programming Languages

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

Higher Programming Languages

can be represented as program text that

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

Why C++?

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

Why C++?

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

General consensus:

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

Why C++?

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

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.

Syntax and Semantics

- Corresponding rules for a computer program are simpler but also more strict because computers are relatively stupid.

Syntax and Semantics of C++

Syntax

- When is a text a *C++ program*?
- I.e. is it *grammatically* correct?

Semantics

- What does a program *mean*?
- Which algorithm does a program *implement*?

Programming Tools

- **Editor:** Program to modify, edit and store C++ program texts
- **Compiler:** program to translate a program text into machine language

Programming Tools


- **Computer:** machine to execute machine language programs
- **Operating System:** program to organize all procedures such as file handling, editor-, compiler- and program execution.

The first C++ program

```
// 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;
}
```

Most important ingredients...

Statements

```
// 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;  Do something (read in 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;
}
```

```
// 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 ←—— Compute a value (a^2)!
    b = b * b;     // b = a^4
    // output b * b, i.e., a^8
    std::cout << a << "^8 = " << b * b << "\n";
    return 0;
}
```

“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;
}
```

“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

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;}
```

Comments and Layout

The compiler does not care...

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

... but we do!

“Accessories:” Include and Main Function

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


“Accessories:” Include and Main Function

```
// Program: power8.cpp
// Raise a number to the eighth power.
#include <iostream> ← include directive
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;
}
```

“Accessories:” Include and Main Function

```
// Program: power8.cpp
// Raise a number to the eighth power.
#include <iostream>
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;
}
```

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;  
}
```

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

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; ← return statement  
}
```

Statements – Effects

```
int main() {  
    // input  
    std::cout << "Compute a^8 for a =? ";  
    int a;  
    std::cin >> a;  
    // computation  
    int b = a * a;  
    b = b * b;  
    // 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^2 into b

Effect: saving the computed value of b^2 into b

Effect: return the value 0

Effect: output of the value of a and the computed value of a^8

Statements – Variable Definitions

```
int main() {  
    // input  
    std::cout << "Compute a^8 for a =? ";  
    int a; ← declaration statement  
    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;  
}
```

type names

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*

Variables

- represent (varying) values
- have
 - *name*
 - *type*
 - *value*
 - *address*

Example

`int a;` defines a variable with

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

Expressions: compute a value!

- represent *Computations*

Expressions: compute a value!

- represent *Computations*
- are either **primary** (b)

Expressions: compute a value!

- represent *Computations*
- are either **primary** (b)
- or **composed** (b*b)...

Expressions: compute a value!

- represent *Computations*
- are either **primary** (b)
- or **composed** ($b*b$)...
- ... from different expressions, using **operators**

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

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

```
// 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;
```

Expressions

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

variable name, primary expression (+ name and address)

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

variable name, primary expression (+ name and address)

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

```
return 0;
```

literal, primary 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; // b = a^4

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

return 0;
```

composite expression

composite expression

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

```
// computation
```

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

```
b = b * b; ← Two times composed expression
```

```
// output b * b, i.e., a^8
```

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

↑
return (Four times composed expression

L-Values and R-Values

```
// 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;
```

L-Values and R-Values

```
// input
std::cout << "Compute a^8 for a =? ";
int a;
std::cin >> a;  ← L-value (expression + address)

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

// output b * b, i.e., a^8
std::cout << a << "^8 = " << b * b << ".\n";
return 0;      ← R-Value (expression that is not an L-value)
```

L-Values and R-Values

```
// 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;
```

The image illustrates L-values and R-values in C++ code. Red boxes highlight the expressions `"Compute a^8 for a =? "`, `b * b` (in the assignment `b = b * b;`), and `b * b` (in the output statement). Red arrows labeled "R-Value" point to these expressions, indicating they are R-values. The text `// output b * b, i.e., a^8` is also present.

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-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

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)

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)

Every E-Bike can be used as normal bike, but not the other way round

L-Values and R-Values

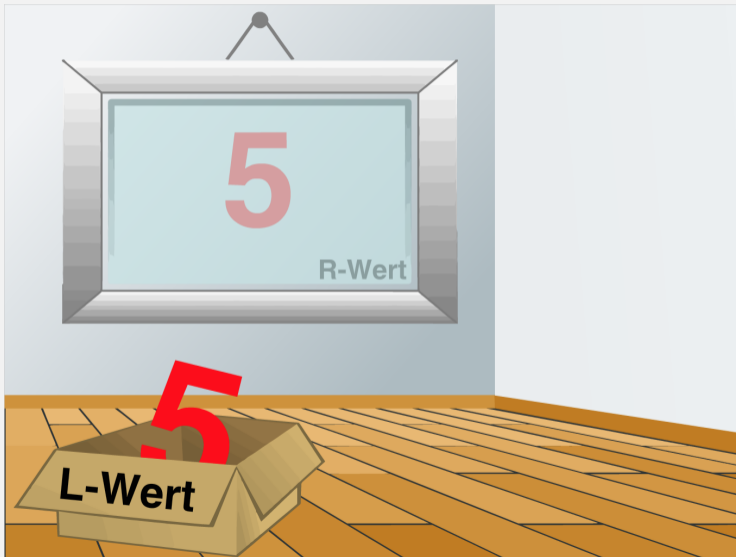
R-Wert (“**R**ight of the assignment operator”)

- 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

L-Value and R-Value



```
// 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;
```

Operators and Operands

Building Blocks

```
// 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;
```

Diagram annotations:

- left operand (output stream) points to `std::cout`
- output operator points to `<<`
- right operand (string) points to `"Compute a^8 for a=? "`


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

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

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

Diagram illustrating the components of the input statement `std::cin >> a;`:

- `std::cin`: left operand (input stream)
- `>>`: input operator
- `a`: right operand (variable name)

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

assignment operator

multiplication operator

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;
}
```

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,

9 * celsius / 5 + 32

- Arithmetic expression,
- **three literals**, one variable, three operator symbols

9 * celsius / 5 + 32

- Arithmetic expression,
- three literals, **one variable**, three operator symbols

9 * celsius / 5 + 32

- Arithmetic expression,
- three literals, one variable, three operator symbols

9 * celsius / 5 + 32

- Arithmetic expression,
- three literals, one 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`

Precedence

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`

Associativity

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

of the operands in an unambiguous way.

Expression Trees

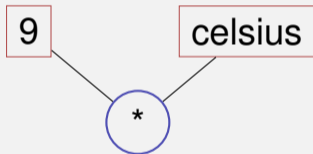
Parentheses yield the expression tree

`9 * celsius / 5 + 32`

Expression Trees

Parentheses yield the expression tree

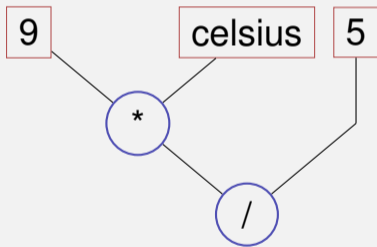
`(9 * celsius) / 5 + 32`



Expression Trees

Parentheses yield the expression tree

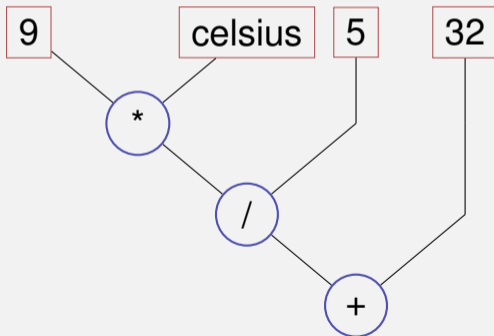
`((9 * celsius) / 5) + 32`



Expression Trees

Parentheses yield the expression tree

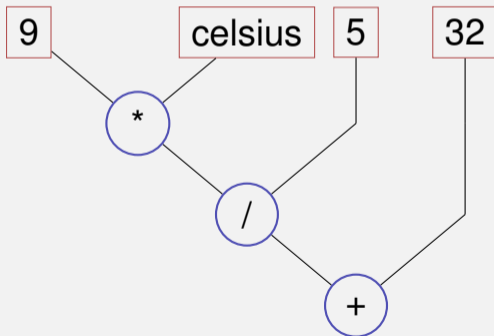
`((9 * celsius) / 5) + 32)`



Evaluation Order

"From top to bottom" in the expression tree

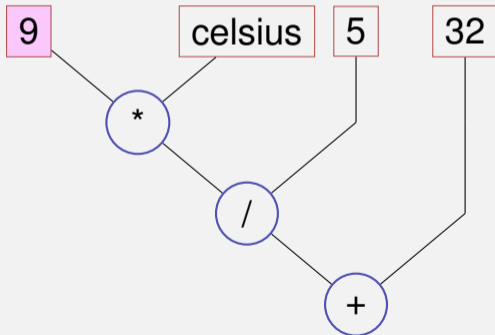
9 * celsius / 5 + 32



Evaluation Order

"From top to bottom" in the expression tree

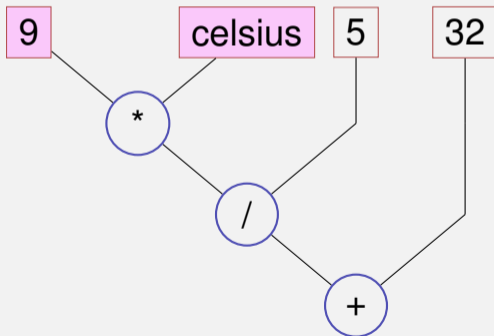
9 * celsius / 5 + 32



Evaluation Order

"From top to bottom" in the expression tree

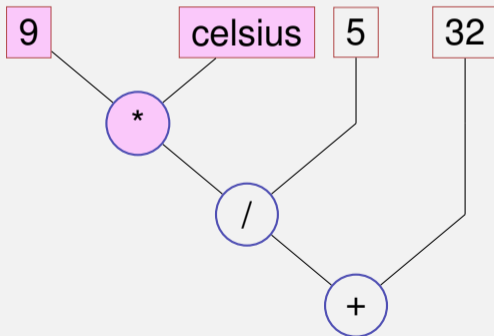
9 * celsius / 5 + 32



Evaluation Order

"From top to bottom" in the expression tree

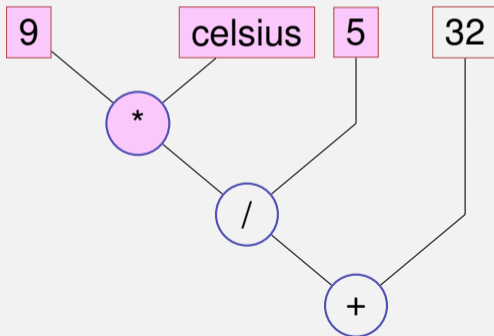
9 * celsius / 5 + 32



Evaluation Order

"From top to bottom" in the expression tree

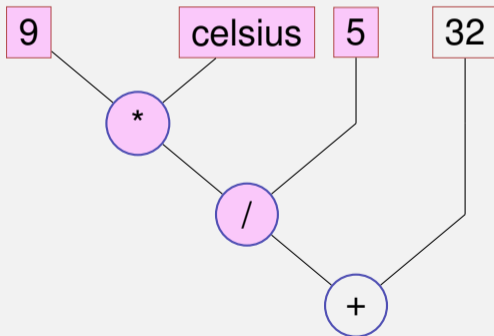
9 * celsius / 5 + 32



Evaluation Order

"From top to bottom" in the expression tree

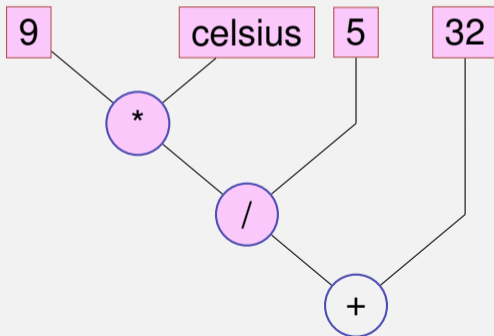
9 * celsius / 5 + 32



Evaluation Order

"From top to bottom" in the expression tree

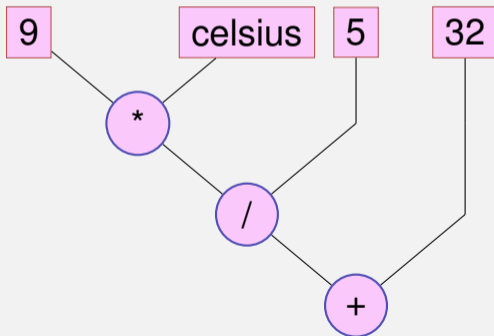
9 * celsius / 5 + 32



Evaluation Order

"From top to bottom" in the expression tree

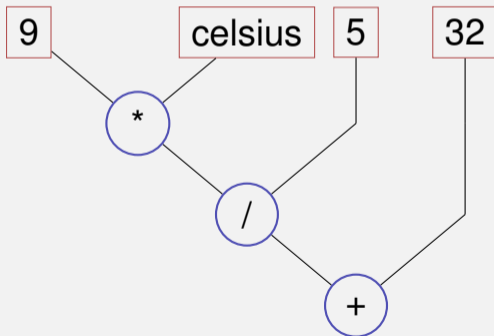
9 * celsius / 5 + 32



Evaluation Order

Order is not determined uniquely:

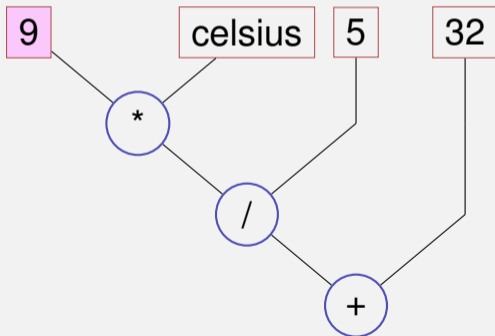
`9 * celsius / 5 + 32`



Evaluation Order

Order is not determined uniquely:

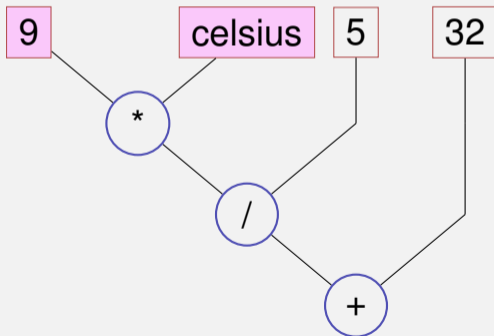
`9 * celsius / 5 + 32`



Evaluation Order

Order is not determined uniquely:

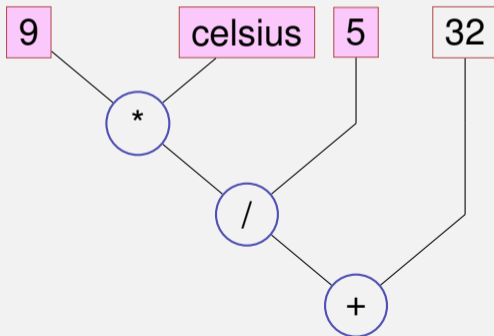
`9 * celsius / 5 + 32`



Evaluation Order

Order is not determined uniquely:

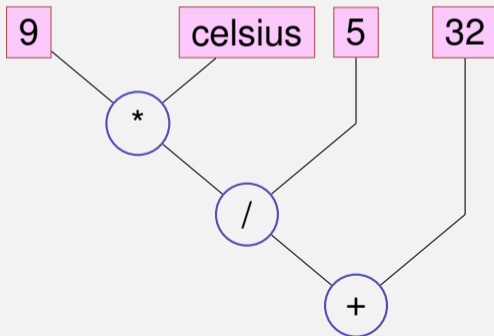
`9 * celsius / 5 + 32`



Evaluation Order

Order is not determined uniquely:

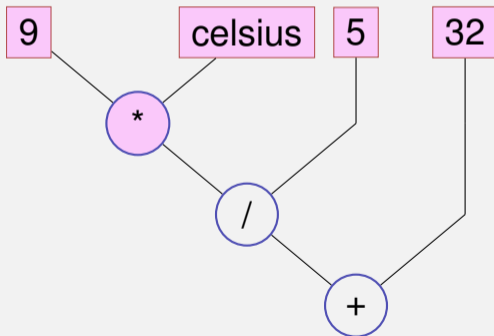
`9 * celsius / 5 + 32`



Evaluation Order

Order is not determined uniquely:

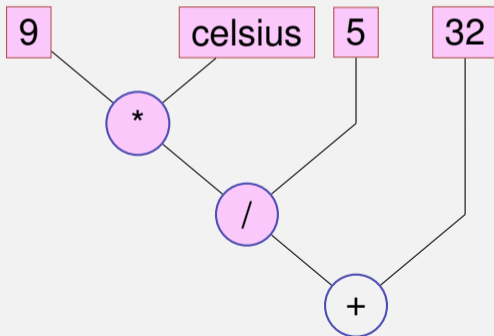
`9 * celsius / 5 + 32`



Evaluation Order

Order is not determined uniquely:

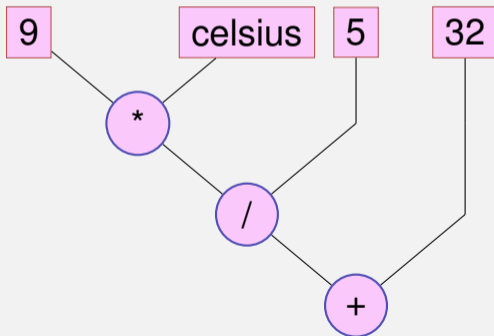
`9 * celsius / 5 + 32`



Evaluation Order

Order is not determined uniquely:

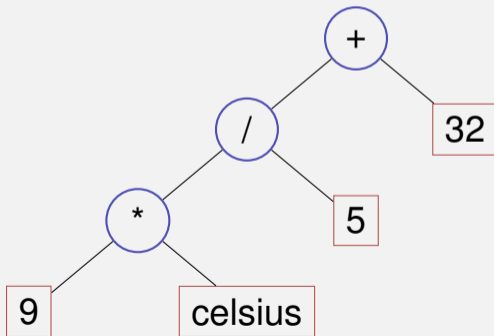
`9 * celsius / 5 + 32`



Expression Trees – Notation

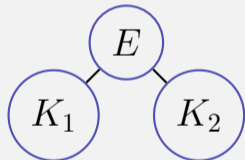
Common notation: root on top

9 * celsius / 5 + 32



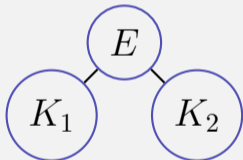
Evaluation Order – more formally

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



Evaluation Order – more formally

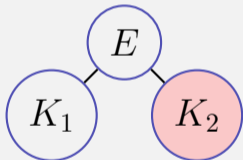
- Valid order: any node is evaluated *after* its children



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

Evaluation Order – more formally

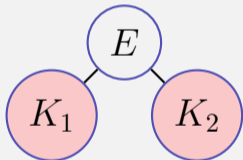
- Valid order: any node is evaluated *after* its children



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

Evaluation Order – more formally

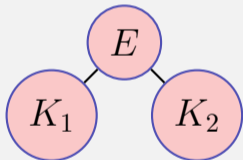
- Valid order: any node is evaluated *after* its children



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

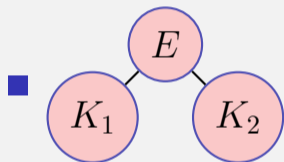
Evaluation Order – more formally

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



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

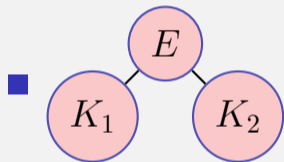
Evaluation Order – more formally



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

- "Good expression": any valid evaluation order leads to the same result.

Evaluation Order – more formally



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

- Example for a “bad expression”: $a*(a=2)$

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
Modulo	%	2	14	links
Addition	+	2	13	left
Subtraction	-	2	13	left

Arithmetic operations

	Symbol	Arity	Precedence	Associativity
Unary +	+	1	16	right
Negation	-	1	16	right
Multiplication	*	2	14	left
Division	-a : R-value → R-value			left
Modulo	%	2	14	links
Addition	+	2	13	left
Subtraction	-	2	13	left

Arithmetic operations

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

Arithmetic operations

	Symbol	Arity	Precedence	Associativity
Unary +	+	1	16	right
Negation	-	1	16	right
Multiplication	*	2	14	left
$a+b : \text{R-value} \times \text{R-value} \rightarrow \text{R-value}$				
Modulo	%	2	14	left
Addition	+	2	13	left
Subtraction	-	2	13	left

Arithmetic operations

	Symbol	Arity	Precedence	Associativity
Unary +	+	1	16	right
Negation	-	1	16	right
-a+b+c				
Addition	+	2	13	left
Subtraction	-	2	13	left

Arithmetic operations

	Symbol	Arity	Precedence	Associativity
Unary +	+	1	16	right
Negation	-	1	16	right
-a+b+c				
Addition	+	2	13	left
Subtraction	-	2	13	left

Arithmetic operations

	Symbol	Arity	Precedence	Associativity
Unary +	+	1	16	right
Negation	-	1	16	right
$-a+b+c = ((-a) + b) + c$ $R\text{-value} \times R\text{-value} \times R\text{-value} \rightarrow R\text{-value}$				
Addition	+	2	13	left
Subtraction	-	2	13	left

Interlude: Assignment expression – in more detail

- Already known: $a = b$ means
Assignment of b (R-value) to a (L-value).
Returns: L-value

Interlude: 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?

Interlude: 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

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

Interlude: Assignment expression – in more detail

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

Example multiple assignment:

$a = b = 0 \implies b=0; a=0$

Division

- Operator `/` implements integer division

`5 / 2` has value 2

Division

- Operator `/` implements integer division

```
5 / 2 has value 2
```

- In `fahrenheit.cpp`

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

```
15 degrees Celsius are 59 degrees Fahrenheit
```

Division

- In `fahrenheit.cpp`

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

15 degrees Celsius are 59 degrees Fahrenheit

Division

- In `fahrenheit.cpp`

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

15 degrees Celsius are 59 degrees Fahrenheit

- Mathematically equivalent...

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

Division

- In `fahrenheit.cpp`

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

15 degrees Celsius are 59 degrees Fahrenheit

- Mathematically equivalent...

```
1 * celsius + 32
```

Division

- In `fahrenheit.cpp`

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

```
15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematically equivalent...

```
15 + 32
```

Division

- In `fahrenheit.cpp`

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

```
15 degrees Celsius are 59 degrees Fahrenheit
```

- Mathematically equivalent...

```
47
```

Division

- 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

Loss of Precision

Guideline

- Watch out for potential loss of precision
- Postpone operations with potential loss of precision to avoid “error escalation”

Division and Modulo

- Modulo-operator computes the rest of the integer division

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

Division and Modulo

- Modulo-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.
```

Increment and decrement

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

Disadvantages

- relatively long

Increment and decrement

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

Disadvantages

- relatively long
- `expr` is evaluated twice
 - Later: L-valued expressions whose evaluation is “expensive”

Increment and decrement

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

Disadvantages

- relatively long
- `expr` is evaluated twice
 - Later: L-valued expressions whose evaluation is “expensive”
 - `expr` could have an effect (but should not, cf. guideline)

In-/Decrement Operators

Post-Increment

```
expr++
```

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

In-/Decrement Operators

Pre-increment

`++expr`

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

In-/Decrement Operators

Post-Decrement

```
expr--
```

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

In-/Decrement Operators

Prä-Dekrement

`--expr`

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

In-/Decrement Operators

Example

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

In-/Decrement Operators

Example

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

In-/Decrement Operators

Example

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

In-/Decrement Operators

Example

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

C++ **vs.** ++C

Strictly speaking our language should be named ++C because

- it is an advancement of the language C

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`

\Leftrightarrow

`a = a + b`

Arithmetic Assignments

`a += b`

\Leftrightarrow

`a = a + b`

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

Binary Number Representations

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

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

corresponds to the number $b_n \cdot 2^n + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0$

Binary Number Representations

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

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

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

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

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 $32+8+2+1$.

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.

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.



Binary Numbers: Numbers of the Computer?

Truth: Computers calculate using binary numbers.

NEUE ZÜRCHER ZEITUNG

TECHNIK

Mittwoch, 30. August 1950 Blatt 13
Mittagsausgabe Nr. 1796 (50)

Das programmgesteuerte Rechengertät an der Eidgenössischen Technischen Hochschule in Zürich

Die Entwicklung programmgesteuerter Rechenmaschinen in den Vereinigten Staaten von Amerika ist in den Artikeln „Elektronische Rechenmaschinen“ (vgl. Nr. 2149 der „N. Z. Z.“ vom 13. Oktober 1948) und „Die neueste elektronische Rechenmaschine“ (vgl. Nr. 872 der „N. Z. Z.“ vom 26. April 1950) behandelt. Nachstehend soll von einem Gerät deutscher Herkunft — Zuse K-6, Neukirchen — die Rolle 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. Dem ist dieses Institut in der Lage, dem in der Schweiz immer stärker werdenden Bedarf nach einer leistungsfähigen Zentraltabelle für numerische Rechnungen wenigstens teilweise gerecht zu werden. Bereits sind einige mathematische Probleme behandelt worden, und die Erfüllung vieler anderer Aufgaben ist vorbereitet.

Merkmale des Gerätes

Das Gerät ist ein Glied in dem progressiven Entwicklungsprogramm des Ingenieurs Konrad Zuse; es wurde im Auftrag des Instituts für angewandte Mathematik der E. T. H. unter Berücksichtigung von dessen Wünschen und Ideen von Zuse als „Modell Z 4“ konstruiert. Die ursprüngliche Entwicklung in Deutschland erfolgte in den Kriegsjahren und verlief völlig unabhängig von den Untersuchungen des Vereinigten Staates. Es ist äusserst interessant festzustellen, wie für die meisten wesentlichen funktionalen Probleme bedenkterweise genau dieselbe Lösung gefunden wurde, wie aber andererseits gewissen Fragen sekundärer Wichtigkeit eine ganz unterschiedliche Bedeutung beigegeben wurde.

Eine kurze technische Charakterisierung lautet wie folgt: Elektronenmechanisches Gerät mit 2200 Relais, 21 Schrittschaltern und einem Speicher für 64 Zahlen, welcher mit neuartigen, mechanischen Schaltgliedern arbeitet; Vervollständigung des Dualsystems und der halblogarithmischen Darstellung; Multiplikationszeit 2,5 Sekunden; Programmsteuerung mit Hilfe zweier Lochstreifen, auf die willkürlich umgeschaltet werden kann; Eingabe von Zahlen durch eine Tastatur oder durch einen Lochstreifen; Abgabe der Resultate durch Lampenfeld, Lochstreifen oder Druckwerk.

Das duale Zahlssystem

Allgemein wird programmgesteuertes Rechengertät häufig als duale Zahlssystem zugrunde gelegt, welches nur die zwei Zahlensymbole 0 und 1 verwendet, während das bekannte Dezimalsystem

lesen wir eine Dezimalzahl von rechts nach links, so erhält sich ein Glied in dem progressiven Entwicklungsprogramm des Ingenieurs Konrad Zuse; es wurde im Auftrag des Instituts für angewandte Mathematik der E. T. H. unter Berücksichtigung von dessen Wünschen und Ideen von Zuse als „Modell Z 4“ konstruiert. Die ursprüngliche Entwicklung in Deutschland erfolgte in den Kriegsjahren und verlief völlig unabhängig von den Untersuchungen des Vereinigten Staates. Es ist äusserst interessant festzustellen, wie für die meisten wesentlichen funktionalen Probleme bedenkterweise genau dieselbe Lösung gefunden wurde, wie aber andererseits gewissen Fragen sekundärer Wichtigkeit eine ganz unterschiedliche Bedeutung beigegeben wurde.

$$a \cdot 10^3 + b \cdot 10^2 + c \cdot 10^1 + d \cdot 10^0 + e \cdot 10^{-1} + f \cdot 10^{-2} + g \cdot 10^{-3}$$

Die Zahl 1 wird in beiden Systemen gleich dargestellt. Um jedoch duale von dezimalen Zahlen deutlich zu trennen, schreiben wir die duale 1 als 1_2 . Dagegen weicht schon die 2 ab, indem sie im 10_2 lautet, denn dies bedeutet $1 \cdot 10^1 + 0 \cdot 10^0 = 2$. Wenn einer Zahl (ohne Stellen nach dem Komma) bereits eine Null zugefügt wird, so vergrössert sie sich um den Faktor 2 (und sieht, wie im Dualsystem, um den Faktor 10). Auf diese Weise kann aus $10_2 = 2$ auf einfachste Weise gebildet werden: $100_2 = 4$, $1000_2 = 8$, $10000_2 = 16$, usw.

Die Dualzahl 10101_2 bedeutet aus also: $1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 21$

Ganz analog sind etwaige Stellen nach dem Komma zu interpretieren; so wird $1,011_2$ wie folgt interpretiert:

$$1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} = 1 + \frac{1}{4} + \frac{1}{8} = 1,375$$

Der große Vorteil, der das Dualsystem für Rechenautomaten so geeignet macht, nämlich die Reduktion der Anzahl der verwendeten Symbole auf nur zwei, wird allerdings durch einen Nachteil erkauft: Es braucht mehr Stellen, um eine bestimmte Zahl anzustellen. Die einstellige Zahl 8

Änderung des Maßstabes durchgeführt 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 ausgeordnete duale Zahlensystem verirrt wird, ist das Beispiel im Dezimalsystem durchgeführt; doch wird daran erinnert, daß das Gerät in Wirklichkeit mit dualen Zahlen rechnet.

Es soll also etwa addiert werden: $2,345678 \times 10^3 + 9,876543 \times 10^1$ (Man beachte, daß die jeweilige Zahl stets zwischen 1 und 10 liegt, also das Komma nachher ersten Stelle ist). Nun müssen die beiden Summanden „ausgerichtet“ werden, d. h. die beiden Exponenten sind einander gleich zu machen und man erhält der kleinere Exponent den Wert des größeren, also 2. Die Zahlen unten nun, richtig untereinander geschrieben und addiert, wie folgt:

$$\begin{array}{r} 2,345678 \times 10^3 \\ 0,098765 \times 10^3 \\ \hline 2,555543 \times 10^3 \end{array}$$

Es ist ersichtlich, daß bei der kleineren der beiden Zahlen rechtz. einige Stellen abgeschrieben werden mußten; denn wenn die Summanden siebenstellig gegeben waren, so soll auch das Resultat nicht mehr als sieben Stellen enthalten.



Abb. 2. Der Schalter bei der Fertigung eines Lochstreifen. Die Abfächer für den Lochstreifen sind deutlich sichtbar.

Befehle können „belting“ gegeben werden, d. h. ihre Ausführung wird von der Natur eines errechneten Resultates abhängig gemacht. Erst dadurch werden die außerordentlich vielen Permut-

Binary Numbers: Numbers of the Computer?

Stereotype: computers are talking 0/1 gibberish

Biooioo Biooioo Biooioo

01001110 01011010 01011010

Freitag, 8. Juni 2012 · Nr. 131 · 233. Jhg.

01001010 01010110 01001101

www.nzz.ch · Fr. 4.00 · € 3.50



01000010 01100101
01110010 01101001

01100011 01101000 01110100 01100101

00100000 11111100 01100010
01100101 01110010 00100000
01101110 01100101 01110101 011-
00101 01110011 00100000 010-
01101 01100001 01110011

01121001 01100001

01100011 01100001 01100000 0110-
0001 01101110 00100000 01010011 01111-
001 01110010 01100001 01101110
0000101 00001010 00001101 00000010
0000101 0101101 0110111 0101101 010-
00010 0100001 0001110 01100010 01100001
01100011 01101000 01110010 01100101 011-
10010 00100000 01110110 0001110 011001
00100000 01010011 01100001 01100000 011-
00001 01110101 01100000 01101100 0100-
001 011010100 01110000 01010000

01100110 01100101

01100010 01101110 01100111 01100001 011-
0000 01100001 01101100 01100100 010-
0001 01101110 00000101 00000000 00000101
00000000 00001100 01100001 01110101 011-
10000 00000000 00000000 01100101 0111-
0010 01101001 01100001 01100000 01100000

01100101 01110001 00100000 01001101 011-
00001 01110001 01110011 01100000 01101-
011 01100101 01110010 01010000 01110011
01110000 01100001 01110010 01110000 011-
00010 00000001 01101010 01101010 0110110
01100100 01100101 01101110 00101110 001-
00000 011000100 01101001 01100101 001-
00000 00101000 01100101 01100110 01100001
01100101 01110010 01110011 01101101
01100111 00100000 01101010 01100001 011-
00010 01101000 01110010 01100101 00000-
000 01110001 01101110 01100111 01100000
01101110 01100001 01101110 01101110 0111-
0000 01100011 00000000 00001011 01010100
01100101 01110010 01110010 01101111 011-
10000 01101010 01110011 01110010 01100001
01101110 10110111 00100000 01100100 011-
00001 01100110 11111100 01110010

00100000 01110110

01100101 01110010 01100001 01101110 0111-
0000 01110111 01110010 01110010 01101010
01101100 01100001 01100000 001-
0110 00000101 00000101 0000101 000-
00010 01000010 11111100 01110010 011001-
11 00100000 01000000 01100101 01100101
01100001 01101000 01101111 01100110 0111-
0010 00101100 00100000 00000000 01100000

01000110 01101100 11111100

01100001 01101000 01101100 01101100 01100001 01101110 01100011 01110001 01100101 01101100 01100101 01101110 01100000 0000-
0000 01101000 01101110 00100000 01010000 01100001 01110100 01110010 01100001 01110001 0000101 00000100 00000000 01101101
01100101 00100000 01100111 01100101 01101101 01100101 01100101 01100101 01100101 01100101

01101100

Computing Tricks

- Estimate the orders of magnitude of powers of two.²:

$$2^{10} = 1024 = 1\text{Ki} \approx 10^3.$$

$$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi}.$$

$$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}.$$

²Decimal 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)

Computing Tricks

- Estimate the orders of magnitude of powers of two.²:

$$2^{10} = 1024 = 1\text{Ki} \approx 10^3.$$

$$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi}.$$

$$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}.$$

²Decimal 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)

Computing Tricks

- Estimate the orders of magnitude of powers of two.²:

$$2^{10} = 1024 = 1\text{Ki} \approx 10^3.$$

$$2^{32} = 4 \cdot (1024)^3 = 4\text{Gi}.$$

$$2^{64} = 16\text{Ei} \approx 16 \cdot 10^{18}.$$

²Decimal 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.

Why Hexadecimal Numbers?

- A Hex-Nibble requires exactly 4 bits.
- “compact representation of binary numbers”

Why Hexadecimal Numbers?

- A Hex-Nibble requires exactly 4 bits.
- “compact representation of binary numbers”

32-bit numbers: `0x00000000` -- `0xffffffff` .

`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”

Why Hexadecimal Numbers?

- A Hex-Nibble requires exactly 4 bits.
- “compact representation of binary numbers”

32-bit numbers: 0x00000000 -- 0xffffffff .

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”

Why Hexadecimal Numbers?

- A Hex-Nibble requires exactly 4 bits.
- “compact representation of binary numbers”

32-bit numbers: 0x00000000 -- 0xffffffff .

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”

Why Hexadecimal Numbers?

- A Hex-Nibble requires exactly 4 bits.
- “compact representation of binary numbers”

32-bit numbers: 0x00000000 -- 0xffffffff .

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”

Why Hexadecimal Numbers?

“For programmers and technicians”

(user manual chess computer *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:

$$ABCD = (A \times 16^3) + (B \times 16^2) + (C \times 16^1) + (D \times 16^0)$$

Für A gilt: 7 = -1; 8 = 0; 9 = +1 usw.

Eine Bauereinheit (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:
 $(14 \times 16^0) + (5 \times 16^1) + (0 \times 16^2) + (0 \times 16^3) = 14 + 80 + 0 + 0 =$
 $= +94 \text{ Punkte.}$

7F80

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

Example: Hex-Colors

#00FF00



r

g

b

Example: Hex-Colors

#FFFFFF00

r g b

Example: Hex-Colors

#808080



r g b

Example: Hex-Colors

#FF0050



r g b

Domain of Type int

```
// 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;
}
```

Domain of Type int

```
// 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;
}
```

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

Domain of Type int

```
// 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;
}
```

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

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

Domain of the Type `int`

- Representation with B bits. Domain

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

Where does this partitioning come from?

- On most platforms $B = 32$

Domain of the Type `int`

- Representation with B bits. Domain

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

Where does this partitioning come from?

- For the type `int` C++ guarantees $B \geq 16$

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

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

- 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

$x + 2^B$

Computing with Binary Numbers (4 digits)

Simple Addition

$$\begin{array}{r} 2 \\ +3 \\ \hline 5 \end{array}$$

$$\begin{array}{r} 0010 \\ +0011 \\ \hline 0101 \end{array}$$

Computing with Binary Numbers (4 digits)

Simple Subtraction

5

−3

2

0101

−0011

0010

Computing with Binary Numbers (4 digits)

Addition with Overflow

$$\begin{array}{r} 7 \\ +9 \\ \hline 16 \end{array}$$

$$\begin{array}{r} 0111 \\ +1001 \\ \hline (1)0000 \end{array}$$

Computing with Binary Numbers (4 digits)

Negative Numbers?

$$\begin{array}{r} 5 \\ +(-5) \\ \hline 0 \end{array}$$

$$\begin{array}{r} 0101 \\ \quad ??? \\ \hline (1)0000 \end{array}$$

Computing with Binary Numbers (4 digits)

Simpler -1

$$\begin{array}{r} 1 \\ +(-1) \\ \hline 0 \end{array}$$

$$\begin{array}{r} 0001 \\ 1111 \\ \hline (1)0000 \end{array}$$

Computing with Binary Numbers (4 digits)

Utilize this:

3

+?

-1

0011

+????

1111

Computing with Binary Numbers (4 digits)

Invert!

$$\begin{array}{r} 3 \\ +(-4) \\ \hline -1 \end{array}$$

$$\begin{array}{r} 0011 \\ +1100 \\ \hline 1111 \hat{=} 2^B - 1 \end{array}$$

Computing with Binary Numbers (4 digits)

$$\begin{array}{r} a \\ +(-a - 1) \\ \hline -1 \end{array}$$

$$\begin{array}{r} a \\ \bar{a} \\ \hline 1111 \hat{=} 2^B - 1 \end{array}$$

Computing with Binary Numbers (4 digits)

- Negation: inversion and addition of 1

$$-a \hat{=} \bar{a} + 1$$

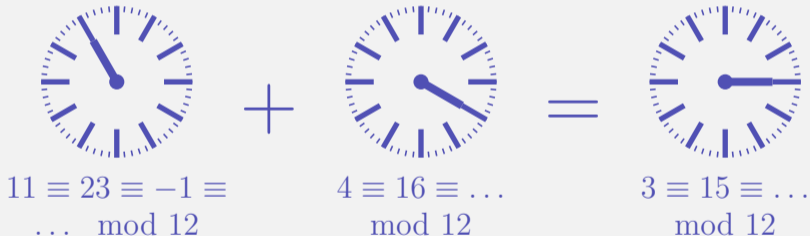
Computing with Binary Numbers (4 digits)

- Wrap around semantics (calculating modulo 2^B)

$$-a \hat{=} 2^B - a$$

Why this works

Modulo arithmetics: Compute on a circle³



³The arithmetics also work with decimal numbers (and for multiplication).

Negative Numbers (3 Digits)

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

Negative Numbers (3 Digits)

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

Negative Numbers (3 Digits)

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

Negative Numbers (3 Digits)

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

Negative Numbers (3 Digits)

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

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		

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		

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 *and* it contributes to the value.

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";
```

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*

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*

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*

The Type `bool` in C++

- represents *logical values*
- Literals `false` and `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)

arithmetic type \times arithmetic type \rightarrow bool

R-value \times R-value \rightarrow R-value

Relational Operators

`a < b` (smaller than)

```
bool b = (1 < 3); // b =
```

Relational Operators

`a < b` (smaller than)

```
bool b = (1 < 3); // b = true
```

Relational Operators

`a >= b` (greater than)

```
int a = 0;  
bool b = (a >= 3); // b =
```

Relational Operators

`a >= b` (greater than)

```
int a = 0;  
bool b = (a >= 3); // b = false
```

Relational Operators

a == b (equals)

```
int a = 4;  
bool b = (a % 3 == 1); // b =
```


Relational Operators

a == b (equals)

```
int a = 4;  
bool b = (a % 3 == 1); // b = true
```

Relational Operators

`a != b` (not equal)

```
int a = 1;  
bool b = (a != 2*a-1); // b =
```

Relational Operators

`a != b` (not equal)

```
int a = 1;  
bool b = (a != 2*a-1); // b = false
```

Boolean Functions in Mathematics

- Boolean function

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

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

- “logical And”

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

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

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

Logical Operator &&

a && b (logical and)

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

Logical Operator &&

a && b (logical and)

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

- “logical Or”

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

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

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

Logical Operator ||

a || b (logical or)

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

Logical Operator ||

a || b (logical or)

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

- “logical Not”

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

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

x	NOT(x)
0	1
1	0

Logical Operator !

!b (logical not)

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

Logical Operator !

!b (logical not)

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

Precedences

`!b && a`

Precedences

`!b && a`
⇕
`(!b) && a`

Precedences

a && b || c && d

Precedences

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

Precedences

a || b && c || d

Precedences

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

Precedences

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

Precedences

The unary logical operator !

binds more strongly than

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

Precedences

The unary logical operator !

binds more strongly than

binary arithmetic operators. These

bind more strongly than

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

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

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

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)
```


Completeness

- AND, OR and NOT are the boolean functions available in C++.

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

$$x \oplus y$$

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

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

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

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

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

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

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

$$(x \ || \ y) \ \&\& \ !(x \ \&\& \ y)$$

Completeness Proof

- Identify binary boolean functions with their characteristic vector.

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

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

Completeness Proof

- Identify binary boolean functions with their characteristic vector.

x	y	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}

$$f_{0001} = \text{AND}(x, y)$$

$$f_{0010} = \text{AND}(x, \text{NOT}(y))$$

$$f_{0100} = \text{AND}(y, \text{NOT}(x))$$

$$f_{1000} = \text{NOT}(\text{OR}(x, y))$$

Completeness Proof

- Step 2: generate all functions by applying logical or

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

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

bool vs int: Conversion

- `bool` can be used whenever `int` is expected

<code>bool</code>	→	<code>int</code>
<i>true</i>	→	1
<i>false</i>	→	0

bool vs int: Conversion

- `bool` can be used whenever `int` is expected – and vice versa.

<code>bool</code>	→	<code>int</code>
<i>true</i>	→	1
<i>false</i>	→	0
<code>int</code>	→	<code>bool</code>
<code>≠0</code>	→	<i>true</i>
0	→	<i>false</i>

bool vs int: Conversion

- `bool` can be used whenever `int` is expected – and vice versa.

<code>bool</code>	→	<code>int</code>
<i>true</i>	→	1
<i>false</i>	→	0
<code>int</code>	→	<code>bool</code>
<code>≠0</code>	→	<i>true</i>
0	→	<i>false</i>

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


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.

<code>bool</code>	→	<code>int</code>
<i>true</i>	→	1
<i>false</i>	→	0
<code>int</code>	→	<code>bool</code>
<code>≠0</code>	→	<i>true</i>
0	→	<i>false</i>

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

DeMorgan Rules

- $!(a \ \&\& \ b) == (!a \ || \ !b)$

DeMorgan Rules

■ $!(a \ \&\& \ b) == (!a \ || \ !b)$

$!(\text{rich } \textit{and} \ \text{beautiful}) == (\text{poor } \textit{or} \ \text{ugly})$

DeMorgan Rules

■ $!(a \ \&\& \ b) == (!a \ || \ !b)$

■ $!(a \ || \ b) == (!a \ \&\& \ !b)$

$!(\text{rich } \textit{and} \ \text{beautiful}) == (\text{poor } \textit{or} \ \text{ugly})$

Application: either ... or (XOR)

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

Application: either ... or (XOR)

`(x || y) && !(x && y)` x or y, and not both

Application: either ... or (XOR)

`(x || y) && !(x && y)` `x or y, and not both`

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

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

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)`

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

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)`

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.

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 has value 6 \Rightarrow

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

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 has value 6 \Rightarrow

```
true && z / x > y
```

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 has value 6 \Rightarrow

```
true && z / x > y
```


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 has value 0 \Rightarrow

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

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 has value 0 \Rightarrow

```
false && z / x > y
```

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 has value 0 \Rightarrow

false

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 has value 0 \Rightarrow

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

\Rightarrow No division by 0

4. Defensive Programming

Constants and Assertions

Sources of Errors

- Errors that the compiler can find:
syntactical and some semantical errors

Sources of Errors

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

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

Constants

- are variables with immutable value

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

- Usage: `const` before the definition

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*”

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*”

The Compiler as Your Friend: Constants

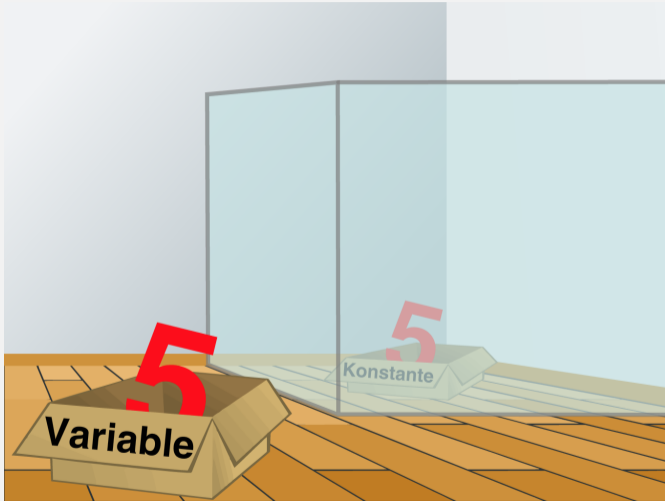
- Compiler checks that the `const`-promise is kept

```
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.

Avoid Sources of Bugs

1. Exact knowledge of the wanted program behavior

Avoid Sources of Bugs

1. Exact knowledge of the wanted program behavior

» It's not a bug, it's a feature! «

Avoid Sources of Bugs

1. Exact knowledge of the wanted program behavior
2. Check at many places in the code if the program is still on track

Avoid Sources of Bugs

1. Exact knowledge of the wanted program behavior
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

Against Runtime Errors: *Assertions*

`assert(expr)`

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

Against Runtime Errors: *Assertions*

`assert(expr)`

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

Assertions for the $\text{gcd}(x, y)$

Check if the program is on track ...

```
// Input x and y
std::cout << "x =? ";
std::cin >> x;
std::cout << "y =? ";
std::cin >> y;
```

Input arguments for calculation

```
// Check validity of inputs
assert(x > 0 && y > 0);
```

```
... // Compute gcd(x,y), store result in variable a
```

Assertions for the $\text{gcd}(x, y)$

Check if the program is on track ...

```
// Input x and y
```

```
std::cout << "x =? ";
```

```
std::cin >> x;
```

```
std::cout << "y =? ";
```

```
std::cin >> y;
```

```
// Check validity of inputs
```

```
assert(x > 0 && y > 0); ← Precondition for the ongoing computation
```

```
... // Compute gcd(x,y), store result in variable a
```


Assertions for the $\text{gcd}(x, y)$

... and question the obvious! ...

...

```
assert(x > 0 && y > 0);
```

← Precondition for the ongoing computation

```
... // Compute gcd(x,y), store result in variable a
```

```
assert (a >= 1);
```

```
assert (x % a == 0 && y % a == 0);
```

```
for (int i = a+1; i <= x && i <= y; ++i)
```

```
    assert(!(x % i == 0 && y % i == 0));
```

Assertions for the $\text{gcd}(x, y)$

... and question the obvious! ...

...

```
assert(x > 0 && y > 0);
```

```
... // Compute gcd(x,y), store result in variable a
```

```
assert (a >= 1);
```

```
assert (x % a == 0 && y % a == 0);
```

```
for (int i = a+1; i <= x && i <= y; ++i)
```

```
    assert(!(x % i == 0 && y % i == 0));
```

Properties of the
gcd

Switch off Assertions

```
#define NDEBUG // To ignore assertions  
#include<cassert>
```

...

```
assert(x > 0 && y > 0); // Ignored
```

```
... // Compute gcd(x,y), store result in variable a
```

```
assert(a >= 1); // Ignored
```

...

Fail-Fast with Assertions

- Real software: many C++ files, complex control flow



Fail-Fast with Assertions

- Real software: many C++ files, complex control flow



Fail-Fast with Assertions

- Real software: many C++ files, complex control flow
- Errors surface late(r) → impedes error localisation



Fail-Fast with Assertions

- Real software: many C++ files, complex control flow
- Errors surface late(r) → impedes error localisation
- Assertions: Detect errors early

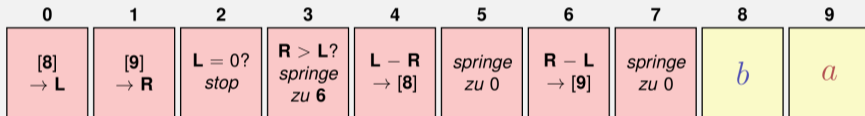


5. Control Structures I

Selection Statements, Iteration Statements, Termination, Blocks

Control Flow

- Up to now: *linear* (from top to bottom)
- Interesting programs require “branches” and “jumps”



Selection Statements

implement branches

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

if-Statement

```
if ( condition )  
    statement
```

if-Statement

```
if ( condition )  
    statement
```

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

if-Statement

```
if ( condition )  
    statement
```

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

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

if-Statement

```
if ( condition )  
    statement
```

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

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

if-else-statement

```
if ( condition )  
    statement1  
else  
    statement2
```

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-else-statement

```
if ( condition )  
    statement1  
else  
    statement2
```

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

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

if-else-statement

```
if ( condition )  
    statement1  
else  
    statement2
```

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

- *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";  
else  
    std::cout << "odd";
```

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 $1 + 2 + \dots + n$

```
// 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";
```

Compute $1 + 2 + \dots + n$

```
// 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";
```

for-Statement Example

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

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

i

s

for-Statement Example

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

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

i	s
<hr/>	
$i==1$	

for-Statement Example

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

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

i	s
$i == 1$	$i <= 2?$

for-Statement Example

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

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

i		s
i==1	wahr	

for-Statement Example

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

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

i		s
$i == 1$	wahr	$s == 1$

for-Statement Example

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

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

i		s
$i==1$	wahr	$s == 1$
$i==2$		

for-Statement Example

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

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

i		s
$i==1$	wahr	$s == 1$
$i==2$	$i <= 2?$	

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	

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

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		

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	i <= 2?	

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	

for-Statement Example

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

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

<i>i</i>		<i>s</i>
<i>i</i> ==1	wahr	<i>s</i> == 1
<i>i</i> ==2	wahr	<i>s</i> == 3
<i>i</i> ==3	falsch	
		<i>s</i> == 3

for-Statement: Syntax

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

for-Statement: Syntax

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

- *init statement*: expression statement, declaration statement, null statement

for-Statement: Syntax

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

- *init statement*: expression statement, declaration statement, null statement
- *condition*: convertible to `bool`

for-Statement: Syntax

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

- *init statement*: expression statement, declaration statement, null statement
- *condition*: convertible to `bool`
- *expression*: any expression

for-Statement: Syntax

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

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

Gauß as a Child (1777 - 1855)

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

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 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

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

The Solution of Gauß

- The requested number is

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

- This is half of

$$\begin{array}{r} 1 + 2 + \dots + 99 + 100 \\ + 100 + 99 + \dots + 2 + 1 \\ \hline = 101 + 101 + \dots + 101 + 101 \end{array}$$

The Solution of Gauß

- The requested number is

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

- This is half of

$$\begin{array}{r} 1 + 2 + \dots + 99 + 100 \\ + 100 + 99 + \dots + 2 + 1 \\ \hline = 101 + 101 + \dots + 101 + 101 \end{array}$$

- 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* .

for-Statement: Termination

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

Here and in most cases:

- 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.

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 (init; cond; expr) stmt;
```

Halting Problem

Undecidability of the Halting Problem

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

⁴Alan Turing, 1936. Theoretical questions of this kind were the main motivation for Alan Turing to construct a computing machine.

Halting Problem

Undecidability of the Halting Problem

There is no C++ program that can determine for each 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.⁴

⁴Alan Turing, 1936. Theoretical questions 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 $d \in \{2, \dots, n - 1\}$ divides n .

Example: Prime Number Test

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

A loop that can test this:

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

Example: Prime Number Test

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

A loop that can test this:

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

(body is the null statement)

Example: Termination

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

- Progress: Initial value $d=2$, then plus 1 in every iteration ($++d$)

Example: Termination

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

- Progress: Initial value $d=2$, then plus 1 in every iteration ($++d$)
- Exit: $n\%d \neq 0$ evaluates to `false` as soon as a divisor is found — at the latest, once $d == n$

Example: Termination

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

- Progress: Initial value $d=2$, then plus 1 in every iteration ($++d$)
- Exit: $n\%d \neq 0$ evaluates to `false` as soon as a divisor is found — at the latest, once $d == n$
- Progress guarantees that the exit condition will be reached

Example: Correctness

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

Every potential divisor $2 \leq d \leq n$ will be tested. If the loop terminates with $d == n$ then and only then is n prime.

Blocks

- Blocks group a number of statements to a new statement

```
{statement1 statement2 ... statementN}
```

Blocks

- Blocks group a number of statements to a new statement
- Example: body of the main function

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

Blocks

- Blocks group a number of statements to a new statement
- Example: loop body

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

Blocks

- Blocks group a number of statements to a new statement
- Beispiel: if / else

```
if (d < n) // d is a divisor of n in {2,...,n-1}
    std::cout << n << " = " << d << " * " << n / d << ".\n";
else {
    assert (d == n);
    std::cout << n << " is prime.\n";
}
```


6. 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“

Potential Scope

in the block

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

in function body

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

in control statement

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

Potential Scope

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

```
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;
}
```

Potential Scope

```
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;
}
```

Real Scope

```
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;
}
```

Local Variables

```
int main()
{
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        std::cout << ++i; // outputs
        int k = 2;
        std::cout << --k; // outputs
    }
}
```


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

```
int main()
{
    int i = 5;
    for (int j = 0; j < 5; ++j) {
        std::cout << ++i; // outputs
        int k = 2;
        std::cout << --k; // outputs
    }
}
```

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

while Statement

```
while ( condition )  
    statement
```

while Statement

```
while ( condition )  
    statement
```

is equivalent to

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

Example: The Collatz-Sequence

$(n \in \mathbb{N})$

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

The Collatz-Sequence

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

n=5: 5

The Collatz-Sequence

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

n=5: 5, 16

The Collatz-Sequence

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

n=5: 5, 16, 8

The Collatz-Sequence

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

n=5: 5, 16, 8, 4

The Collatz-Sequence

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

n=5: 5, 16, 8, 4, 2

The Collatz-Sequence

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

n=5: 5, 16, 8, 4, 2, 1

The Collatz-Sequence

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

n=5: 5, 16, 8, 4, 2, 1, 4

The Collatz-Sequence

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

n=5: 5, 16, 8, 4, 2, 1, 4, 2

The Collatz-Sequence

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

n=5: 5, 16, 8, 4, 2, 1, 4, 2, 1

The Collatz-Sequence

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

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

The Collatz Sequence in C++

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
```


do Statement

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

do Statement

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

is equivalent to

```
statement  
while ( expression )  
    statement
```

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";
}
```

break and continue in practice

- Advantage: Can avoid nested `if-else` blocks (or complex disjunctions)

break and continue in practice

- Advantage: Can avoid nested `if-else` blocks (or complex disjunctions)
- But they result in additional jumps (for- and backwards) and thus potentially complicate the control flow

break and continue in practice

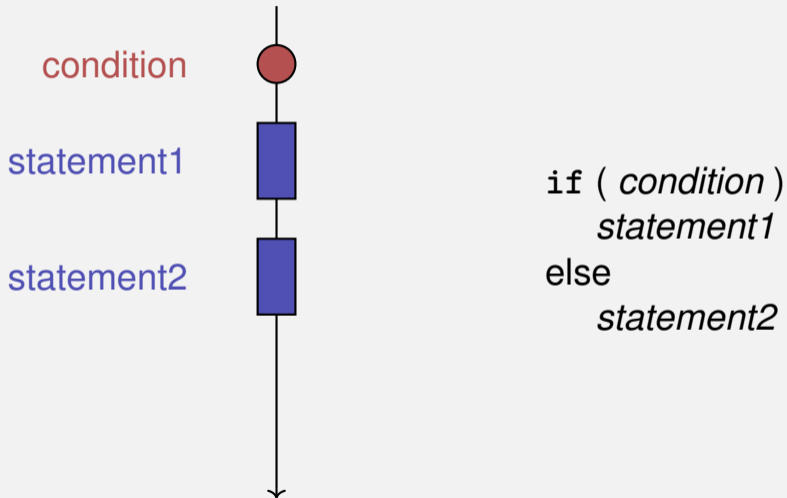
- Advantage: Can avoid nested `if-else` blocks (or complex disjunctions)
- But they result in additional jumps (for- and backwards) and thus potentially complicate the control flow
- Their use is thus controversial, and should be carefully considered

Calculator with continue

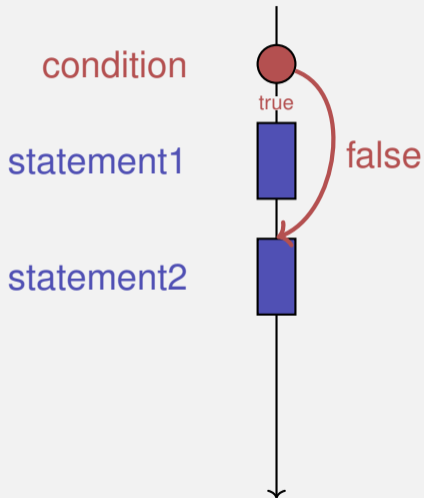
Ignore negative input:

```
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";
}
```


Control Flow if else

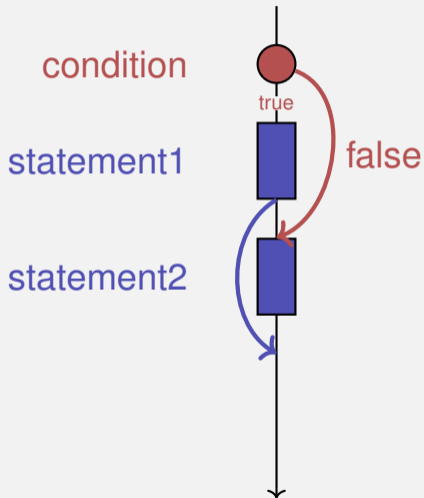


Control Flow if else



```
if ( condition )  
    statement1  
else  
    statement2
```

Control Flow if else



```
if ( condition )  
    statement1  
else  
    statement2
```

Control Flow for

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

init-statement

condition

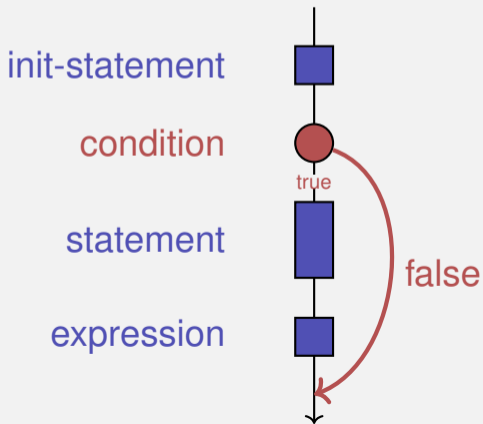
statement

expression



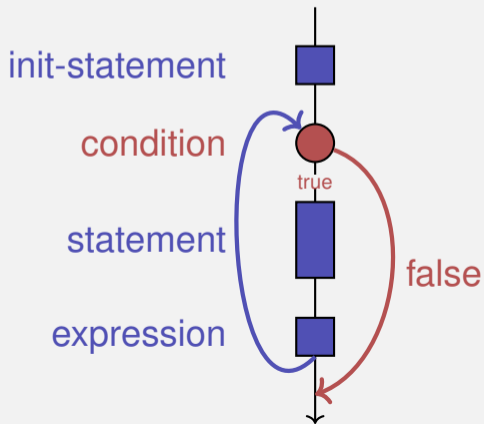
Control Flow for

`for (init statement condition ; expression)
 statement`

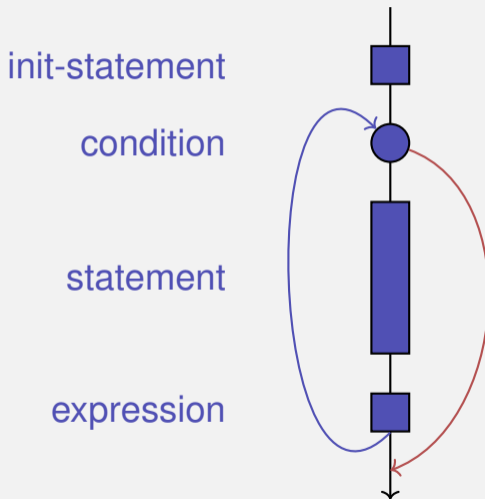


Control Flow for

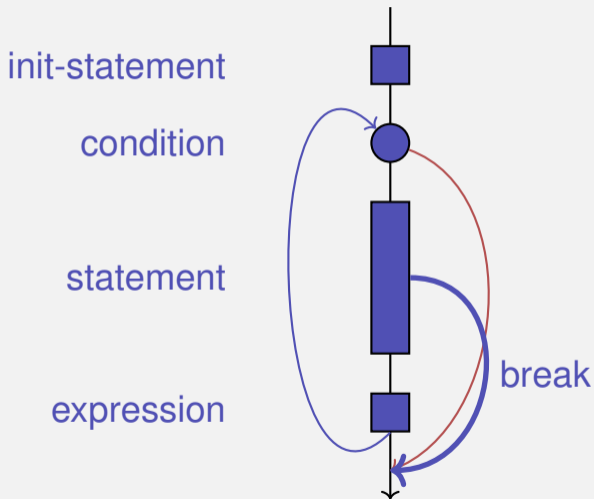
`for (init statement condition ; expression)`
`statement`



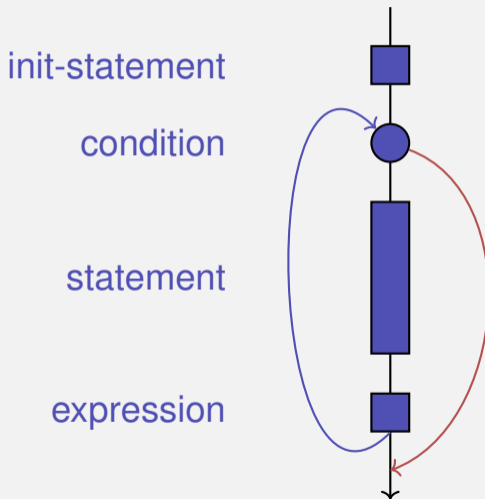
Control Flow break and continue in for



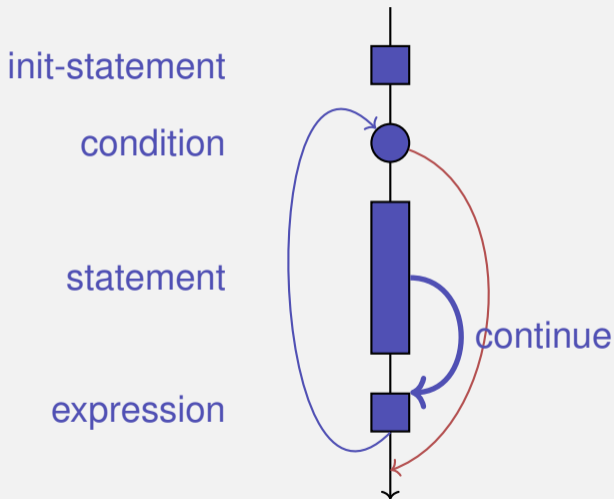
Control Flow `break` and `continue` in `for`



Control Flow break and continue in for



Control Flow break and `continue` in `for`



Control Flow: the Good old Times?

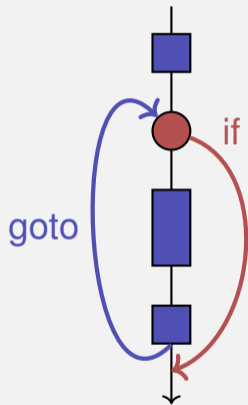
Observation

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

Control Flow: the Good old Times?

Observation

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



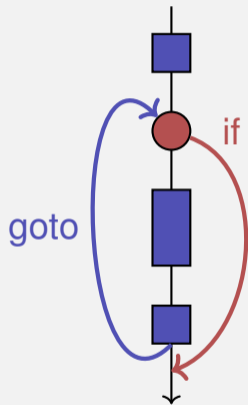
Control Flow: the Good old Times?

Observation

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

Languages based on them:

- Machine Language



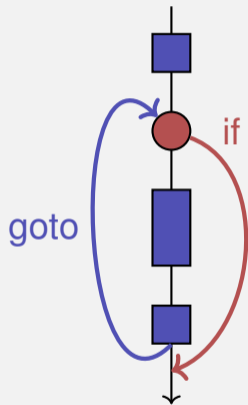
Control Flow: the Good old Times?

Observation

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

Languages based on them:

- Machine Language
- Assembler (“higher” machine language)



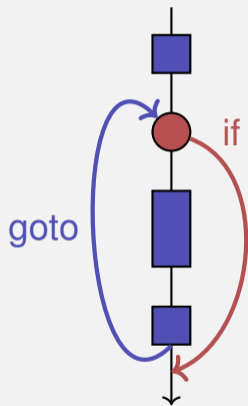
Control Flow: the Good old Times?

Observation

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

Languages based on them:

- Machine Language
- Assembler (“higher” machine language)
- BASIC, the first programming 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 `of ????????????`

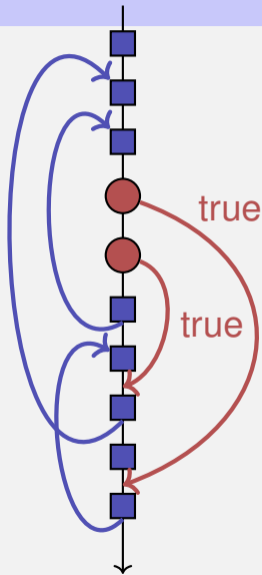
using the programming language BASIC:

```
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
```

Spaghetti-Code with goto

Output of all prime numbers
using the programming language 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

The “right” Iteration Statement

Goals: readability, conciseness, in particular

- few statements

The “right” Iteration Statement

Goals: readability, conciseness, in particular

- few statements
- few lines of code

The “right” Iteration Statement

Goals: readability, conciseness, in particular

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

The “right” Iteration Statement

Goals: readability, conciseness, in particular

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

The “right” Iteration Statement

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

Often not all goals can be achieved simultaneously.

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

First (correct) attempt:

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

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

Less statements, *less* lines:

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

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

Less statements, *simpler* control flow:

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

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

Less statements, *simpler* control flow:

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

This is the “right” iteration statement

Outputting Grades

1. Functional requirement:

6 → "Excellent ... You passed!"

5,4 → "You passed!"

3 → "Close, but ... You failed!"

2,1 → "You failed!"

otherwise → "Error!"

Outputting Grades

1. Functional requirement:

6 → "Excellent ... You passed!"

5,4 → "You passed!"

3 → "Close, but ... You failed!"

2,1 → "You failed!"

otherwise → "Error!"

2. Moreover: Avoid duplication of text and code

Outputting Grades with `if` Statements

```
int grade;
...
if (grade == 6) std::cout << "Excellent ... ";
if (4 <= grade && grade <= 6) {
    std::cout << "You passed!";
} else if (1 <= grade && grade < 4) {
    if (grade == 3) std::cout << "Close, but ... ";
    std::cout << "You failed!";
} else std::cout << "Error!";
```

Outputting Grades with `if` Statements


```
int grade;
...
if (grade == 6) std::cout << "Excellent ... ";
if (4 <= grade && grade <= 6) {
    std::cout << "You passed!";
} else if (1 <= grade && grade < 4) {
    if (grade == 3) std::cout << "Close, but ... ";
    std::cout << "You failed!";
} else std::cout << "Error!";
```

Disadvantage: Control flow – and thus program behaviour – not quite obvious

Outputting Grades with `switch` Statement

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```


Outputting Grades with `switch` Statement

```
switch (grade) {  Jump to matching case
  case 6: std::cout << "Excellent ... ";
  case 5:
  case 4: std::cout << "You passed!";
    break;
  case 3: std::cout << "Close, but ... ";
  case 2:
  case 1: std::cout << "You failed!";
    break;
  default: std::cout << "Error!";
}
```

Outputting Grades with `switch` Statement

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:   
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```

Fall-through



Outputting Grades with `switch` Statement


```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break; ← Exit switch  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```

Fall-through

Outputting Grades with `switch` Statement

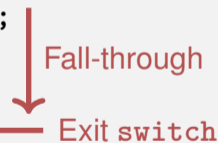
```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```

Fall-through



Outputting Grades with `switch` Statement

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break; ← Exit switch  
    default: std::cout << "Error!";  
}
```



The diagram consists of two red arrows. One arrow points vertically downwards from the right side of the 'case 3' line to the 'break;' line of 'case 1', with the text 'Fall-through' written to its right. A second arrow points horizontally from the right side of the 'break;' line of 'case 1' to the left side of the 'break;' line of 'case 1', with the text 'Exit switch' written below it.

Outputting Grades with `switch` Statement

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!"; ← In all other cases  
}
```

Outputting Grades with `switch` Statement

```
switch (grade) {  
    case 6: std::cout << "Excellent ... ";  
    case 5:  
    case 4: std::cout << "You passed!";  
        break;  
    case 3: std::cout << "Close, but ... ";  
    case 2:  
    case 1: std::cout << "You failed!";  
        break;  
    default: std::cout << "Error!";  
}
```

Advantage: Control flow clearly recognisable

The `switch`-Statement

```
switch (condition)  
    statement
```

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

The `switch`-Statement

```
switch (condition)  
    statement
```

- *condition*: Expression, convertible to integral type
- *statement*: arbitrary statement, in which `case` and `default`-labels are permitted, `break` has a special meaning.
- Use of fall-through property is controversial and should be carefully considered (corresponding compiler warning can be enabled)

7. Floating-point Numbers I

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

“Proper” Calculation

```
// 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";
```

28 degrees Celsius are 82 degrees Fahrenheit.

“Proper” Calculation

```
// 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";
```

28 degrees Celsius are 82 degrees Fahrenheit.

↑

richtig wäre 82.4

“Proper” Calculation

```
// Input
std::cout << "Temperature in degrees Celsius =? ";
float celsius; // Enable fractional numbers
std::cin >> celsius;

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

28 degrees Celsius are 82.4 degrees Fahrenheit.

Fixed-point numbers

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

Fixed-point numbers

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

82.4 = 0000082.400

Fixed-point numbers

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

82.4 = 0000082.400

Disadvantages

- Value range is getting *even* smaller than for integers.

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

- Representability depends on the position of the decimal point.

Floating-point numbers

- Observation: same number, different representations with varying “efficiency”, e.g.

$$\begin{aligned} 0.0824 &= 0.00824 \cdot 10^1 &= 0.824 \cdot 10^{-1} \\ &= 8.24 \cdot 10^{-2} &= 824 \cdot 10^{-4} \end{aligned}$$

Number of *significant digits* remains constant

Floating-point numbers

- Observation: same number, different representations with varying “efficiency”, e.g.

$$\begin{aligned}0.0824 &= 0.00824 \cdot 10^1 &= 0.824 \cdot 10^{-1} \\ &= 8.24 \cdot 10^{-2} &= 824 \cdot 10^{-4}\end{aligned}$$

Number of *significant digits* remains constant

- Floating-point number representation thus:
 - Fixed number of significant places (e.g. 10),
 - Plus position of the decimal point via exponent
 - Number is $Mantissa \times 10^{Exponent}$

Types float and double

- are the fundamental C++ types for floating point numbers
- approximate the field of real numbers $(\mathbb{R}, +, \times)$ from mathematics

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:
 - `float`: approx. 7 digits, exponent up to ± 38
 - `double`: approx. 15 digits, exponent up to ± 308

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:
 - `float`: approx. 7 digits, exponent up to ± 38
 - `double`: approx. 15 digits, exponent up to ± 308
- are fast on most computers (hardware support)

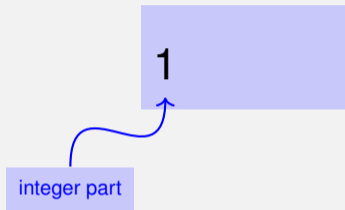
Arithmetic Operators

Analogous to `int`, but ...

- Division operator `/` models a “proper” division (real-valued, not integer)
- No modulo operator, i.e. no `%`

Literals

are different from integers

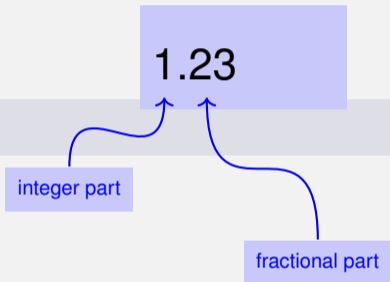


Literals

are different from integers by providing

- decimal point

```
1.0 : type double, value 1
```



Literals

are different from integers by providing

- decimal point

```
1.0 : type double, value 1
```

integer part

1

e-7

exponent

- or exponent.

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

Literals

are different from integers by providing

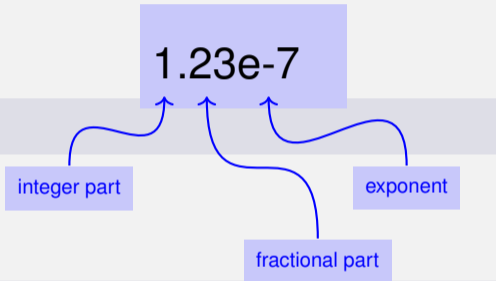
- decimal point

`1.0` : type `double`, value 1

- and / or exponent.

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

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



Literals

are different from integers by providing

- decimal point

`1.0` : type `double`, value 1

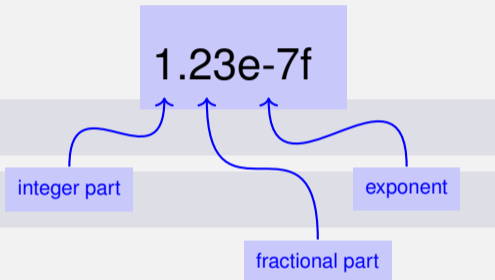
`1.27f` : type `float`, value 1.27

- 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

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

using the first 10 terms.

Computing with float: Euler Number

```
std::cout << "Approximating the Euler number... \n";

// values for i-th iteration, initialized for i = 0
float t = 1.0f; // term 1/i!
float e = 1.0f; // i-th approximation of e

// iteration 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";
}
```

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.

Mixed Expressions, Conversion

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

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
```

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

↑
Typ float, value 28

Mixed Expressions, Conversion

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

`9 * 28.0f / 5 + 32`

Mixed Expressions, Conversion

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

9 * 28.0f / 5 + 32

is converted to float : 9.0f

Mixed Expressions, Conversion

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

`252.0f / 5 + 32`

is converted to float : 5.0f

Mixed Expressions, Conversion

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

50.4f + 32

is converted to float : 32.0f

Mixed Expressions, Conversion

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

82.4f

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";
```

Holes in the value range

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

input 1.5

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

input 1.0

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

input 0.5

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

Holes in the value range

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

input 1.5

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

input 1.0

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

input 0.5

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

output 0

Holes in the value range

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

input 1.1

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

input 1.0

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

input 0.1

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

Holes in the value range

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

input 1.1

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

input 1.0

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

input 0.1

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

output 2.23517e-8

Holes in the value range

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

input 1.1

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

input 1.0

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

input 0.1

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

output 2.23517e-8

What is going on here?

Value range

Integer Types:

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

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”.

8. 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.

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

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

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

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

Representations of the decimal number 0.1 (with $\beta = 10$):

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

Different representations due to choice of exponent

Normalized representation

Normalized number:

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

Normalized representation

Normalized number:

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

Remark 1

The normalized representation is unique and therefore preferred.

Normalized representation

Normalized number:

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

Remark 2

The number 0, as well as all numbers smaller than $\beta^{e_{\min}}$, have no normalized representation (we will come back to this later)

Set of Normalized Numbers

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

Normalized Representation

Example $F^*(2, 3, -2, 2)$

(only positive numbers)

$d_0.d_1d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
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



Normalized Representation

Example $F^*(2, \mathbf{3}, -2, 2)$

(only positive numbers)

$d_0.d_1d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
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



Normalized Representation

Example $F^*(2, 3, -2, 2)$

(only positive numbers)

$d_0.d_1d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
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



Normalized Representation

Example $F^*(2, 3, -2, 2)$

(only positive numbers)

$d_0.d_1d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
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



Normalized Representation

Example $F^*(2, 3, -2, 2)$

(only positive numbers)

$d_0.d_1d_2$	$e = -2$	$e = -1$	$e = 0$	$e = 1$	$e = 2$
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)

Binary and Decimal Systems

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

Conversion

$$(0 < x < 2)$$

Computation of the *binary representation*:

$$x = \sum_{i=0}^{\infty} b_i 2^{-i}$$

Conversion

$$(0 < x < 2)$$

Computation of the *binary representation*:

$$x = b_0.b_1b_2b_3 \dots$$

Conversion

$$(0 < x < 2)$$

Computation of the *binary representation*:

$$\begin{aligned}x &= b_0 \bullet b_1 b_2 b_3 \dots \\ &= b_0 + 0 \bullet b_1 b_2 b_3 \dots\end{aligned}$$

Conversion

$$(0 < x < 2)$$

Computation of the *binary representation*:

$$\begin{aligned}x &= b_0 \bullet b_1 b_2 b_3 \dots \\ &= b_0 + 0 \bullet b_1 b_2 b_3 \dots \\ &\implies\end{aligned}$$

Conversion

$$(0 < x < 2)$$

Computation of the *binary representation*:

$$x = b_0 \bullet b_1 b_2 b_3 \dots$$

$$= b_0 + 0 \bullet b_1 b_2 b_3 \dots$$

$$\implies$$

$$(x - b_0) = 0 \bullet b_1 b_2 b_3 b_4 \dots$$

Conversion

$$(0 < x < 2)$$

Computation of the *binary representation*:

$$\begin{aligned}x &= b_0.b_1b_2b_3\dots \\ &= b_0 + 0.b_1b_2b_3\dots\end{aligned}$$

\implies

$$2 \cdot (x - b_0) = b_1.b_2b_3b_4\dots$$

Conversion

$$(0 < x < 2)$$

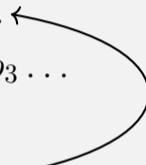
Computation of the *binary representation*:

$$\begin{aligned}x &= b_0 \bullet b_1 b_2 b_3 \dots \leftarrow \\ &= b_0 + 0 \bullet b_1 b_2 b_3 \dots \\ &\implies \\ 2 \cdot (x - b_0) &= b_1 \bullet b_2 b_3 b_4 \dots\end{aligned}$$

Conversion

$$(0 < x < 2)$$

Computation of the *binary representation*:

$$\begin{aligned}x &= b_0.b_1b_2b_3\dots \leftarrow \\ &= b_0 + 0.b_1b_2b_3\dots \\ &\implies \\ 2 \cdot (x - b_0) &= b_1.b_2b_3b_4\dots\end{aligned}$$


```
for (int b_0; x != 0; x = 2 * (x - b_0)) {  
    b_0 = (x >= 1);  
    std::cout << b_0;  
}
```

Example (binary)

$$\begin{aligned}x &= \mathbf{1}.01011 \\ &= \mathbf{1} + 0.01011\end{aligned}$$

\implies

$$2 \cdot (x - \mathbf{1}) = 0.1011$$

Example (binary)

$$\begin{aligned}x &= 1.\mathbf{01011} \\ &= 1 + 0.\mathbf{01011}\end{aligned}$$

\implies

$$2 \cdot (x - 1) = \mathbf{0.1011}$$

Example (binary)

$$x = \mathbf{0}.1011$$

$$= \mathbf{0} + 0.1011$$

\implies

$$2 \cdot (x - \mathbf{0}) = 1.011$$

Example (binary)

$$x = 0.\mathbf{1011}$$

$$= 0 + 0.\mathbf{1011}$$

\implies

$$2 \cdot (x - 0) = \mathbf{1.011}$$

Example (binary)

$$x = \mathbf{1}.011$$

$$= \mathbf{1} + 0.011$$

\implies

$$2 \cdot (x - \mathbf{1}) = 0.11$$

Example (binary)

$$x = 1.\mathbf{011}$$

$$= 1 + 0.\mathbf{011}$$

\implies

$$2 \cdot (x - 1) = \mathbf{0.11}$$

Example (binary)

$$\begin{aligned}x &= \mathbf{0}.11 \\ &= \mathbf{0} + 0.11 \\ &\implies \\ 2 \cdot (x - \mathbf{0}) &= 1.1\end{aligned}$$

Example (binary)

$$\begin{aligned}x &= 0.\mathbf{11} \\ &= 0 + 0.\mathbf{11} \\ &\implies \\ 2 \cdot (x - 0) &= \mathbf{1.1}\end{aligned}$$

Example (binary)

$$\begin{aligned}x &= \mathbf{1}.1 \\ &= \mathbf{1} + 0.1\end{aligned}$$

\implies

$$2 \cdot (x - \mathbf{1}) = 1$$

Example (binary)

$$x = 1.\mathbf{1}$$

$$= 1 + 0.\mathbf{1}$$

\implies

$$2 \cdot (x - 1) = \mathbf{1}$$

Example (binary)

$$x = \mathbf{1}$$

$$= \mathbf{1} + 0$$

$$\implies$$

$$2 \cdot (x - \mathbf{1}) = 0$$

Example (binary)

$$x = 1$$

$$= 1 + 0$$

$$\implies$$

$$2 \cdot (x - 1) = \mathbf{0}$$

Binary representation of 1.1_{10}

$$\begin{array}{r} x \quad b_i \quad x - b_i \quad 2(x - b_i) \\ \hline 1.1 \quad b_0 = \mathbf{1} \end{array}$$

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2

Binary representation of 1.1_{10}

x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_1 = \mathbf{0}$		

Binary representation of 1.1_{10}

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

Binary representation of 1.1_{10}

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	$b_2 = \mathbf{0}$		

Binary representation of 1.1_{10}

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

Binary representation of 1.1_{10}

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	$b_3 = \mathbf{0}$		

Binary representation of 1.1_{10}

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

Binary representation of 1.1_{10}

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	$b_4 = \mathbf{1}$		

Binary representation of 1.1_{10}

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

Binary representation of 1.1_{10}

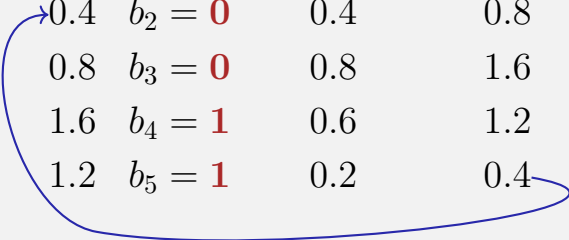
x	b_i	$x - b_i$	$2(x - b_i)$
1.1	$b_0 = \mathbf{1}$	0.1	0.2
0.2	$b_1 = \mathbf{0}$	0.2	0.4
0.4	$b_2 = \mathbf{0}$	0.4	0.8
0.8	$b_3 = \mathbf{0}$	0.8	1.6
1.6	$b_4 = \mathbf{1}$	0.6	1.2
1.2	$b_5 = \mathbf{1}$		

Binary representation of 1.1_{10}

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

Binary representation of 1.1_{10}

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



Binary representation of 1.1_{10}

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

$\Rightarrow 1.000\overline{11}$, periodic, *not* finite

Binary Number Representations of 1.1 and 0.1

- are not finite \Rightarrow conversion errors
- `1.1f` und `0.1f`: *Approximations* of 1.1 and 0.1
- In `diff.cpp`: $1.1 - 1.0 \neq 0.1$

Binary Number Representations of 1.1 and 0.1

- are not finite \Rightarrow conversion errors
- `1.1f` und `0.1f`: *Approximations* of 1.1 and 0.1
- In `diff.cpp`: `1.1 - 1.0 \neq 0.1`

Binary Number Representations of 1.1 and 0.1

- are not finite \Rightarrow conversion errors
- `1.1f` und `0.1f`: *Approximations* of 1.1 and 0.1
- In `diff.cpp`: $1.1 - 1.0 \neq 0.1$

Binary Number Representations of 1.1 and 0.1

on my computer:

$$\begin{aligned} 1.1 &= \underline{1.100000000000000000}888178\dots \\ 1.1f &= \underline{1.1000000}238418\dots \end{aligned}$$

Computing with Floating-point Numbers

is nearly as simple as with integers.

Computing with Floating-point Numbers

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

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 1.011 \cdot 2^{-1} \end{array}$$

1. adjust exponents by denormalizing one number

Computing with Floating-point Numbers

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

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \end{array} \checkmark$$

1. adjust exponents by denormalizing one number

Computing with Floating-point Numbers

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

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \\ \hline \end{array}$$

2. binary addition of the significands

Computing with Floating-point Numbers

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

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \\ \hline = 100.101 \cdot 2^{-2} \checkmark \end{array}$$

2. binary addition of the significands

Computing with Floating-point Numbers

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

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \\ \hline = 100.101 \cdot 2^{-2} \end{array}$$

3. renormalize

Computing with Floating-point Numbers

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

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \\ \hline = 1.00101 \cdot 2^0 \checkmark \end{array}$$

3. renormalize

Computing with Floating-point Numbers

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

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \\ \hline = 1.00101 \cdot 2^0 \end{array}$$

4. round to p significant places, if necessary

Computing with Floating-point Numbers

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

$$\begin{array}{r} 1.111 \cdot 2^{-2} \\ + 10.110 \cdot 2^{-2} \\ \hline = 1.001 \cdot 2^0 \checkmark \end{array}$$

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

The IEEE Standard 754

- defines floating-point number systems and their rounding behavior
- is used nearly everywhere

The IEEE Standard 754

- Single precision (`float`) numbers:

$F^*(2, 24, -126, 127)$ (32 bit) plus 0, ∞ , ...

- Double precision (`double`) numbers:

$F^*(2, 53, -1022, 1023)$ (64 bit) plus 0, ∞ , ...

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

The IEEE Standard 754

- Single precision (`float`) numbers:

$F^*(2, 24, -126, 127)$ (32 bit) plus 0, ∞ , ...

- Double precision (`double`) numbers:

$F^*(2, 53, -1022, 1023)$ (64 bit) plus 0, ∞ , ...

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

Example: 32-bit Representation of a Floating Point Number



± Exponent

Mantisse

± $2^{-126}, \dots, 2^{127}$
 $0, \infty, \dots$

1.000000000000000000000000000000
...
1.111111111111111111111111111111

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";
```

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";
```


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

Rule 2

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

Rule 2

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

$$\begin{array}{r} 1.000 \cdot 2^5 \\ +1.000 \cdot 2^0 \end{array}$$

Rule 2

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

$$\begin{aligned} & 1.000 \cdot 2^5 \\ & + 1.000 \cdot 2^0 \\ & = 1.00001 \cdot 2^5 \end{aligned}$$

Rule 2

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

$$\begin{aligned} & 1.000 \cdot 2^5 \\ & + 1.000 \cdot 2^0 \\ & = 1.00001 \cdot 2^5 \end{aligned}$$

“=” $1.000 \cdot 2^5$ (Rounding on 4 places)

Rule 2

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

$$\begin{aligned} & 1.000 \cdot 2^5 \\ & + 1.000 \cdot 2^0 \\ & = 1.00001 \cdot 2^5 \end{aligned}$$

“=” $1.000 \cdot 2^5$ (Rounding on 4 places)

Addition of 1 does not have any effect!

- The n -th harmonic number is

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

- The n -th harmonic number is

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

- The n -th 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

```
std::cout << "Compute H_n for n =? ";
unsigned int n;
std::cin >> n;

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

float bs = 0;
for (unsigned int i = n; i >= 1; --i)
    bs += 1.0f / i;
std::cout << "Backward sum = " << bs << "\n";
```

Harmonic Numbers

Rule 2

```
std::cout << "Compute H_n for n =? ";  
unsigned int n;  
std::cin >> n;
```

Input: **10000000**

```
float fs = 0;  
for (unsigned int i = 1; i <= n; ++i)  
    fs += 1.0f / i;  
std::cout << "Forward sum = " << fs << "\n";
```

forwards: **15.4037**

```
float bs = 0;  
for (unsigned int i = n; i >= 1; --i)  
    bs += 1.0f / i;  
std::cout << "Backward sum = " << bs << "\n";
```

backwards: **16.686**

Harmonic Numbers

Rule 2

```
std::cout << "Compute H_n for n =? ";  
unsigned int n;  
std::cin >> n;
```

Input: **100000000**

```
float fs = 0;  
for (unsigned int i = 1; i <= n; ++i)  
    fs += 1.0f / i;  
std::cout << "Forward sum = " << fs << "\n";
```

forwards: **15.4037**

```
float bs = 0;  
for (unsigned int i = n; i >= 1; --i)  
    bs += 1.0f / i;  
std::cout << "Backward sum = " << bs << "\n";
```

backwards: **18.8079**

Observation:

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

Observation:

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

Observation:

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

Explanation:

- For $1 + 1/2 + 1/3 + \dots$, later terms are too small to actually contribute
- Problem similar to $2^5 + 1 \text{ “=” } 2^5$

Observation:

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

Explanation:

- For $1 + 1/2 + 1/3 + \dots$, later terms are too small to actually contribute
- Problem similar to $2^5 + 1 \text{ “=” } 2^5$

Observation:

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

Explanation:

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

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

9. Functions I

Defining and Calling Functions, Evaluation of Function Calls, the
Type `void`

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;

std::cout << a << "^" << n << " = " << result << ".\n";
```

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;
```

```
std::cout << a << "^" << n << " = " << result << ".\n";
```

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 << " = " << 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

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

Function to Compute Powers

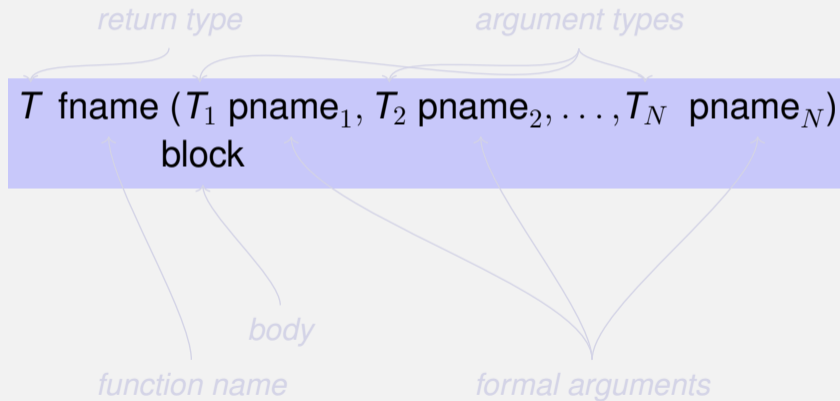
```
// Prog: callpow.cpp
// Define and call a function for computing powers.
#include <iostream>
```

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

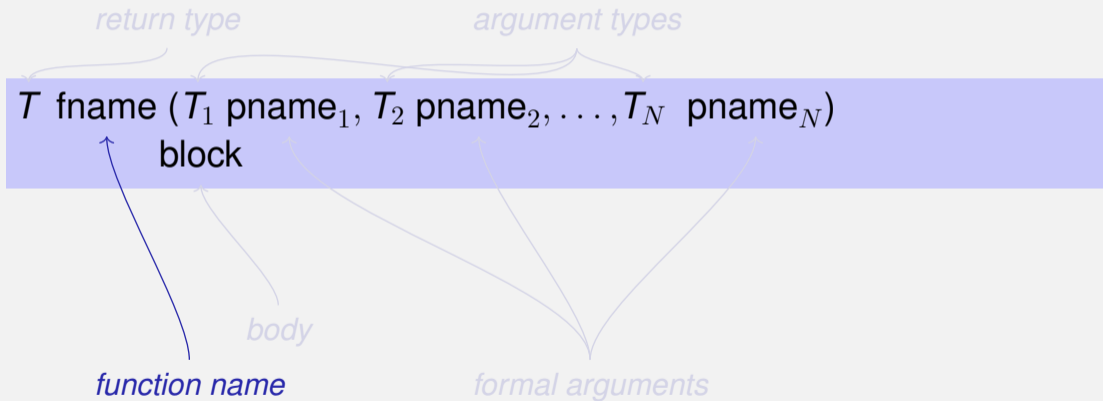
```
int main()
{
    std::cout << pow( 2.0, -2) << "\n"; // outputs 0.25
    std::cout << pow( 1.5, 2) << "\n"; // outputs 2.25
    std::cout << pow(-2.0, 9) << "\n"; // outputs -512

    return 0;
}
```

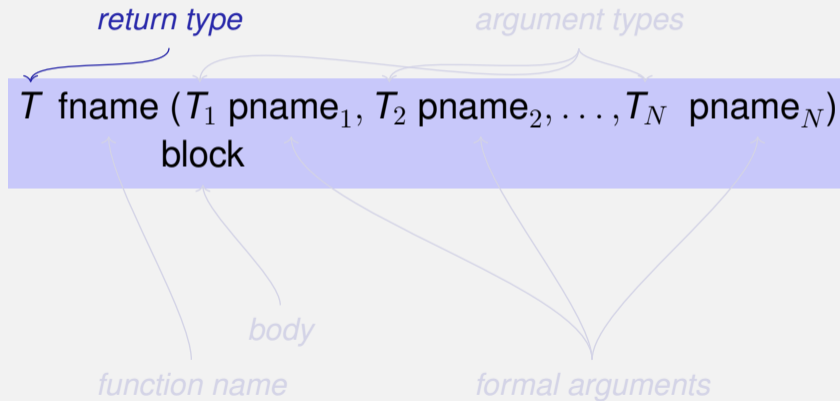
Function Definitions



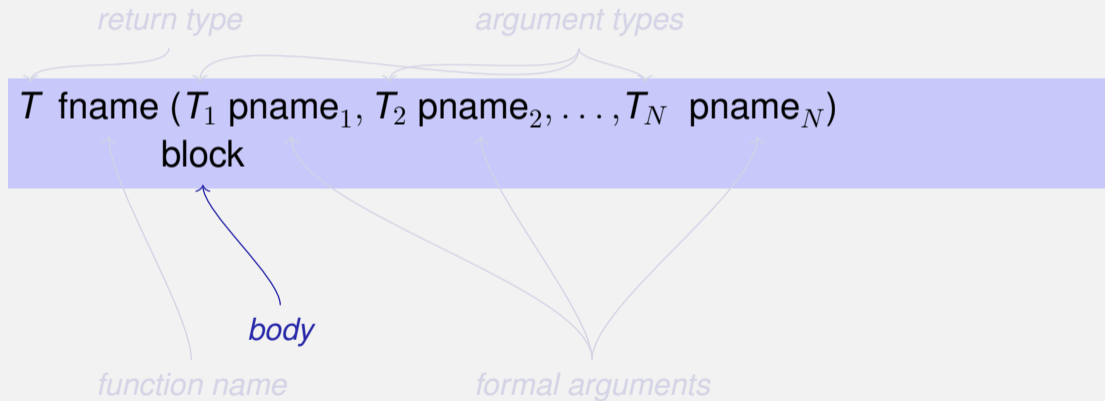
Function Definitions



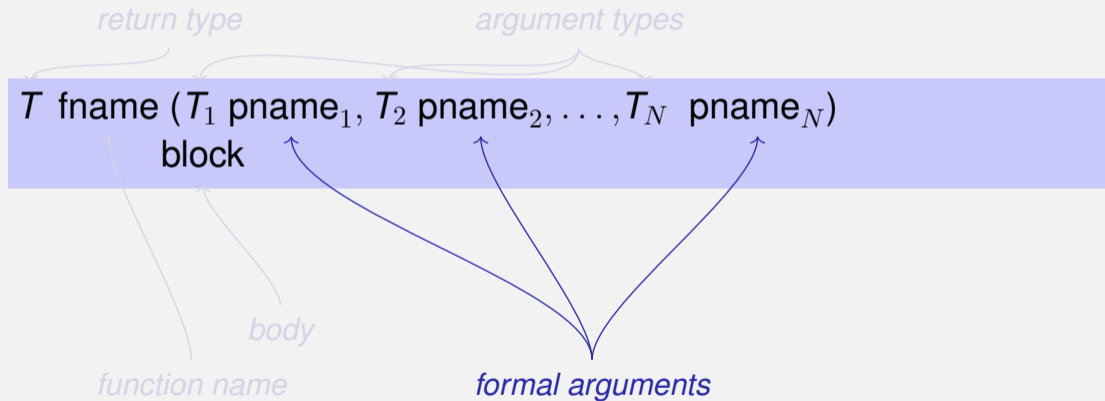
Function Definitions



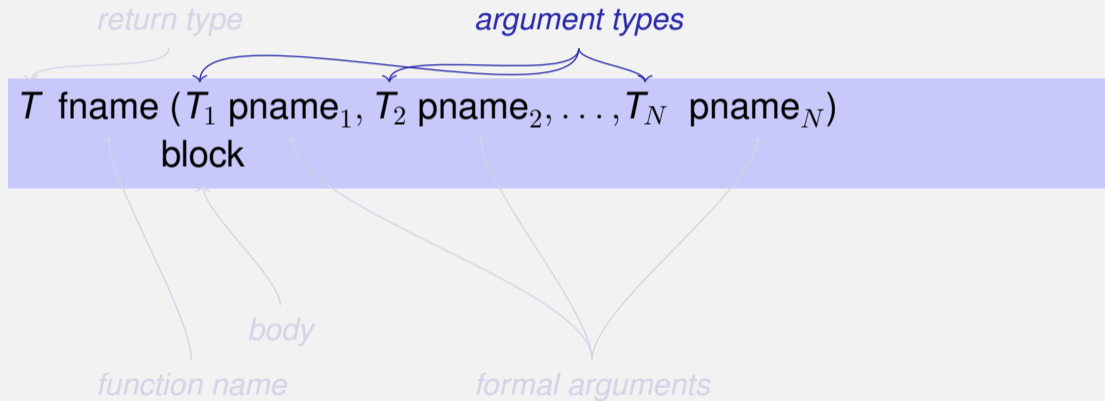
Function Definitions



Function Definitions



Function Definitions



Xor

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

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;
}
```

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 (expression1, expression2, ..., expressionN)`

- 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.

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

Function Calls

`fname (expression1, expression2, ..., expressionN)`

- 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.

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

Function Calls

`fname (expression1, expression2, ..., expressionN)`

- 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.

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
 \hookrightarrow *call-by-value* (also *pass-by-value*), more on this soon
- The function call is an R-value.

Function Calls

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

- Call arguments are R-values
 \hookrightarrow *call-by-value* (also *pass-by-value*), more on this soon
- The function call is an R-value.

*f*name: R-value \times R-value $\times \dots \times$ R-value \longrightarrow R-value

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)
```

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



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)
```

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;
}
```

b=2.0, e=-2

// ok

...

pow (2.0, -2)

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;
}
```



result=1.0

```
...
pow (2.0, -2)
```

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;
}
```



e == -2

```
...
pow (2.0, -2)
```

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;
}
```



b=0.5

```
...
pow (2.0, -2)
```

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;
}
```



e=2

...

```
pow (2.0, -2)
```


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;
}
```



i=0

...

```
pow (2.0, -2)
```

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;
}
```

i=0

result=0.5

...

```
pow (2.0, -2)
```

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;
}
```

i=1

...

```
pow (2.0, -2)
```

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;
}
```

i=1

result=0.25

...

```
pow (2.0, -2)
```

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;
}
```

i=2

...

```
pow (2.0, -2)
```

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;
}
```

→ result=0.25

...

```
pow (2.0, -2)
```

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;
}
```

result=0.25

...

pow (2.0, -2)

Return

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)

Return

value: 0.25

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)  value: 0.25

Scope of Formal Arguments

```
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;  
}
```

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;
}
```

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

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

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

The type void

```
// 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;
}
```

The type `void`

- Fundamental type with empty value range

The type `void`

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

void-Functions

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

10. Functions II

Pre- and Postconditions Stepwise Refinement, Scope, Libraries and Standard Functions

Preconditions

precondition:

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

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.

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

- 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

- 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 $b \neq 0$

White Lies...

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

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...

```
// 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 be
```

Mathematical conditions as a compromise between formal correctness and lax practice

Checking Preconditions...

- Preconditions are only comments.

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.

Postconditions with Asserts

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

Postconditions with Asserts

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

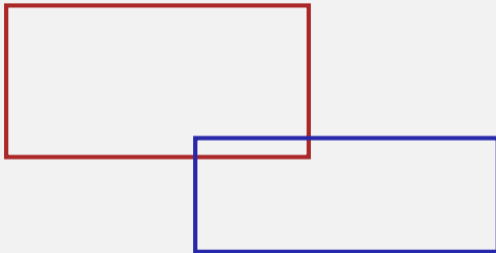
```
// PRE: the discriminant  $p*p/4 - q$  is nonnegative
// POST: returns larger root of the polynomial  $x^2 + p x + q$ 
double root(double p, double q)
{
    assert(p*p/4 >= q); // precondition
    double x1 = - p/2 + sqrt(p*p/4 - q);
    assert(equals(x1*x1+p*x1+q,0)); // postcondition
    return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

Stepwise Refinement

- A simple *technique* to solve complex problems

Example Problem

Find out if two rectangles intersect!



Top-Down Approach

- Formulate a coarse solution using
 - comments
 - fictitious functions
- Repeated refinement:
 - comments \longrightarrow program text
 - fictitious functions \longrightarrow function definitions

Top-Down Approach

- Formulate a coarse solution using
 - comments
 - fictitious functions
- Repeated refinement:
 - comments \longrightarrow program text
 - fictitious functions \longrightarrow function definitions

Coarse Solution

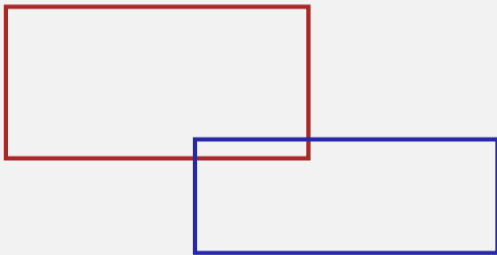
```
int main()
{
    // input rectangles

    // intersection?

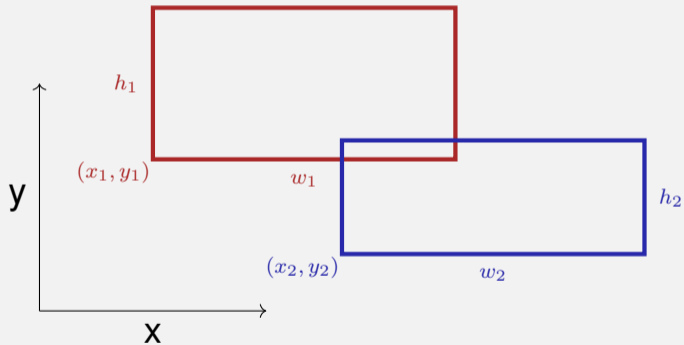
    // output solution

    return 0;
}
```

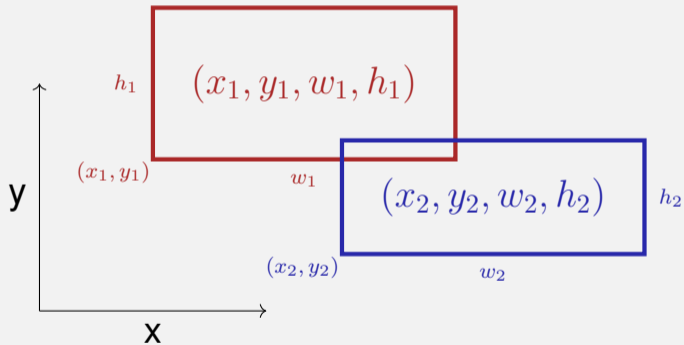
Refinement 1: Input Rectangles



Refinement 1: Input Rectangles

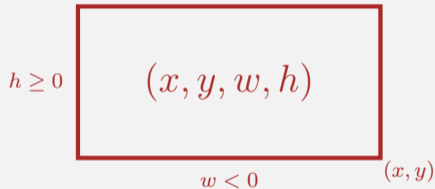


Refinement 1: Input Rectangles



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 ✓

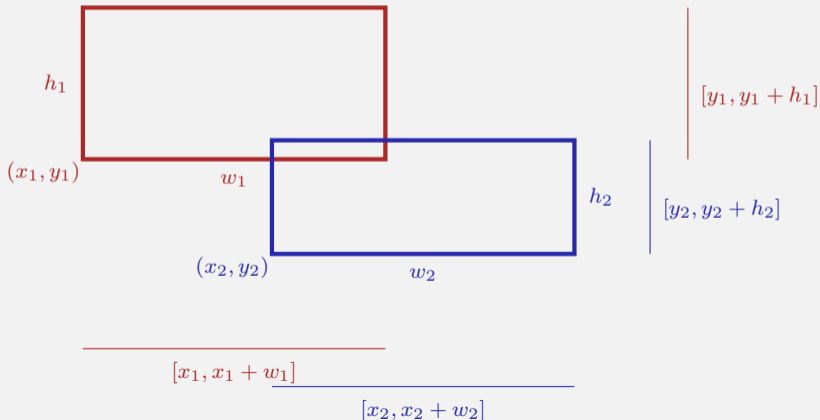
Refinement 3:

... 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: (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

```
// 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;
}
```

Function intervals_intersect ✓

Function rectangles_intersect ✓

Function main ✓

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;  
}
```

already exists in the standard library

```
// POST: the minimum of x and y is returned
```

```
int min(int x, int y){  
    if (x<y) return x; else return y;  
}
```

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

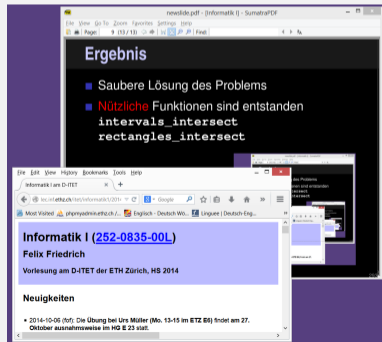
- Clean solution of the problem
- Useful functions have been implemented

`intervals_intersect`

`rectangles_intersect`

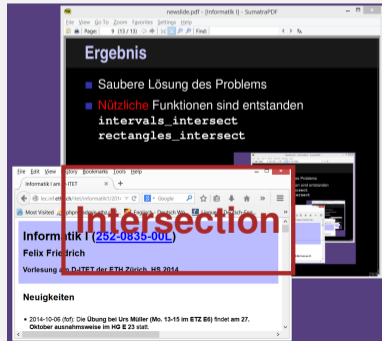
Result

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



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

Scope of a Function

- is the part of the program where a function can be called

Extension by *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); // Gueltingkeitsbereich 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 f(...) // f valid from here
{
    g(...) // g undeclared
}

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



Forward Declarations, why?

Functions that mutually call each other:

```
int g(...); // g valid from here

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

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

Reusability

- Functions such as `rectangles_intersect` and `pow` are useful in many programs.

Reusability

- Functions such as `rectangles_intersect` and `pow` are useful in many programs.
- “Solution”: copy-and-paste the source code

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: Outsource the Function

```
double pow(double b, int e); in  
separate file mymath.cpp
```


Level 1: Include the Function

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

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

```
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;
}
```

Level 1: Include the Function

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

#include <iostream>
#include "mymath.cpp" ← 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 (`mymath.cpp`) into the main program (`callpow2.cpp`).

Disadvantage of Including

- `#include` copies the file (`mymath.cpp`) into the main program (`callpow2.cpp`).
- The compiler has to (re)compile the function definition for each program



```
Terminal — tcsh8.5 — 80x24
Shabdas-iMac:~ admin$ sudo port install amarok
--> Fetching pkgconfig
--> Attempting to fetch pkg-config-0.25.tar.gz from http://aarnet.au.distfiles
    .macports.org/pub/macports/mpdistfiles/pkgconfig
--> Verifying checksum(s) for pkgconfig
--> Extracting pkgconfig
--> Applying patches to pkgconfig
--> Configuring pkgconfig
--> Building pkgconfig
--> Staging pkgconfig into destdir
--> Installing pkgconfig @0.25_1
--> Deactivating pkgconfig @0.23_1
--> Activating pkgconfig @0.25_1
--> Cleaning pkgconfig
--> Computing dependencies for openssl
--> Fetching openssl
--> Attempting to fetch openssl-1.0.0c.tar.gz from http://aarnet.au.distfiles
    .macports.org/pub/macports/mpdistfiles/openssl
--> Verifying checksum(s) for openssl
--> Extracting openssl
--> Applying patches to openssl
--> Configuring openssl
--> Building openssl
--> Staging openssl into destdir
```

Level 2: Separate Compilation

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

mymath.cpp

g++ -c mymath.cpp

```
001110101100101010  
000101110101000111  
00010110001111111  
111100001101010001  
111111101000111010  
010101101011010001  
100101111100101010
```

mymath.o

Level 2: Separate Compilation

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

mymath.h

Level 2: Separate Compilation

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

callpow3.cpp



```
001110101100101010
000101110101000111
000101100010111111
Funktion main
111100001101010001
010101101011010001
100011111010101010
rufe pow auf!
111111101000111010
```

callpow3.o

The linker unites...

```
001110101100101010
000101110101000111
000101110101000111
111100001101010001
111111101000111010
010101101011010001
100101111100101010
```

mymath.o

+

```
001110101100101010
000101110101000111
000101110101000111
111100001101010001
010101101011010001
100101111100101010
111111101000111010
```

callpow3.o

... what belongs together

```
001110101100101010
000101110101000111
0001011 Funktion pow 1
111100001101010001
111111101000111010
010101101011010001
100101111100101010
```

mymath.o

+

```
001110101100101010
000101110101000111
0001011 Funktion main
111100001101010001
010101101011010001
100rufe pow auf! 1010
111111101000111010
```

callpow3.o

=

```
001110101100101010
000101110101000111
0001011 Funktion pow 1
111100001101010001
111111101000111010
010101101011010001
100101111100101010
001110101100101010
000101110101000111
0001011 Funktion main
111100001101010001
010101101011010001
100rufe addr auf! 1010
111111101000111010
```

Executable callpow3

Availability of Source Code?

Observation

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

Availability of Source Code?

Observation

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

Many vendors of libraries do not provide source code.

Availability of Source Code?

Observation

`mymath.cpp` (source code) is not required any more when the `mymath.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

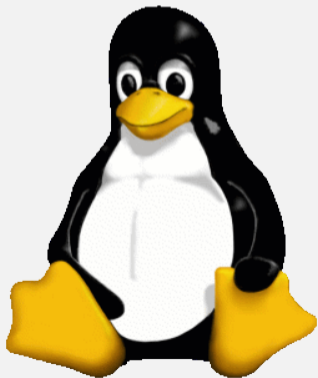
- Source code is generally available.

Open-Source Software

- Source code is generally available.
- Only this allows the continued development of code by users and dedicated “hackers”.

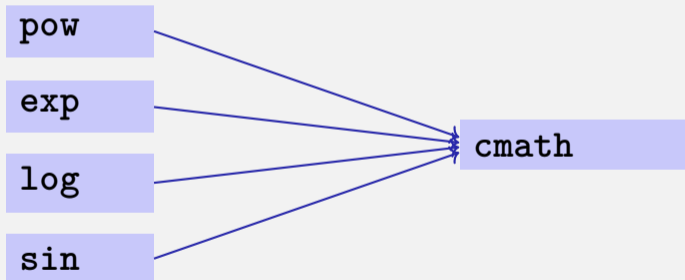
Open-Source Software

- Source code is generally available.
- Only this allows the continued development of code by users and dedicated “hackers”.



Libraries

- Logical grouping of similar functions



Name Spaces...

```
// cmath
namespace std {

    double pow(double b, int e);

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

... Avoid Name Conflicts

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

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

Functions from the Standard Library

- help to avoid re-inventing the wheel (such as with `std::pow`);
- lead to interesting and efficient programs in a simple way;

Functions from the Standard Library

- help to avoid re-inventing the wheel (such as with `std::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.

Example: Prime Number Test with sqrt

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

```
unsigned int bound = std::sqrt(n);  
unsigned int d;  
for (d = 2; d <= bound && 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 $\{2, \dots, \lfloor \sqrt{n} \rfloor\}$ 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).

```
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);  
}
```


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);
}
```

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.

Sneak Preview: Reference Types

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

Sneak Preview: Reference Types

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



Reference types (e.g. `int&`)

11. Reference Types

Reference Types: Definition and Initialization, Pass By Value, Pass 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

Reference Types

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

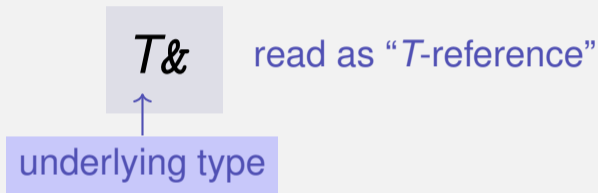
Reference Types

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

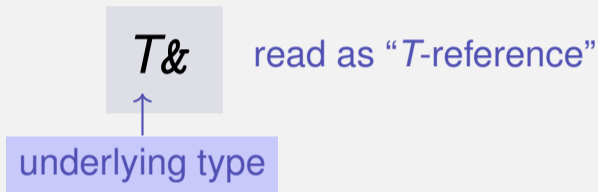
Reference Types



Reference Types: Definition

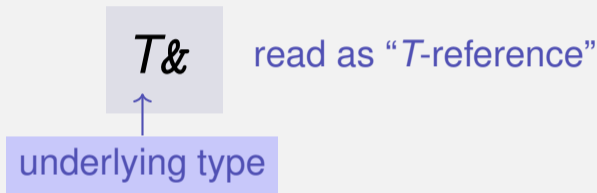


Reference Types: Definition



- $T\&$ has the same range of values and functionality as T , ...

Reference Types: Definition



- $T\&$ 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  
darth_vader = 22;  
  
std::cout << anakin_skywalker;
```


Anakin Skywalker alias Darth Vader

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

```
std::cout << anakin_skywalker;
```

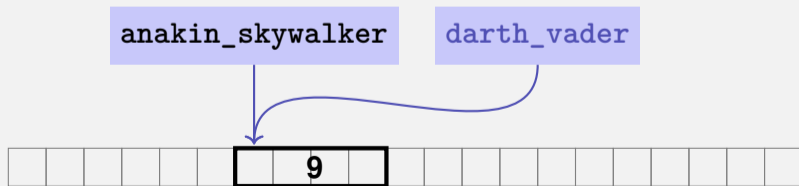
anakin_skywalker



Anakin Skywalker alias Darth Vader

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

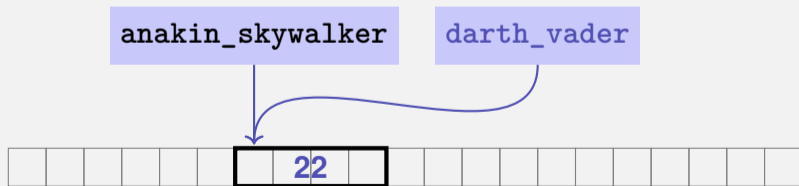
```
std::cout << anakin_skywalker;
```



Anakin Skywalker alias Darth Vader

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

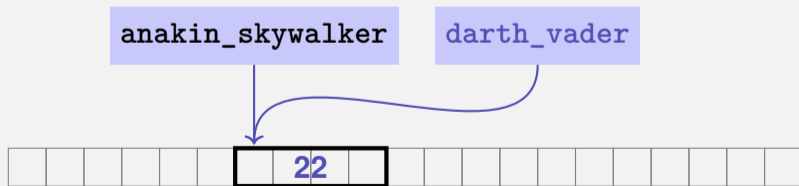
```
std::cout << anakin_skywalker;
```



Anakin Skywalker alias Darth Vader

```
int anakin_skywalker = 9;  
int& darth_vader = anakin_skywalker; // alias  
darth_vader = 22;  
std::cout << anakin_skywalker;
```

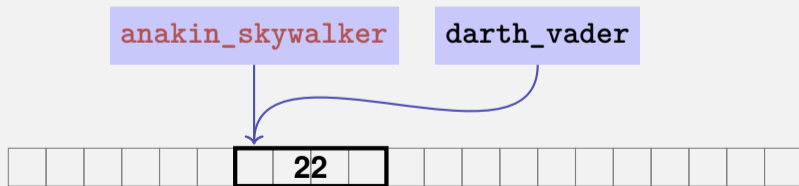
assignment to the L-value behind the alias



Anakin Skywalker alias Darth Vader

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

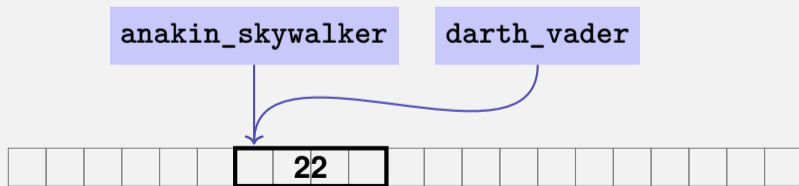
```
std::cout << anakin_skywalker; // 22
```



Anakin Skywalker alias Darth Vader

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

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



Reference Types: Initialization and Assignment

```
int& darth_vader = anakin_skywalker;
```

- A variable of **reference type** (a *reference*) can only be initialized with an **L-Value** .

Reference Types: Initialization and Assignment

```
int& darth_vader = anakin_skywalker;
```

- 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).

Reference Types: Initialization 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
```

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
```

Pass by Reference

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

...
int j = 5;
increment (j);
std::cout << j << "\n"; // 6
```

Pass by Reference

```
void increment (int& i)
{
    ++i;
}
...
int j = 5;
increment (j);
std::cout << j << "\n"; // 6
```



Pass by Reference

```
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
```



Pass by Reference

```
void increment (int& i)
{
    ++i;
}
...
int j = 5;
increment (j);
std::cout << j << "\n"; // 6
```



Pass by Reference

```
void increment (int& i)
{
    ++i;
}
...
int j = 5;
increment (j);
std::cout << j << "\n"; // 6
```



Pass by Reference

Formal argument has reference type:

⇒ **Pass by Reference**

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

Pass by Value


Formal argument does not have a reference type:

⇒ **Pass by Value**

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

References in the Context of intervals_intersect

```
// 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); // Assignments
    h = std::min (b1, b2); // via references
    return l <= h;
}
...
int lo = 0; int hi = 0;
if (intervals_intersect (lo, hi, 0, 2, 1, 3)) // Initialization
    std::cout << "[" << lo << "," << hi << "]" << "\n"; // [1,2]
```



The diagram shows two horizontal intervals on a number line. The first interval is represented by a thick green line segment starting at a_1 and ending at b_1 . The second interval is represented by a thinner green line segment starting at a_2 and ending at b_2 . The intersection of these two intervals is the segment between l and h , which is the maximum of a_1 and a_2 and the minimum of b_1 and b_2 .

References in the Context of intervals_intersect

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

```
bool intervals_intersect (int& l, int& h,
                          int a1, int b1, int a2, int b2) {
    sort (a1, b1); // Initialization
    sort (a2, b2); // Initialization
    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)

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

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

Return by Value / Reference

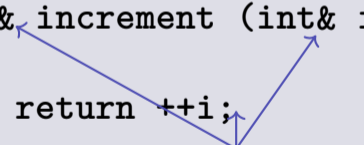
- Even the return type of a function can be a reference type (return by reference)
- In this case the function call itself is an L-value

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


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;
}
```

A diagram with blue arrows. One arrow points from the 'int&' in the function signature to the '++i' in the return statement. Another arrow points from the '++i' in the return statement to the 'int&' in the function signature. A third arrow points from the '++i' in the return statement to the '++i' in the return statement.

exactly the semantics of the pre-increment

Temporary Objects

What is wrong here?

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

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!

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
```

Temporary Objects

What is wrong here?

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



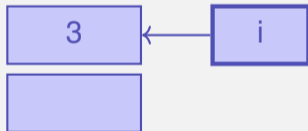
```
int k = 3;
int& j = foo (k); // j is an alias of a zombie
std::cout << j << "\n"; // undefined behavior
```

Temporary Objects

What is wrong here?

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

value of the actual parameter is pushed onto the *call stack*



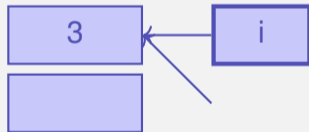
```
int k = 3;
int& j = foo (k); // j is an alias of a zombie
std::cout << j << "\n"; // undefined behavior
```

Temporary Objects

What is wrong here?

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

i is returned as reference



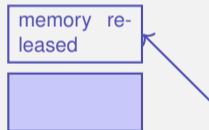
```
int k = 3;
int& j = foo (k); // j is an alias of a zombie
std::cout << j << "\n"; // undefined behavior
```

Temporary Objects

What is wrong here?

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

...and disappears from the stack



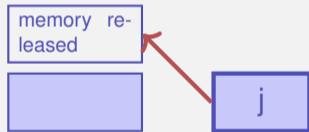
```
int k = 3;
int& j = foo (k); // j is an alias of a zombie
std::cout << j << "\n"; // undefined behavior
```


Temporary Objects

What is wrong here?

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

j becomes alias to released memory



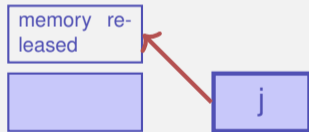
```
int k = 3;
int& j = foo (k); // j is an alias of a zombie
std::cout << j << "\n"; // undefined behavior
```

Temporary Objects

What is wrong here?

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

value of j is output



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

The Reference Guideline

Reference Guideline

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

Const-References

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

Const-References

- have type `const T &`
- type can be interpreted as “`(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-References

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

```
const T& r = rvalue;
```

`r` is initialized with the address of a temporary object with the value of the `rvalue` (pragmatic)

When `const T&` ?

Rule

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

When `const T&` ?

Rule

Argument type `const T&` (pass by *read-only* reference) is used for efficiency reasons instead of `T` (pass 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


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;  
i = 6;
```




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**

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
```

12. Vectors and Strings I

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

Vectors: Motivation

- Now we can iterate over numbers

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

Vectors: Motivation

- Now we can iterate over numbers

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

- ... but not yet over data!

Vectors: Motivation

- Now we can iterate over numbers

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

- ... but not yet over data!
- Vectors store *homogeneous* data.

Vectors: a first Application

The Sieve of Erathostenes

- computes all prime numbers $< n$

Vectors: 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
---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Vectors: 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
----------	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Cross out all real factors of 2 ...

Vectors: 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
----------	---	--------------	---	--------------	---	--------------	---	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----

Cross out all real factors of 2 ...

Vectors: 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
---	---	--------------	---	--------------	---	--------------	---	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----

... and go to the next number

Vectors: 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
---	---	--------------	---	--------------	---	--------------	---	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----	---------------	----

cross out all real factors of 3 ...

Vectors: 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
---	---	--------------	---	--------------	---	--------------	--------------	---------------	----	---------------	----	---------------	---------------	---------------	----	---------------	----	---------------	---------------	---------------	----

cross out all real factors of 3 ...

Vectors: 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
---	---	--------------	---	--------------	---	--------------	--------------	---------------	----	---------------	----	---------------	---------------	---------------	----	---------------	----	---------------	---------------	---------------	----

... and go to the next number

Vectors: 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.

Vectors: 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
---	---	--------------	---	--------------	---	--------------	--------------	---------------	----	---------------	----	---------------	---------------	---------------	----	---------------	----	---------------	---------------	---------------	----

- Question: how do we cross out numbers ??

Vectors: 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
---	---	--------------	---	--------------	---	--------------	--------------	---------------	----	---------------	----	---------------	---------------	---------------	----	---------------	----	---------------	---------------	---------------	----

- Question: how do we cross out numbers ??
- Answer: with a *vector*.

Erathosthenes with Vectors: Initialization

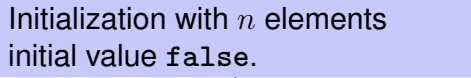
...

```
#include <vector>
```

...

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

Initialization with n elements
initial value false.



↑
element type in triangular brackets



Erathosthenes with Vectors: Computation

```
for (unsigned int i = 2; i < crossed_out.size(); ++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;
    }
```

Memory Layout of a Vector

- A vector occupies a *contiguous* memory area

Memory Layout of a Vector

- A vector occupies a *contiguous* memory area

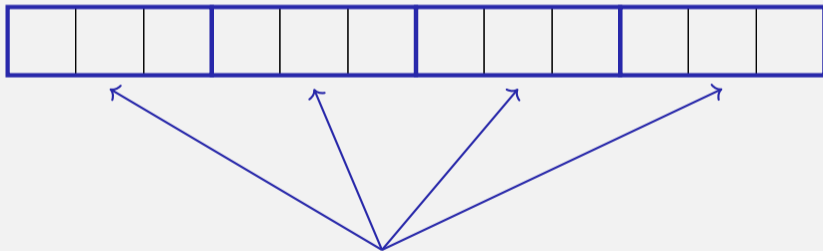


memory cells for a value of type T each

Memory Layout of a Vector

- A vector occupies a *contiguous* memory area

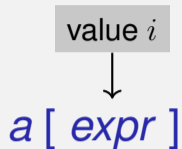
example: a vector with 4 elements



memory cells for a value of type T each

Random Access

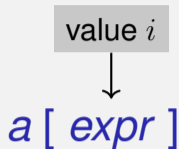
The L-value



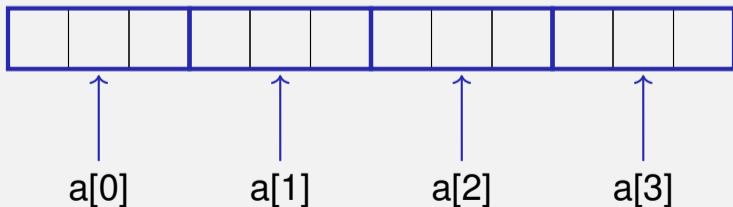
has type T and refers to the i -th element of the vector a (counting from 0!)

Random Access

The L-value



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



Random Access

$a [expr]$

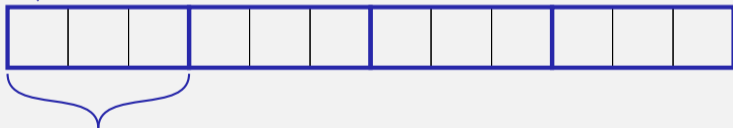
The value i of $expr$ is called *index*.

$[]$: subscript operator

Random Access

- Random access is very efficient:

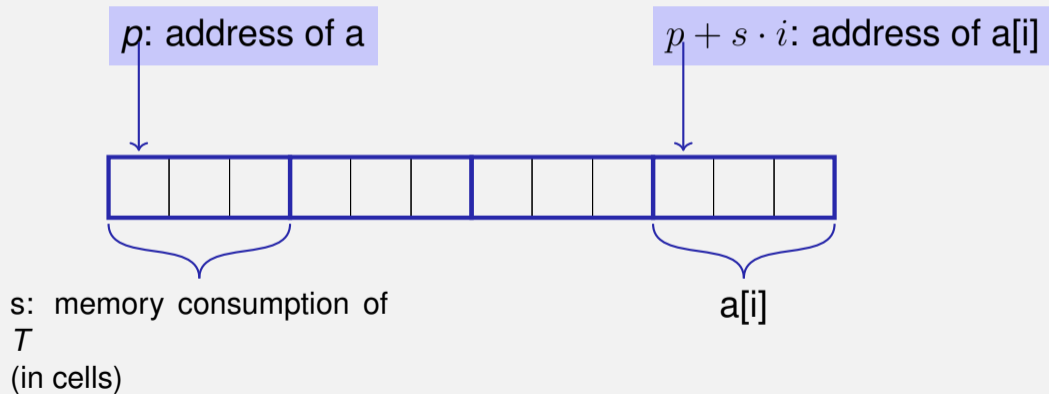
p : address of a , i.e. address of the first memory cell



s : memory consumption of
 T
(in cells)

Random Access

- Random access is very efficient:



Vector Initialization

- `std::vector<int> a (5);`
The five elements of a are zero initialized)

Vector Initialization

- `std::vector<int> a (5);`
The five elements of `a` are zero initialized)
- `std::vector<int> a (5, 2);`
the 5 elements of `a` are initialized with 2.
- `std::vector<int> a {4, 3, 5, 2, 1};`
the vector is initialized with an *initialization list*.
- `std::vector<int> a;`
An initially empty vector is created.

Vector Initialization

- `std::vector<int> a (5);`
The five elements of `a` are zero initialized)
- `std::vector<int> a (5, 2);`
the 5 elements of `a` are initialized with 2.
- `std::vector<int> a {4, 3, 5, 2, 1};`
the vector is initialized with an *initialization list*.
- `std::vector<int> a;`
An initially empty vector is created.

Vector Initialization

- `std::vector<int> a (5);`
The five elements of `a` are zero initialized)
- `std::vector<int> a (5, 2);`
the 5 elements of `a` are initialized with 2.
- `std::vector<int> a {4, 3, 5, 2, 1};`
the vector is initialized with an *initialization list*.
- `std::vector<int> a;`
An initially empty vector is created.

Attention

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

```
std::vector arr (10);  
for (int i=0; i<=10; ++i)  
    arr[i] = 30;
```

Attention

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

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

Attention

Bound Checks

When using a subscript operator on a vector, it is the sole *responsibility of the programmer* to check the validity of element accesses.

Consequences of illegal index accesses

"out of bounds" array exploit

Alle Videos Bilder News Shopping Mehr Einstellungen Tools

Ungefähr 127'000 Ergebnisse (0.30 Sekunden)

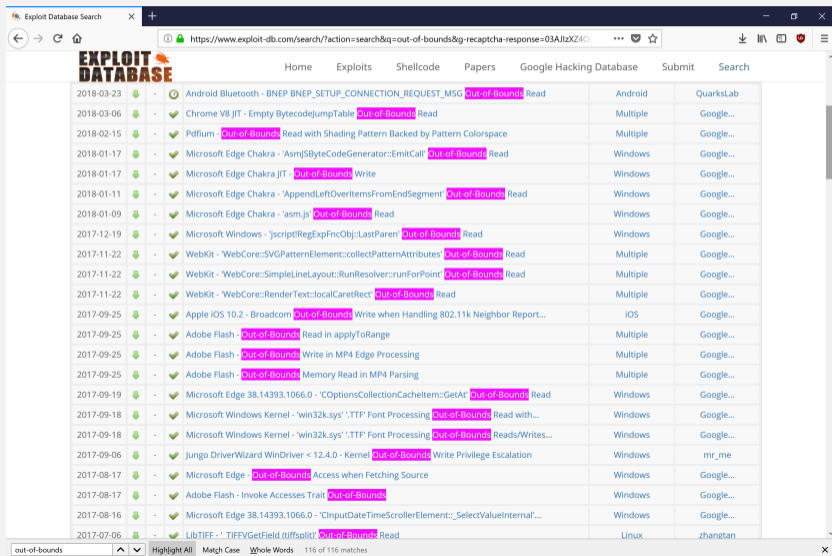
CWE - CWE-125: Out-of-bounds Read (3.0)
<https://cwe.mitre.org> › [CWE List](#) ▼ [Diese Seite übersetzen](#)
However, this method only verifies that the given array index is less than the maximum length of the array but does not check for the minimum value (CWE-839). This will allow a negative value to be accepted as the input array index, which will result in a **out of bounds** read (CWE-125) and may allow access to sensitive ...

CWE - CWE-787: Out-of-bounds Write (3.0)
<https://cwe.mitre.org> › [CWE List](#) ▼ [Diese Seite übersetzen](#)
This typically occurs when the pointer or its index is incremented or decremented to a position beyond the bounds of the buffer or when pointer arithmetic results in a position outside of the valid memory location to name a few. This may result in corruption of sensitive information, a crash, or code execution among other ...

c - How dangerous is it to access an array out of bounds? - Stack ...
<https://stackoverflow.com/.../how-dangerous-is-it-to-access-an-arr...> ▼ [Diese Seite übersetzen](#)
As far as the ISO C standard (the official definition of the language) is concerned, accessing an array outside its bounds has "undefined behavior". The literal meaning of this is: behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no ...

Bypassing ASLR with CVE-2015-0071: An Out-of-Bounds Read ...
<https://blog.trendmicro.com/.../bypassing-aslr-with-cve-2015-007...> ▼ [Diese Seite übersetzen](#)
Bypassing ASLR with CVE-2015-0071: An **Out-of-Bounds** Read Vulnerability. Posted on: March 13, 2015 at 8:10 am ... 0x00: pVtable: 0x08: lenath (the strina lenath): 0x0c: oStrina (pointer to the strina

Consequences of illegal index accesses



The screenshot shows the Exploit Database Search interface. The search query is "out-of-bounds". The results table lists various exploits with their dates, descriptions, and associated platforms. The text "Out-of-Bounds" is highlighted in pink in the original image.

Date	Exploit Description	Platform	Author
2018-03-23	Android Bluetooth - BNEP BNEP_SETUP_CONNECTION_REQUEST_MSG Out-of-Bounds Read	Android	QuarksLab
2018-03-06	Chrome V8 JIT - Empty BytecodeJumpTable Out-of-Bounds Read	Multiple	Google...
2018-02-15	Pdfium - Out-of-Bounds Read with Shading Pattern Backed by Pattern Colorspace	Multiple	Google...
2018-01-17	Microsoft Edge Chakra - 'AsmJSByteCodeGenerators:EmitCall' Out-of-Bounds Read	Windows	Google...
2018-01-17	Microsoft Edge Chakra JIT - Out-of-Bounds Write	Windows	Google...
2018-01-11	Microsoft Edge Chakra - 'AppendLeftOverItemsFromEndSegment' Out-of-Bounds Read	Windows	Google...
2018-01-09	Microsoft Edge Chakra - 'asm.js' Out-of-Bounds Read	Windows	Google...
2017-12-19	Microsoft Windows - 'jscrip!RegExpFncObj::LastParen' Out-of-Bounds Read	Windows	Google...
2017-11-22	WebKit - 'WebCore::SVGPatternElement::collectPatternAttributes' Out-of-Bounds Read	Multiple	Google...
2017-11-22	WebKit - 'WebCore::SimpleLineLayout::RunResolver::runForPoint' Out-of-Bounds Read	Multiple	Google...
2017-11-22	WebKit - 'WebCore::RenderText::localCaretRect' Out-of-Bounds Read	Multiple	Google...
2017-09-25	Apple iOS 10.2 - Broadcom Out-of-Bounds Write when Handling 802.11k Neighbor Report...	iOS	Google...
2017-09-25	Adobe Flash - Out-of-Bounds Read in applyToRange	Multiple	Google...
2017-09-25	Adobe Flash - Out-of-Bounds Write in MP4 Edge Processing	Multiple	Google...
2017-09-25	Adobe Flash - Out-of-Bounds Memory Read in MP4 Parsing	Multiple	Google...
2017-09-19	Microsoft Edge 38.14393.1066.0 - 'COptionsCollectionCacheItem::GetAt' Out-of-Bounds Read	Windows	Google...
2017-09-18	Microsoft Windows Kernel - 'win32k.sys'.TTF Font Processing Out-of-Bounds Read with...	Windows	Google...
2017-09-18	Microsoft Windows Kernel - 'win32k.sys'.TTF Font Processing Out-of-Bounds Reads/Writes...	Windows	Google...
2017-09-06	Jungo DriverWizard WinDriver < 12.4.0 - Kernel Out-of-Bounds Write Privilege Escalation	Windows	mr_me
2017-08-17	Microsoft Edge - Out-of-Bounds Access when Fetching Source	Windows	Google...
2017-08-17	Adobe Flash - Invoke Accesses Trait Out-of-Bounds	Windows	Google...
2017-08-16	Microsoft Edge 38.14393.1066.0 - 'CinputDateTimeScrollerElement::_SelectValueInternal'...	Windows	Google...
2017-07-06	LibTIFF - 'TIFFGetField(tiffsplit)' Out-of-Bounds Read	Linux	zhangtan

Search filters: out-of-bounds | Highlight All | Match Case | Whole Words | 116 of 116 matches

Vectors are Comfortable

```
std::vector<int> v (10);  
v.at(5) = 3; // with bound check  
v.push_back(8); // 8 is appended  
std::vector<int> w = v; // w is initialized with v  
int sz = v.size(); // sz = 11
```

Characters and Texts

- We have seen texts before:

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


Characters and Texts

- We have seen texts before:

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

String-Literal

Characters and Texts

- We have seen texts before:

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

String-Literal

- can we really work with texts?

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: `std::string` \approx vector of `char` elements

The type `char` (“character”)

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

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'

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 diagram shows the code `char c = 'a'` with two blue arrows. One arrow points from the text "defines variable c of type char with value 'a'" to the variable `c`. The other arrow points from the text "literal of type char" to the character literal `'a'`.

The type `char` (“character”)

is formally an integer type

- values convertible to `int` / `unsigned int`

The type `char` (“character”)

is formally an integer type

- values convertible to `int` / `unsigned int`
- values typically occupy 8 Bit

The type `char` (“character”)

is formally an integer type

- values convertible to `int` / `unsigned int`
- values typically occupy 8 Bit

The type `char` (“character”)

is formally an integer type

- values convertible to `int` / `unsigned int`
- values typically occupy 8 Bit

domain:

$\{-128, \dots, 127\}$ or $\{0, \dots, 255\}$

The ASCII-Code

- defines concrete conversion rules
`char` \longrightarrow `int` / `unsigned int`

The ASCII-Code

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

Zeichen \longrightarrow $\{0, \dots, 127\}$

'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
```

The ASCII-Code

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

Zeichen \longrightarrow $\{0, \dots, 127\}$

'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.

# 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

# 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 encode further 128 characters – this is history



# 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 |

# 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 |

# Einige Zeichen in UTF-8




| Symbol | Codierung (jeweils 16 Bit) |
|--------|----------------------------|
|--------|----------------------------|

|   |                            |
|---|----------------------------|
| س | 11101111 10101111 10111001 |
|---|----------------------------|

# Einige Zeichen in UTF-8

| Symbol                                                                            | Codierung (jeweils 16 Bit) |
|-----------------------------------------------------------------------------------|----------------------------|
|  | 11101111 10101111 10111001 |
|  | 11100010 10011000 10100000 |
|  | 11100010 10011000 10000011 |
|  | 11100010 10011000 10011001 |

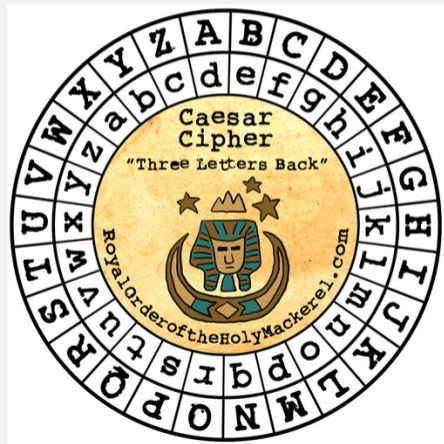
# Einige Zeichen in UTF-8

| Symbol                                                                            | Codierung (jeweils 16 Bit) |
|-----------------------------------------------------------------------------------|----------------------------|
|  | 11101111 10101111 10111001 |
|  | 11100010 10011000 10100000 |
|  | 11100010 10011000 10000011 |
|  | 11100010 10011000 10011001 |
| A                                                                                 | 01000001                   |

# Caesar-Code

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

|     |       |   |     |       |
|-----|-------|---|-----|-------|
| ' ' | (32)  | → | ' ' | (124) |
| '!' | (33)  | → | '}' | (125) |
|     | ⋮     |   |     |       |
| 'D' | (68)  | → | 'A' | (65)  |
| 'E' | (69)  | → | 'B' | (66)  |
|     | ⋮     |   |     |       |
| ~   | (126) | → | '{' | (123) |



```
// pre: divisor > 0
// post: return the remainder of dividend / divisor
// with 0 <= result < divisor
int mod(int dividend, int divisor);

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

```
// pre: divisor > 0
// post: return the remainder of dividend / divisor
// with 0 <= result < divisor
int mod(int dividend, int divisor);

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

"- 32" transforms interval [32, 126] to [0, 94]

"32 +" transforms interval [0, 94] back to [32, 126]

mod(x,95) is the representative of  $x \pmod{95}$  in interval [0, 94]



```
// POST: Each character read from std::cin was shifted cyclically
// by s characters and afterwards written to std::cout
void caesar(int s) {
 std::cin >> std::noskipws; // #include <ios>

 char next;
 while (std::cin >> next) {
 std::cout << shift(next, s);
 }
}
```

spaces and newline characters  
shall *not* be ignored

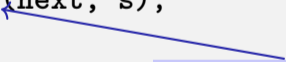
```
// POST: Each character read from std::cin was shifted cyclically
// by s characters and afterwards written to std::cout
void caesar(int s) {
 std::cin >> std::noskipws; // #include <ios>

 char next;
 while (std::cin >> next) ← {
 std::cout << shift(next, s),
 }
}
```

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

```
// POST: Each character read from std::cin was shifted cyclically
// by s characters and afterwards written to std::cout
void caesar(int s) {
 std::cin >> std::noskipws; // #include <ios>

 char next;
 while (std::cin >> next) {
 std::cout << shift(next, s);
 }
}
```



shifts only printable characters.

```
int main() {
 int s;
 std::cin >> s;

 // Shift input by s
 caesar(s);

 return 0;
}
```

Encode: shift by  $n$  (here: 3)

```
3.
Hello World, my password is 1234.
Khoor#Zruog/#p|#sdvvzrug#lv#45671
```

Encode: shift by  $-n$  (here: -3)

```
-3.
Khoor#Zruog/#p|#sdvvzrug#lv#45671
Hello World, my password is 1234.
```

# Caesar-Code: Generalisation

```
void caesar(int s) {
 std::cin >> std::noskipws;

 char next;
 while (std::cin >> next) {
 std::cout << shift(next, s);
 }
}
```

- Currently only from `std::cin` to `std::cout`

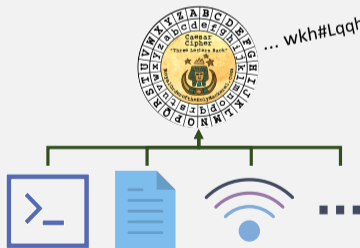
# Caesar-Code: Generalisation

```
void caesar(int s) {
 std::cin >> std::noskipws;

 char next;
 while (std::cin >> next) {
 std::cout << shift(next, s);
 }
}
```

- Currently only from `std::cin` to `std::cout`

- Better: from arbitrary character source (console, file, ...) to arbitrary character sink (console, ...)



Icons: flaticon.com; authors Smashicons, Kirill Kazachek; CC 3.0 BY

# Caesar-Code: Generalisation

```
void caesar(std::istream& in,
 std::ostream& out,
 int s) {

 in >> std::noskipws;

 char next;
 while (in >> next) {
 out << shift(next, s);
 }
}
```

- `std::istream/std::ostream` is an *generic input/output stream* of chars

# Caesar-Code: Generalisation

```
void caesar(std::istream& in,
 std::ostream& out,
 int s) {

 in >> std::noskipws;

 char next;
 while (in >> next) {
 out << shift(next, s);
 }
}
```

- `std::istream/std::ostream` is an *generic input/output stream* of chars
- Function is called with *specific streams*, e.g.: Console (`std::cin/cout`), Files (`std::i/ofstream`), Strings (`std::i/ostringstream`)



# Caesar-Code: Generalisation, Example 1

```
#include <iostream>
```

```
...
```

```
// in void main():
```

```
caesar(std::cin, std::cout, s);
```

Calling the generalised caesar function: from `std::cin` to  
`std::cout`

# Caesar-Code: Generalisation, Example 2

```
#include <iostream>
#include <fstream>
...

// in void main():
std::string from_file_name = ...; // Name of file to read from
std::string to_file_name = ...; // Name of file to write to
std::ifstream from(from_file_name); // Input file stream
std::ofstream to(to_file_name); // Output file stream

caesar(from, to, s);
```

Calling the generalised caesar function: from file to file

# Caesar-Code: Generalisation, Example 3

```
#include <iostream>
#include <sstream>
...

// in void main():
std::string plaintext = "My password is 1234";
std::istringstream from(plaintext);

caesar(from, std::cout, s);
```

Calling the generalised caesar function: from a string to `std::cout`

# 13. Vectors and Strings II

Strings, Multidimensional Vector/Vectors of Vectors, Shortest Paths,  
Vectors as Function Arguments

# Texts

- Text “to be or not to be” could be represented as `vector<char>`

# Texts

- Text “to be or not to be” could be represented as `vector<char>`
- Texts are ubiquitous, however, and thus have their own type in the standard library: `std::string`
- Requires `#include <string>`

# Using `std::string`

- Declaration, and initialisation with a literal:

```
std::string text = "Essen ist fertig!"
```

# Using `std::string`

- Declaration, and initialisation with a literal:

```
std::string text = "Essen ist fertig!"
```

- Initialise with variable length:

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



# Using `std::string`

- Declaration, and initialisation with a literal:

```
std::string text = "Essen ist fertig!"
```

- Initialise with variable length:

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

- Comparing texts:

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

# Using `std::string`

- Querying size:

```
for (unsigned int i = 0; i < text.size(); ++i) ...
```

# Using `std::string`

- Querying size:

```
for (unsigned int i = 0; i < text.size(); ++i) ...
```

- Reading single characters:

```
if (text[0] == 'a') ... // or text.at(0)
```

# Using `std::string`

- Querying size:

```
for (unsigned int i = 0; i < text.size(); ++i) ...
```

- Reading single characters:

```
if (text[0] == 'a') ... // or text.at(0)
```

- Writing single characters:

```
text[0] = 'b'; // or text.at(0)
```

# Using `std::string`

- Concatenate strings:

```
text = ":-";
text += ")";
assert(text == ":-)");
```

- Many more operations; if interested, see <https://en.cppreference.com/w/cpp/string>

# Multidimensional Vectors

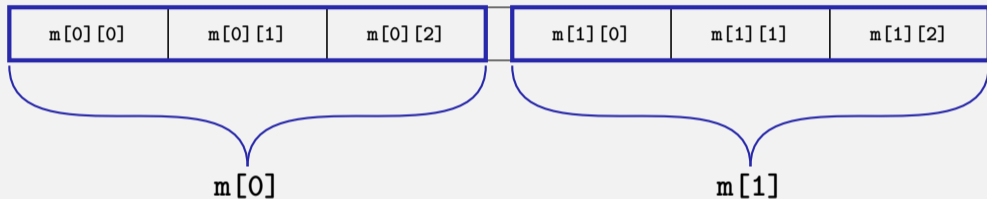
- For storing multidimensional structures such as tables, matrices, ...

- ... *vectors of vectors* can be used:

```
std::vector<std::vector<int>> m; // An empty matrix
```

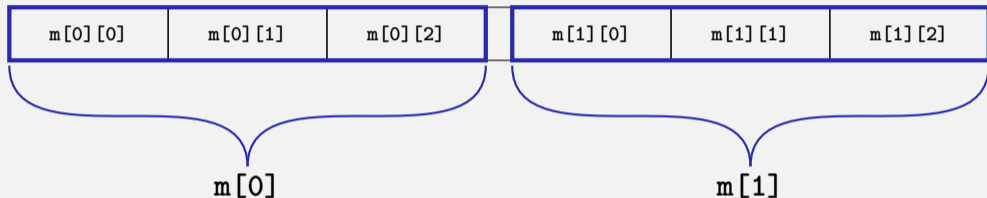
# Multidimensional Vectors

In memory: flat



# Multidimensional Vectors

In memory: flat



in our head: matrix

|   | columns →            |                      |                      |
|---|----------------------|----------------------|----------------------|
|   | 0                    | 1                    | 2                    |
| 0 | <code>m[0][0]</code> | <code>m[0][1]</code> | <code>m[0][2]</code> |
| 1 | <code>m[1][0]</code> | <code>m[1][1]</code> | <code>m[1][2]</code> |



# Multidimensional Vectors: Initialisation Examples

Using literals:

```
// A 3-by-5 matrix
std::vector<std::vector<std::string>> m = {
 {"ZH", "BE", "LU", "BS", "GE"},
 {"FR", "VD", "VS", "NE", "JU"},
 {"AR", "AI", "OW", "IW", "ZG"}
};

assert(m[1][2] == "VS");
```

# Multidimensional Vectors: Initialisation Examples

Fill to specific size:

```
unsigned int a = ...;
unsigned int b = ...;

// An a-by-b matrix with all ones
std::vector<std::vector<int>>
 m(a, std::vector<int>(b, 1));
```

# Multidimensional Vectors: Initialisation Examples

Fill to specific size:

```
unsigned int a = ...;
unsigned int b = ...;

// An a-by-b matrix with all ones
std::vector<std::vector<int>>
 m(a, std::vector<int>(b, 1));
```

(Many further ways of initialising a vector exist)

# Multidimensional Vectors and Type Aliases

- Also possible: vectors of vectors of vectors of ...:  
`std::vector<std::vector<std::vector<...>>>`
- Type names can obviously become looooooong

# Multidimensional Vectors and Type Aliases

- Also possible: vectors of vectors of vectors of ...:  
`std::vector<std::vector<std::vector<...>>>`
- Type names can obviously become looooooong
- The declaration of a *type alias* helps here:

`using Name = Typ;`

Name that can now be used to access the type

existing type

# Type Aliases: Example

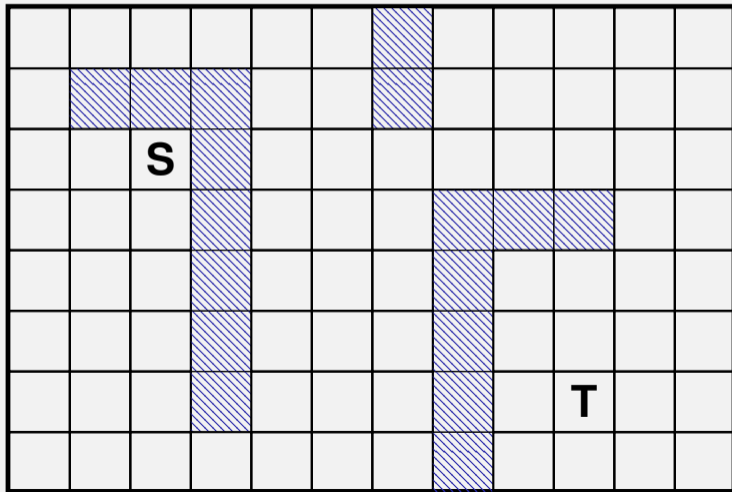
```
#include <iostream>
#include <vector>
using imatrix = std::vector<std::vector<int>>;

// POST: Matrix 'm' was printed to stream 'to'
void print(imatrix m, std::ostream to);

int main() {
 imatrix m = ...;
 print(m, std::cout);
}
```

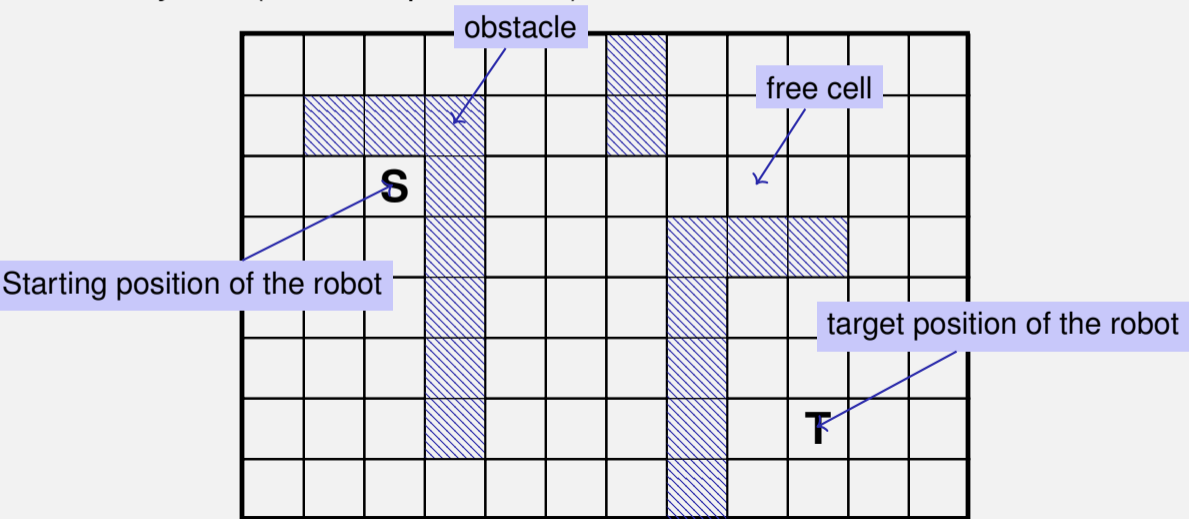
# Application: Shortest Paths

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



# Application: Shortest Paths

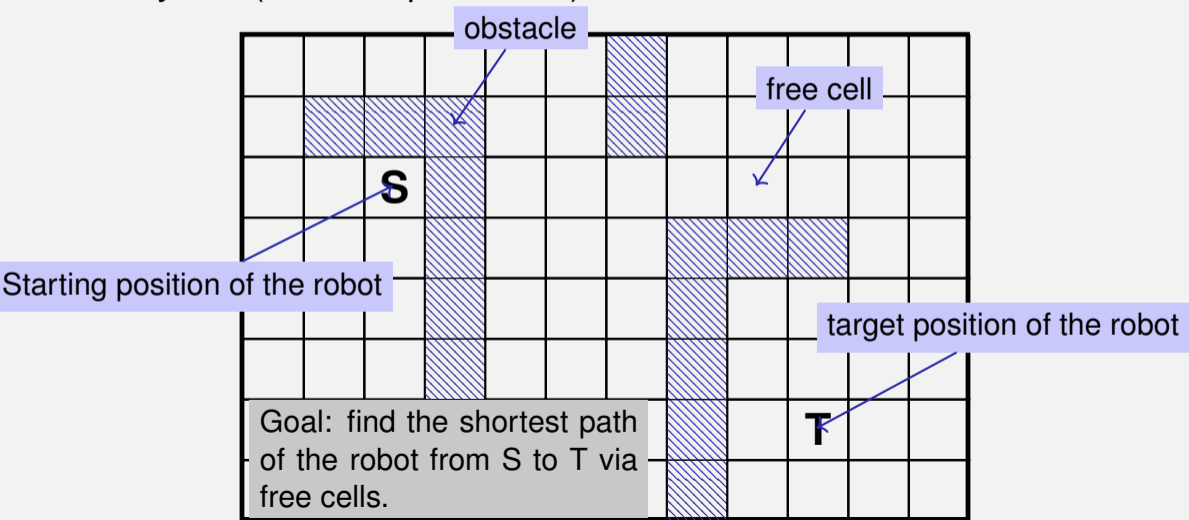
Factory hall ( $n \times m$  square cells)





# Application: Shortest Paths

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



## This problem appears to be different

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

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

# This problem appears to be different

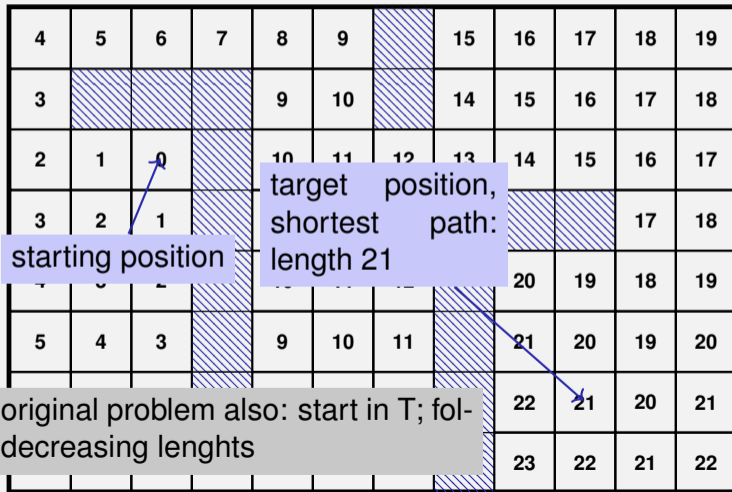
Find the *lengths* of the shortest paths to *all* possible targets.

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

This solves the original problem also: start in T; follow a path with decreasing lengths

# This problem appears to be different

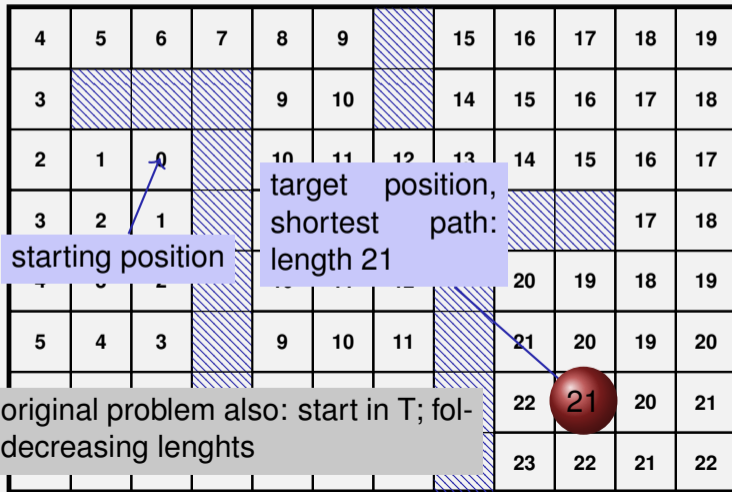
Find the *lengths* of the shortest paths to *all* possible targets.



This solves the original problem also: start in T; follow a path with decreasing lengths

# This problem appears to be different

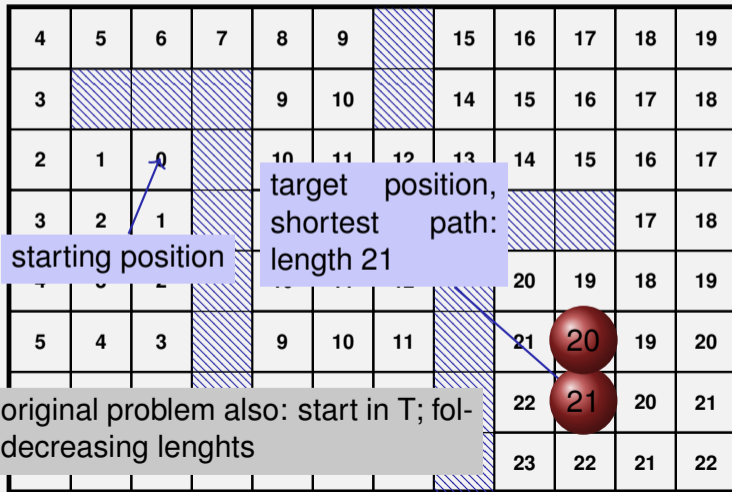
Find the *lengths* of the shortest paths to *all* possible targets.



This solves the original problem also: start in T; follow a path with decreasing lengths

# This problem appears to be different

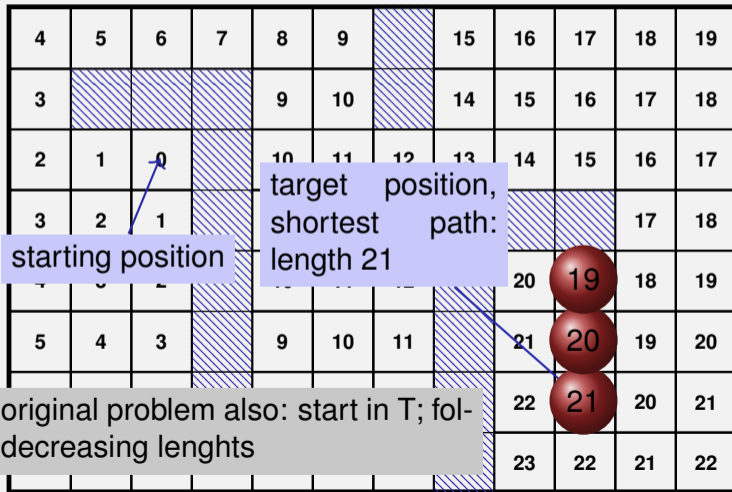
Find the *lengths* of the shortest paths to *all* possible targets.



This solves the original problem also: start in T; follow a path with decreasing lengths

# This problem appears to be different

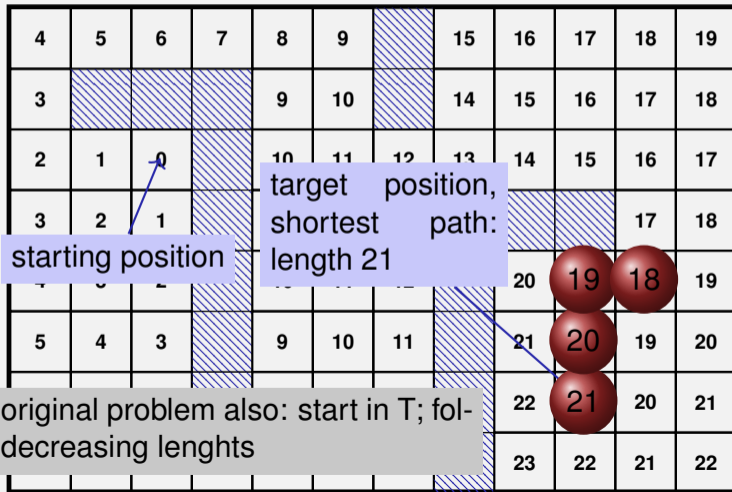
Find the *lengths* of the shortest paths to *all* possible targets.



This solves the original problem also: start in T; follow a path with decreasing lengths

# This problem appears to be different

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



starting position

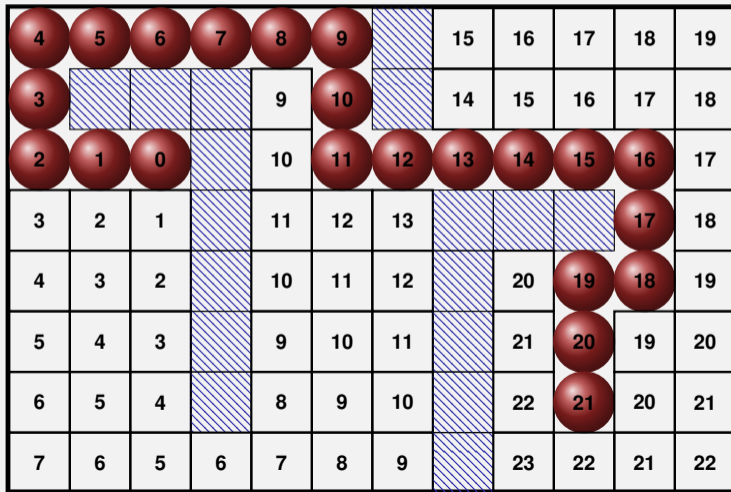
target position,  
shortest path:  
length 21

This solves the original problem also: start in T; follow a path with decreasing lengths

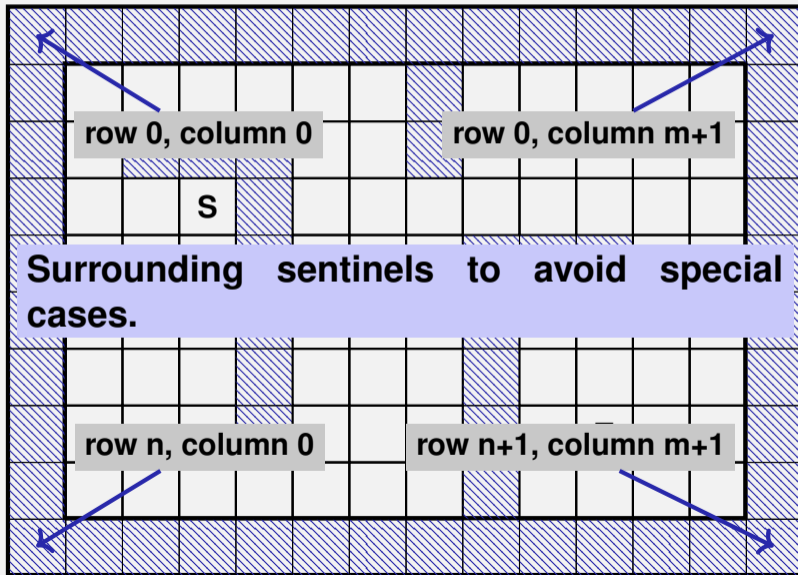


# This problem appears to be different

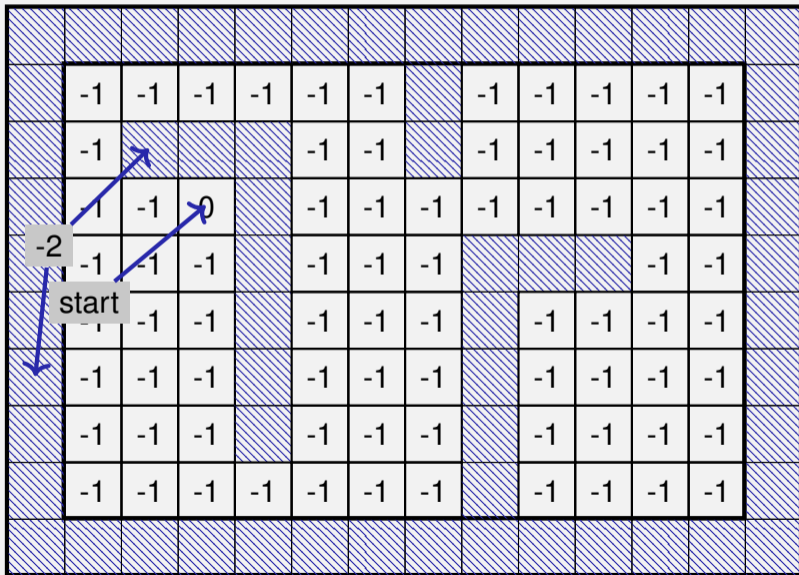
Find the *lengths* of the shortest paths to *all* possible targets.



# Preparation: Sentinels



# Preparation: Initial Marking



## The Shortest Path Program

```
// 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));

// Einlesen der Hallenbelegung, initiale Markierung
// (Handout)
...
// Markierung der umschliessenden Waende (Handout)
...
```

# The Shortest Path Program

```
// 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));
```

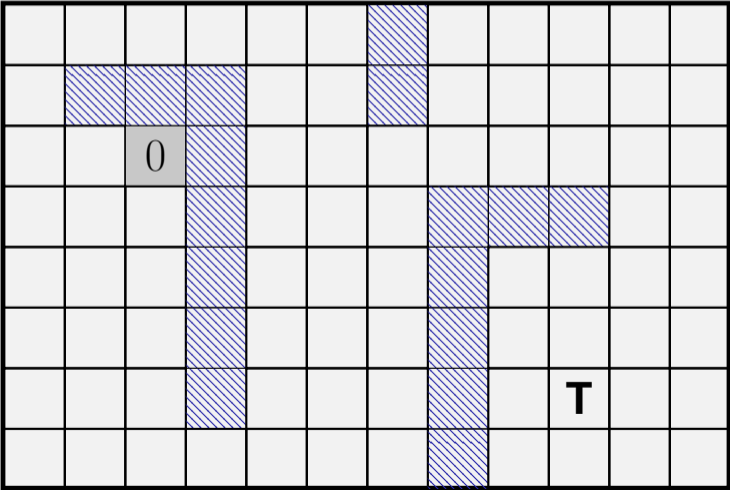
Sentinel



```
// Einlesen der Hallenbelegung, initiale Markierung
// (Handout)
...
// Markierung der umschliessenden Waende (Handout)
...
```

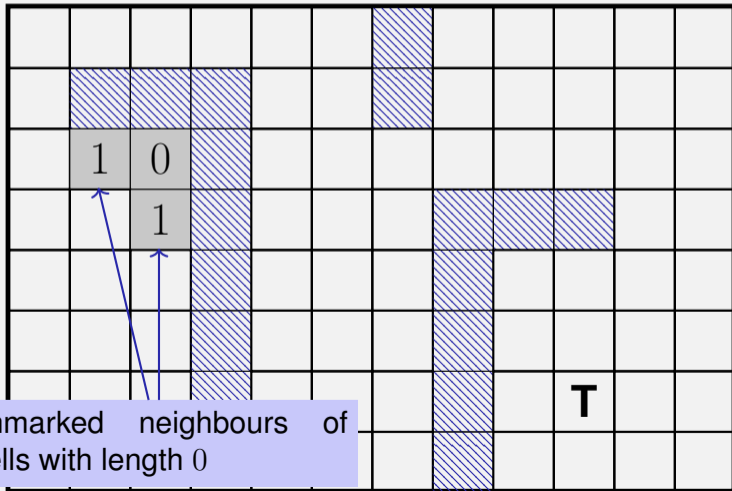
# Mark all Cells with their Path Lengths

Step 0: all cells with path length 0



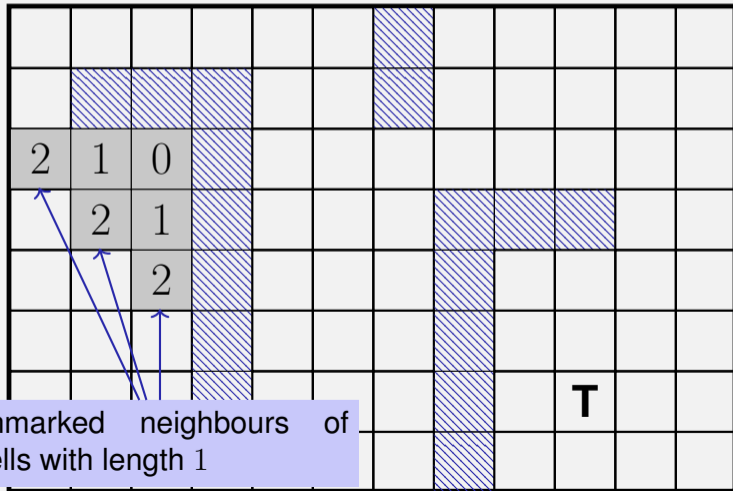
# Mark all Cells with their Path Lengths

Step 1: all cells with path length 1



# Mark all Cells with their Path Lengths

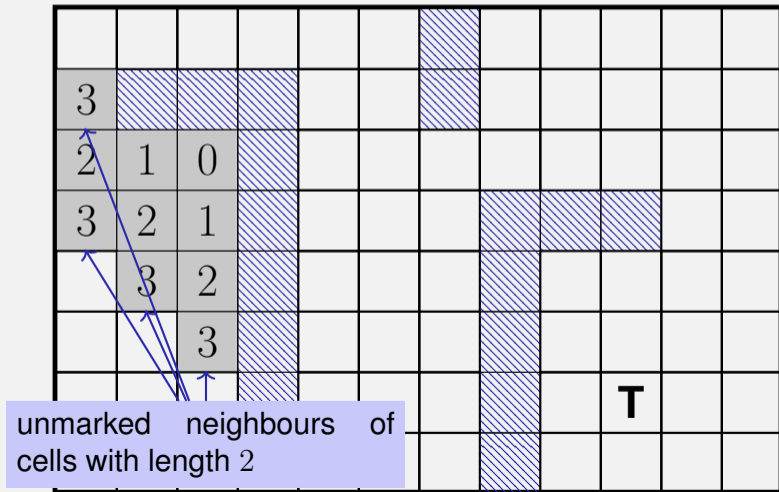
Step 2: all cells with path length 2





# Mark all Cells with their Path Lengths

Step 3: all cells with path length 3



# 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;
}
```

# Main Loop

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

```
for (int i=1;; ++i) {
 bool progress = false; ← indicates if in sweep through all cells
 there was progress
 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;
}
```

# Main Loop

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

```
for (int i=1;; ++i) {
 bool progress = false;
 for (int r=1; r<n+1; ++r) ← sweep over all cells
 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;
}
```

# 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;
}
```

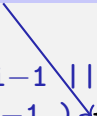
cell already marked or obstacle

# Main Loop

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

```
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;
}
```

a neighbour has path length  $i - 1$ . The sentinels guarantee that there are always 4 neighbours



# Main Loop

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

```
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; ← no progress, all reachable cells
}
```

no progress, all reachable cells  
marked; done.

# The Shortest Paths Program

- Algorithm: *Breadth First Search*



# The Shortest Paths Program

- Algorithm: *Breadth First Search*
- The program can become pretty slow because for each  $i$  all cells are traversed

# 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  $i - 1$ .
- Improvement: stop once the goal has been reached

# Vectors as Function Arguments

- Recall the following:

```
#include <iostream>
```

```
#include <vector>
```

```
// POST: Matrix 'm' was printed to std::cout
```

```
void print(std::vector<std::vector<int>> m);
```

```
int main() {
```

```
 std::vector<std::vector<int>> m = ...;
```

```
 print(m);
```

```
}
```

# Printing a Matrix: Version 1

- Recall the following:

```
// POST: Matrix 'm' was printed to std::cout
void print(std::vector<std::vector<int>> m);
...
print(m);
```

# Printing a Matrix: Version 1

- Recall the following:

```
// POST: Matrix 'm' was printed to std::cout
void print(std::vector<std::vector<int>> m);
...
print(m);
```

- Disadvantage: When calling `print(m)` the (potentially large) matrix `m` will be copied (*call-by-value*)  $\Rightarrow$  inefficient

# Printing a Matrix: Version 2

- Better: Pass by reference (*call-by-reference*)

```
// POST: Matrix 'm' was printed to std::cout
void print(std::vector<std::vector<int>>& m);
...
print(m);
```

# Printing a Matrix: Version 2

- Better: Pass by reference (*call-by-reference*)

```
// POST: Matrix 'm' was printed to std::cout
void print(std::vector<std::vector<int>>& m);
...
print(m);
```

- Disadvantage: `print(m)` could modify the matrix  $\Rightarrow$  potentially error-prone

# Printing a Matrix: Version 3

- Better: Pass by `const` reference

```
// POST: Matrix 'm' was printed to std::cout
void print(const std::vector<std::vector<int>>& m);
...
print(m);
```



# Printing a Matrix: Version 3

- Better: Pass by `const` reference

```
// POST: Matrix 'm' was printed to std::cout
void print(const std::vector<std::vector<int>>& m);
...
print(m);
```

- Now: Efficient, but nevertheless not more error-prone

# 14. Recursion 1

Mathematical Recursion, Termination, Call Stack, Examples,  
Recursion vs. Iteration, n-Queen Problem, Lindenmayer Systems

# Mathematical Recursion

- Many mathematical functions can be naturally defined **recursively**.

# 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 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. . .

# Infinite Recursion

- is as bad as an infinite loop...
- ...but even worse: it burns time **and** memory

# Infinite Recursion

- is as bad as an infinite loop...
- ...but even worse: it burns time **and** memory

```
void f()
{
 f(); // f() -> f() -> ... stack overflow
}
```



# Infinite Recursion

- is as bad as an infinite loop...
- ...but even worse: it burns time **and** memory

```
void f()
{
 f(); // f() -> f() -> ... stack overflow
}
```

*Ein Euro ist ein Euro.*

Wim Duisenberg, erster Präsident der EZB

# Recursive Functions: Termination

As with loops we need

- progress towards termination

# Recursive Functions: Termination

As with loops we need

- progress towards termination

`fac(n)` :

terminates immediately for  $n \leq 1$ , otherwise the function is called recursively with  $< n$  .

# Recursive Functions: Termination

As with loops we need

- progress towards termination

`fac(n)` :

terminates immediately for  $n \leq 1$ , otherwise the function is called recursively 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
}
```

Call of `fac(4)`

# Recursive Functions: Evaluation

Example: fac(4)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 4
 if (n <= 1) return 1;
 return n * fac(n-1); // n > 1
}
```

Initialization of the formal argument

# Recursive Functions: Evaluation

Example: fac(4)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 4
 if (n <= 1) return 1;
 return n * fac(n-1); // n > 1
}
```

Evaluation of the return expression

# Recursive Functions: Evaluation

Example: `fac(4)`

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 4
 if (n <= 1) return 1;
 return n * fac(n-1); // n > 1
}
```

recursive call with argument  $n - 1 == 3$



# Recursive Functions: Evaluation

Example: fac(4)

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 3
 if (n <= 1) return 1;
 return n * fac(n-1); // n > 1
}
```

Initialization of the formal argument

# Recursive Functions: Evaluation

Example: `fac(4)`

```
// POST: return value is n!
unsigned int fac (unsigned int n)
{ // n = 3
 if (n <= 1) return 1;
 return n * fac(n-1); // n > 1
}
```

Now there are two  $n$ . That of `fac(4)` and that of `fac(3)`

Initialization of the formal argument

# 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
}
```

The  $n$  of the current call is used:  $n = 3$

Initialization of the formal argument

# The Call Stack

```
std::cout << fac(4)
```

# The Call Stack

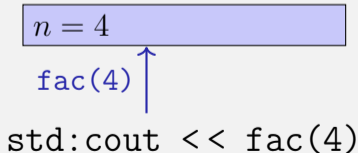
For each function call:

```
fac(4) ↑
std::cout << fac(4)
```

# The Call Stack

For each function call:

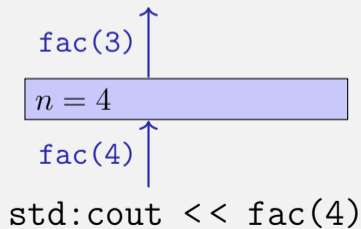
- push value of the call argument onto the stack



# The Call Stack

For each function call:

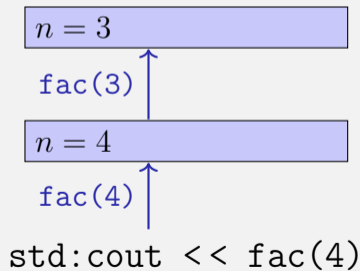
- push value of the call argument onto the stack



# The Call Stack

For each function call:

- push value of the call argument onto the stack

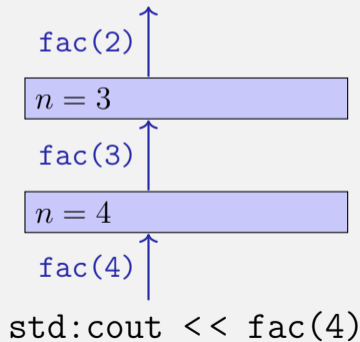




# The Call Stack

For each function call:

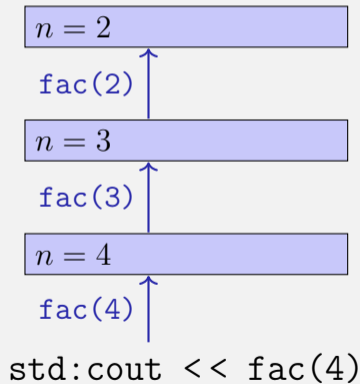
- push value of the call argument onto the stack



# The Call Stack

For each function call:

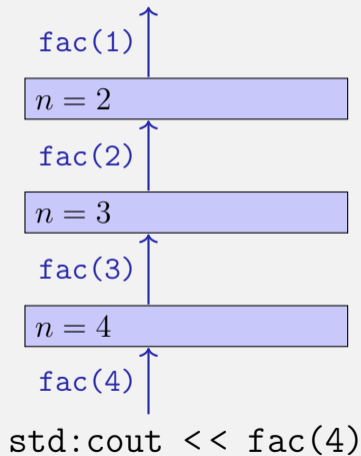
- push value of the call argument onto the stack



# The Call Stack

For each function call:

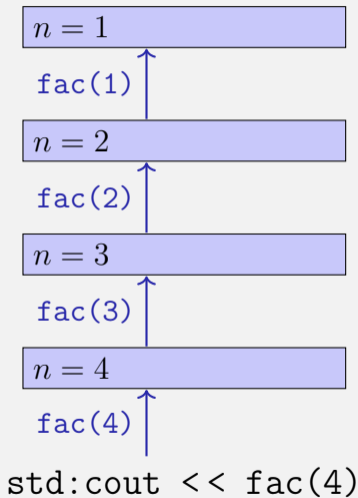
- push value of the call argument onto the stack



# The Call Stack

For each function call:

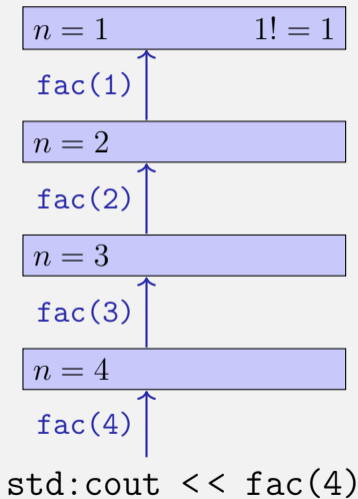
- push value of the call argument onto the stack



# The Call Stack

For each function call:

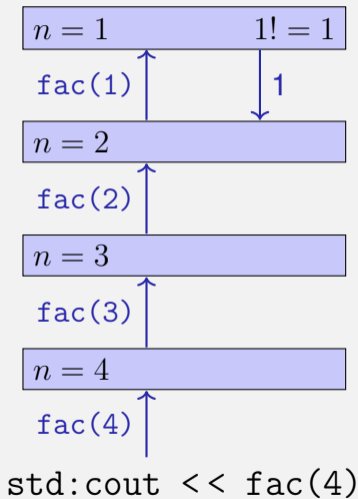
- push value of the call argument onto the stack
- always work with the top value



# The Call Stack

For each function call:

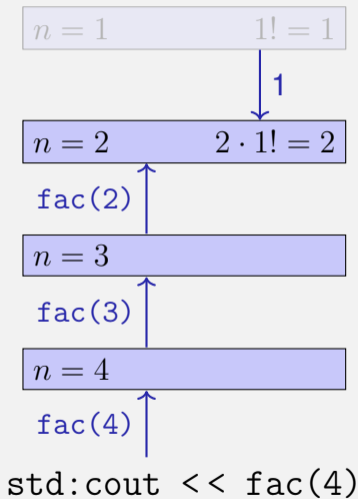
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



# The Call Stack

For each function call:

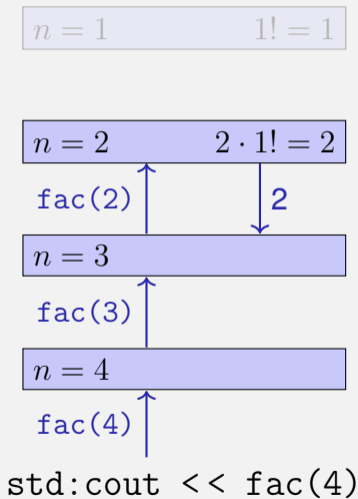
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



# 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

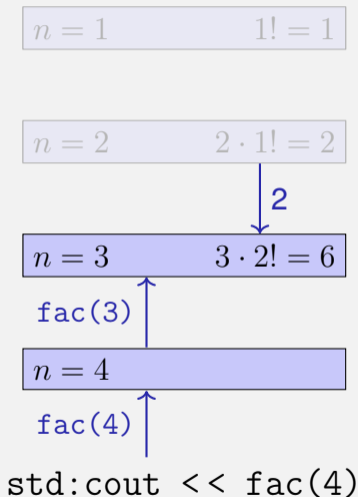




# The Call Stack

For each function call:

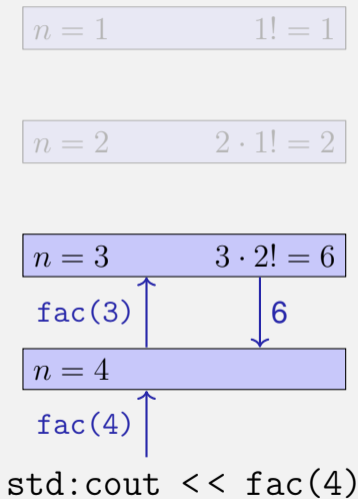
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



# The Call Stack

For each function call:

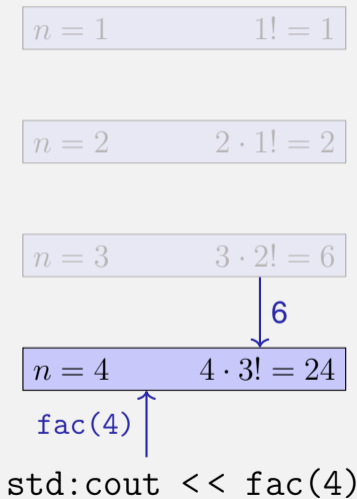
- push value of the call argument onto the stack
- always work with the top value
- at the end of the call the top value is removed from the stack



# 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



# 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

$$n = 1 \qquad 1! = 1$$

$$n = 2 \qquad 2 \cdot 1! = 2$$

$$n = 3 \qquad 3 \cdot 2! = 6$$

$$n = 4 \qquad 4 \cdot 3! = 24$$

`std::cout << fac(4)`

The diagram illustrates the state of the call stack during the execution of `std::cout << fac(4)`. The stack contains the current call with  $n = 4$  and the intermediate calculation  $4 \cdot 3! = 24$ . An upward arrow labeled `fac(4)` indicates the call argument being pushed onto the stack. A downward arrow labeled `24` indicates the return value being passed back to the caller.

# 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

$$n = 1 \qquad 1! = 1$$

$$n = 2 \qquad 2 \cdot 1! = 2$$

$$n = 3 \qquad 3 \cdot 2! = 6$$

$$n = 4 \qquad 4 \cdot 3! = 24$$

24

`std::cout << fac(4)`

# Euclidean Algorithm

- finds the greatest common divisor  $\gcd(a, b)$  of two natural numbers  $a$  and  $b$

# Euclidean Algorithm

- finds the greatest common divisor  $\text{gcd}(a, b)$  of two natural numbers  $a$  and  $b$
- is based on the following mathematical recursion (proof in the lecture notes):

$$\text{gcd}(a, b) = \begin{cases} a, & \text{if } b = 0 \\ \text{gcd}(b, a \bmod b), & \text{otherwise} \end{cases}$$

# Euclidean Algorithm in C++

$$\text{gcd}(a, b) = \begin{cases} a, & \text{if } b = 0 \\ \text{gcd}(b, a \bmod b), & \text{otherwise} \end{cases}$$

```
unsigned int gcd (unsigned int a, unsigned int b)
{
 if (b == 0)
 return a;
 else
 return gcd (b, a % b);
}
```



# Euclidean Algorithm in C++

$$\text{gcd}(a, b) = \begin{cases} a, & \text{if } b = 0 \\ \text{gcd}(b, a \bmod b), & \text{otherwise} \end{cases}$$

```
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

$$F_n := \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2}, & \text{if } n > 1 \end{cases}$$

# 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 . . .

# Fibonacci Numbers in Zurich



# Fibonacci Numbers in C++

$$F_n := \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2}, & \text{if } n > 1 \end{cases}$$

---

```
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
}
```

# Fibonacci Numbers in C++

$$F_n := \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F_{n-1} + F_{n-2}, & \text{if } n > 1 \end{cases}$$

---

```
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.

# 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
}
```

# Fast Fibonacci Numbers

Idea:

- Compute each Fibonacci number only once, in the order  $F_0, F_1, F_2, \dots, F_n!$



# Fast Fibonacci Numbers

Idea:

- Compute each Fibonacci number only once, in the order  $F_0, F_1, F_2, \dots, F_n!$
- Memorize the most recent two numbers (variables a and b)!

# Fast Fibonacci Numbers

Idea:

- Compute each Fibonacci number only once, in the order  $F_0, F_1, F_2, \dots, F_n!$
- 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 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;
}
```

$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$

a

b

# Fast Fibonacci Numbers in 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;
}
```

$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$

a

b

# Fast Fibonacci Numbers in 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;
}
```

$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$

a

b

# Fast Fibonacci Numbers in 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)

$(F_{i-2}, F_{i-1}) \longrightarrow (F_{i-1}, F_i)$

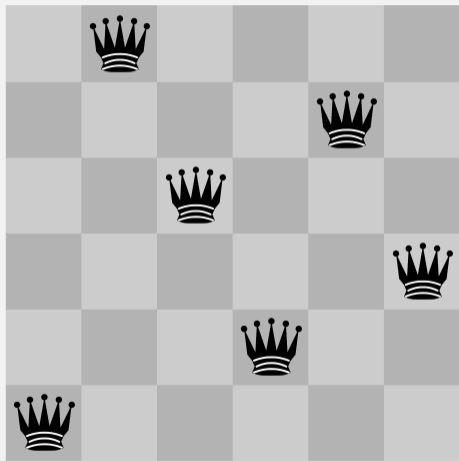
a

b

# The Power of Recursion

- Some problems appear to be hard to solve without recursion. With recursion they become significantly simpler.
- Examples: *The  $n$ -Queens-Problem*, The towers of Hanoi, *Sudoku-Solver*, Expression Parsers, Reversing In- or Output, Searching in Trees, Divide-And-Conquer (e.g. sorting)

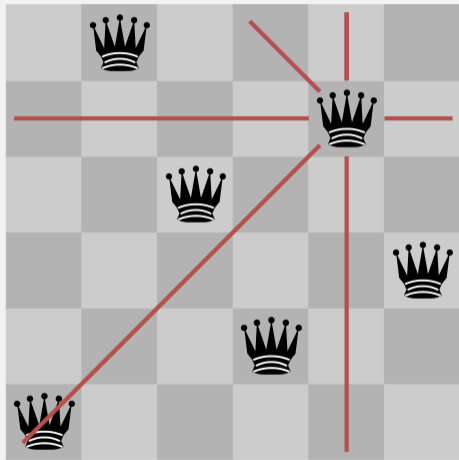
# The $n$ -Queens Problem



- Provided is a  $n$  times  $n$  chessboard
- For example  $n = 6$
- Question: is it possible to position  $n$  queens such that no two queens threaten each other?

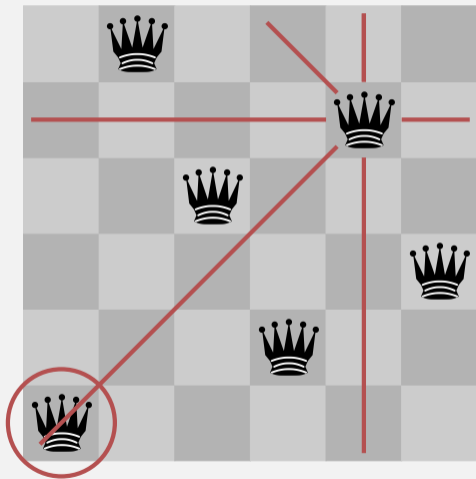


# The $n$ -Queens Problem



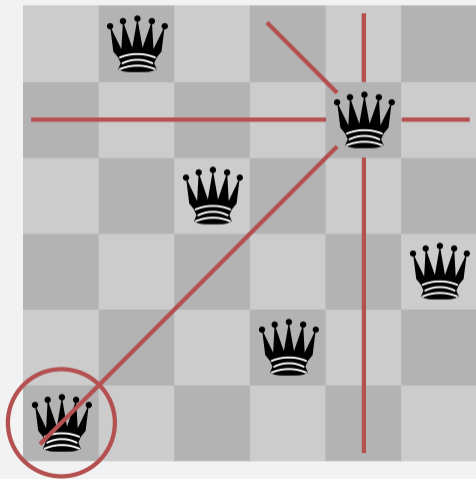
- Provided is a  $n \text{ times } n$  chessboard
- For example  $n = 6$
- Question: is it possible to position  $n$  queens such that no two queens threaten each other?

# The $n$ -Queens Problem



- Provided is a  $n$  times  $n$  chessboard
- For example  $n = 6$
- Question: is it possible to position  $n$  queens such that no two queens threaten each other?

# The $n$ -Queens Problem



- Provided is a  $n$  times  $n$  chessboard
- For example  $n = 6$
- Question: is it possible to position  $n$  queens such that no two queens threaten each other?
- If yes, how many solutions are there?

# Solution?

- Try all possible placements?

# Solution?

- Try all possible placements?
- $\binom{n^2}{n}$  possibilities. Too many!

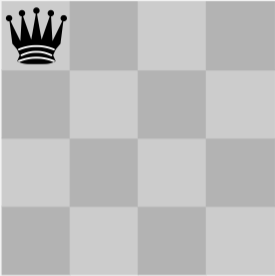
# Solution?

- Try all possible placements?
- $\binom{n^2}{n}$  possibilities. Too many!
- $n^n$  possibilities. Better – but still too many.

# Solution?

- Try all possible placements?
- $\binom{n^2}{n}$  possibilities. Too many!
- $n^n$  possibilities. Better – but still too many.
- Idea: Do not follow paths that obviously fail. (Backtracking)

# Solution with Backtracking



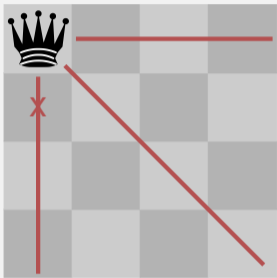
First Queen

queens

|   |
|---|
| 0 |
| 0 |
| 0 |
| 0 |



# Solution with Backtracking

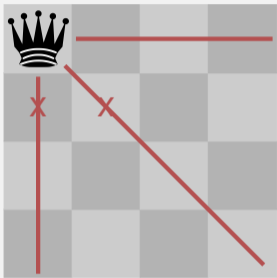


Forbidden  
Squares: no other  
queens may be  
here.

queens

|   |
|---|
| 0 |
| 0 |
| 0 |
| 0 |

# Solution with Backtracking

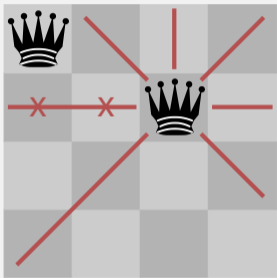


Forbidden  
Squares: no other  
queens may be  
here.

queens

|   |
|---|
| 0 |
| 1 |
| 0 |
| 0 |

# Solution with Backtracking

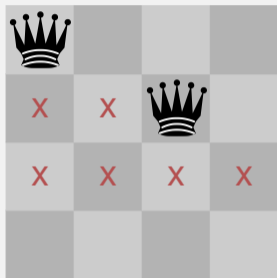


Second Queen in  
next row (no collision)

queens

|   |
|---|
| 0 |
| 2 |
| 0 |
| 0 |

# Solution with Backtracking

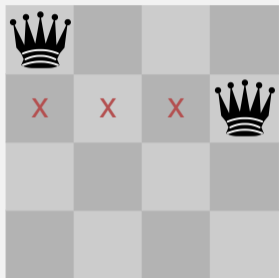


All squares in next row forbidden. Track back !

queens

|   |
|---|
| 0 |
| 2 |
| 4 |
| 0 |

# Solution with Backtracking

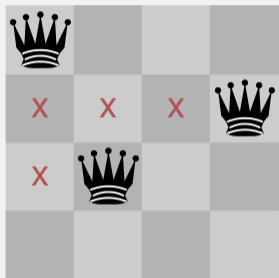


Move queen one step further and try again

queens

|   |
|---|
| 0 |
| 3 |
| 0 |
| 0 |

# Solution with Backtracking

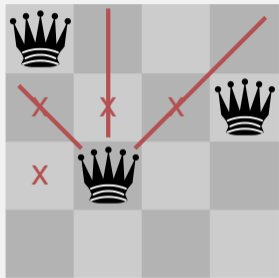


next row

queens

|   |
|---|
| 0 |
| 3 |
| 1 |
| 0 |

# Solution with Backtracking

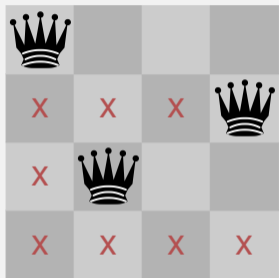


Ok (only previous queens have to be tested)

queens

|   |
|---|
| 0 |
| 3 |
| 1 |
| 0 |

# Solution with Backtracking



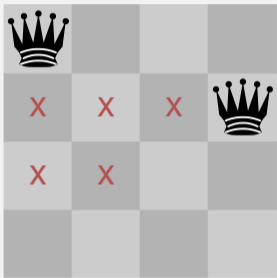
All squares of the next row forbidden.  
Track back.

queens

|   |
|---|
| 0 |
| 3 |
| 1 |
| 4 |



# Solution with Backtracking

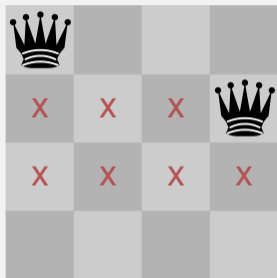


Continue in previous row.

queens

|   |
|---|
| 0 |
| 3 |
| 1 |
| 0 |

# Solution with Backtracking

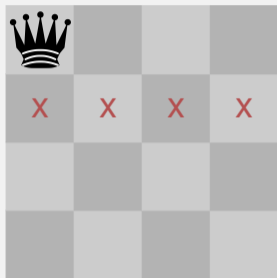


Remaining squares  
also forbidden.  
Track back!

queens

|   |
|---|
| 0 |
| 3 |
| 4 |
| 0 |

# Solution with Backtracking

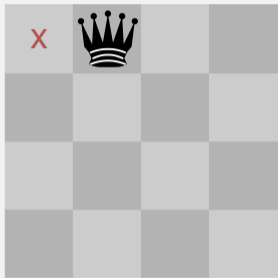


All squares of this row did not yield a solution. Track back!

queens

|   |
|---|
| 0 |
| 4 |
| 0 |
| 0 |

# Solution with Backtracking



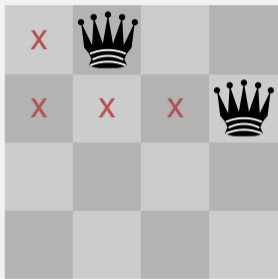
again  
queen  
square

advance  
by one

queens

|   |
|---|
| 1 |
| 0 |
| 0 |
| 0 |

# Solution with Backtracking

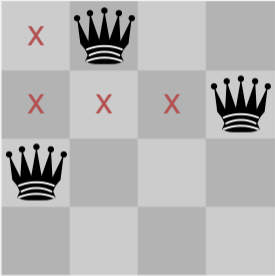


next row

queens

|   |
|---|
| 1 |
| 3 |
| 0 |
| 0 |

# Solution with Backtracking

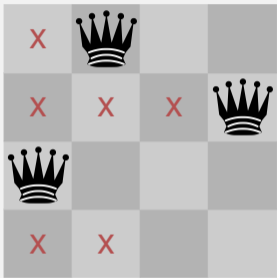


next row

queens

|   |
|---|
| 1 |
| 3 |
| 0 |
| 0 |

# Solution with Backtracking

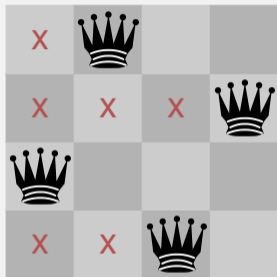


next row

queens

|   |
|---|
| 1 |
| 3 |
| 0 |
| 1 |

# Solution with Backtracking



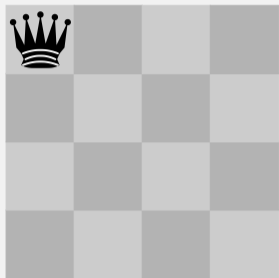
Found a solution

queens

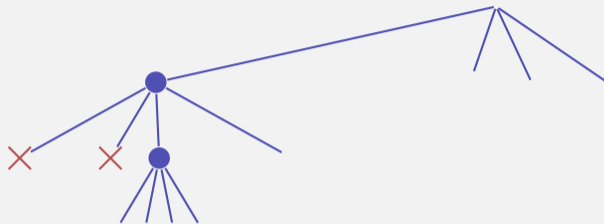
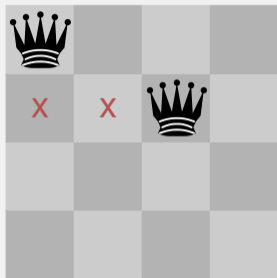
|   |
|---|
| 1 |
| 3 |
| 0 |
| 2 |



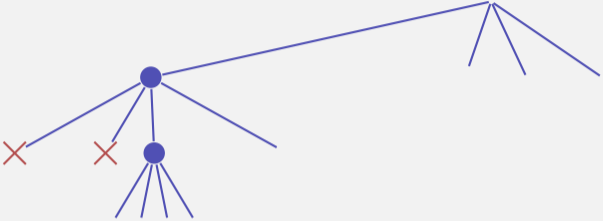
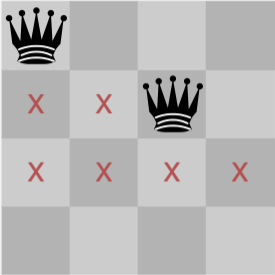
# Search Strategy Visualized as a Tree



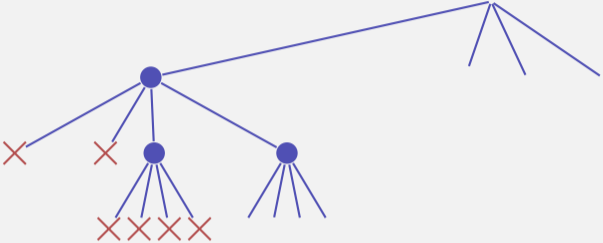
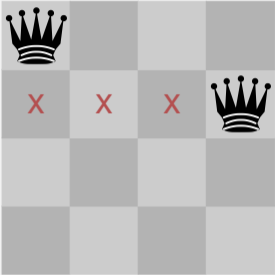
# Search Strategy Visualized as a Tree



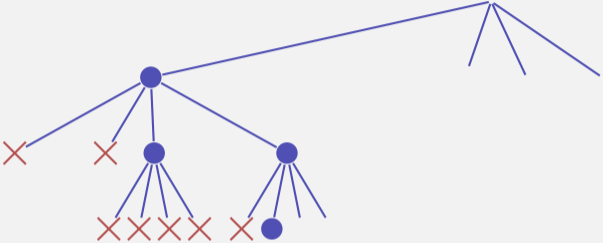
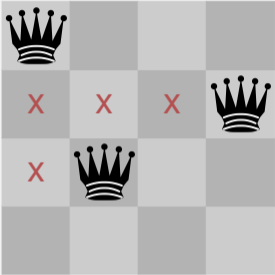
# Search Strategy Visualized as a Tree



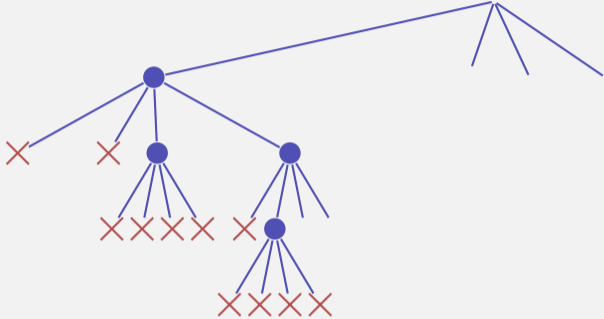
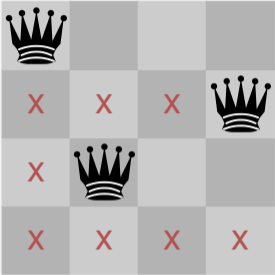
# Search Strategy Visualized as a Tree



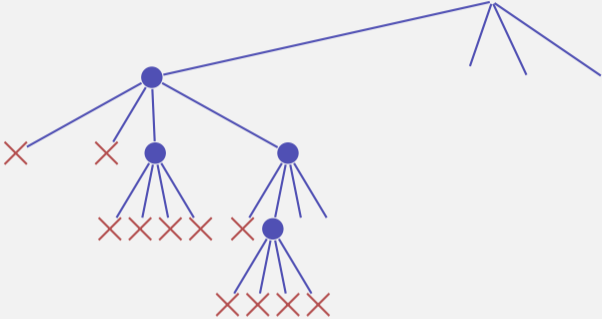
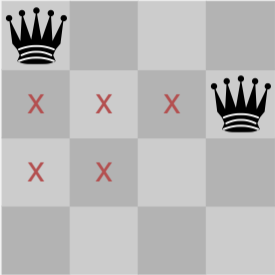
# Search Strategy Visualized as a Tree



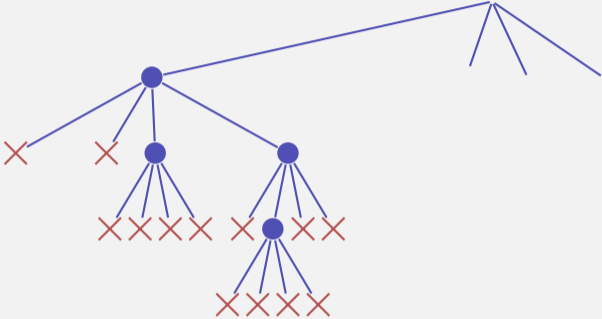
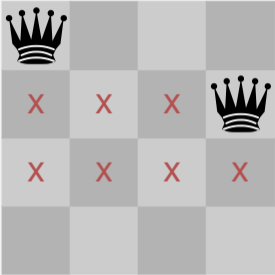
# Search Strategy Visualized as a Tree



# Search Strategy Visualized as a Tree

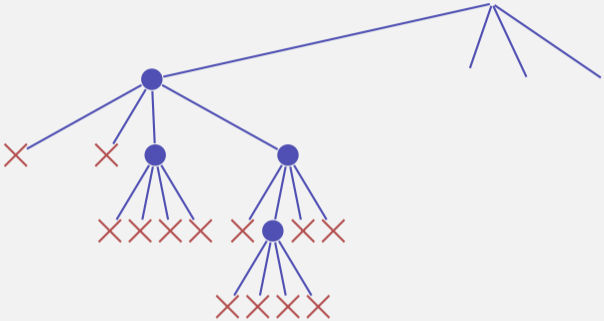
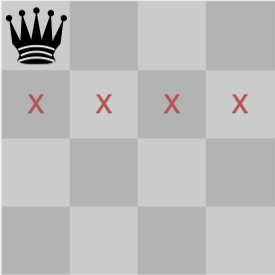


# Search Strategy Visualized as a Tree

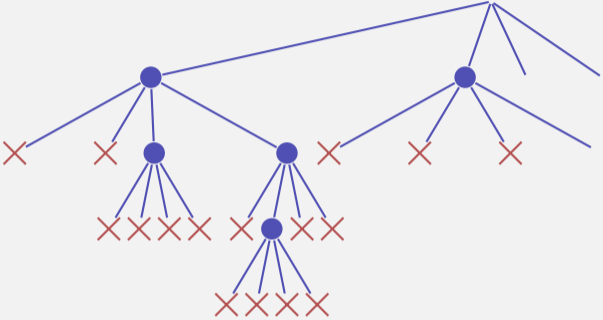
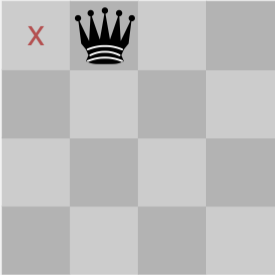




# Search Strategy Visualized as a Tree



# Search Strategy Visualized as a Tree











# Check Queen

```
using Queens = std::vector<unsigned int>;

// post: returns if queen in the given row is valid, i.e.
// does not share a common row, column or diagonal
// with any of the queens on rows 0 to row-1
bool valid(const Queens& queens, unsigned int row){
 unsigned int col = queens[row];
 for (unsigned int r = 0; r != row; ++r){
 unsigned int c = queens[r];
 if (col == c || col - row == c0 - r || col + row == c + r)
 return false; // same column or diagonal
 }
 return true; // no shared column or diagonal
}
```

# Recursion: Find a Solution

```
// pre: all queens from row 0 to row-1 are valid,
// i.e. do not share any common row, column or diagonal
// post: returns if there is a valid position for queens on
// row .. queens.size(). if true is returned then the
// queens vector contains a valid configuration.
bool solve(Queens& queens, unsigned int row){
 if (row == queens.size())
 return true;
 for (unsigned int col = 0; col != queens.size(); ++col){
 queens[row] = col;
 if (valid(queens, row) && solve(queens,row+1))
 return true; // (else check next position)
 }
 return false; // no valid configuration found
}
```



# Recursion: Count all Solutions

```
// pre: all queens from row 0 to row-1 are valid,
// i.e. do not share any common row, column or diagonal
// post: returns the number of valid configurations of the
// remaining queens on rows row ... queens.size()
int nSolutions(Queens& queens, unsigned int row){
 if (row == queens.size())
 return 1;
 int count = 0;
 for (unsigned int col = 0; col != queens.size(); ++col){
 queens[row] = col;
 if (valid(queens, row))
 count += nSolutions(queens,row+1);
 }
 return count;
}
```

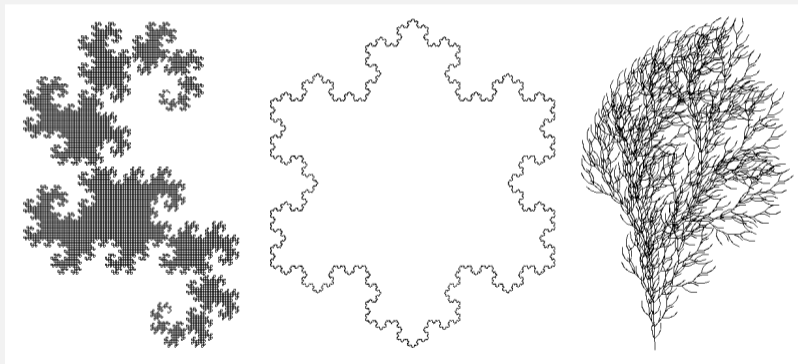
# Main Program

```
// pre: positions of the queens in vector queens
// post: output of the positions of the queens in a graphical way
void print(const Queens& queens);

int main(){
 int n;
 std::cin >> n;
 Queens queens(n);
 if (solve(queens,0)){
 print(queens);
 std::cout << "# solutions:" << nSolutions(queens,0) << std::endl;
 } else
 std::cout << "no solution" << std::endl;
 return 0;
}
```

# Lindenmayer-Systems (L-Systems)

Fractals from Strings and Turtles



# Definition and Example

- alphabet  $\Sigma$

- $\{F, +, -\}$

# Definition and Example

- alphabet  $\Sigma$

- $\Sigma^*$ : finite words over  $\Sigma$

- $\{F, +, -\}$

# Definition and Example

- alphabet  $\Sigma$
- $\Sigma^*$ : finite words over  $\Sigma$
- production  $P : \Sigma \rightarrow \Sigma^*$

- $\{F, +, -\}$

| $c$ | $P(c)$  |
|-----|---------|
| F   | F + F + |
| +   | +       |
| -   | -       |

# Definition and Example

- alphabet  $\Sigma$
- $\Sigma^*$ : finite words over  $\Sigma$
- production  $P : \Sigma \rightarrow \Sigma^*$
- initial word  $s_0 \in \Sigma^*$

- $\{F, +, -\}$

| $c$ | $P(c)$  |
|-----|---------|
| F   | F + F + |
| +   | +       |
| -   | -       |

- F

# Definition and Example

- alphabet  $\Sigma$
- $\Sigma^*$ : finite words over  $\Sigma$
- production  $P : \Sigma \rightarrow \Sigma^*$
- initial word  $s_0 \in \Sigma^*$

- $\{F, +, -\}$

| $c$ | $P(c)$  |
|-----|---------|
| F   | F + F + |
| +   | +       |
| -   | -       |

- F

## Definition

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



# The Language Described

Wörter  $w_0, w_1, w_2, \dots \in \Sigma^*$ :

$$P(F) = F + F +$$

$$w_0 := s_0$$

$$w_0 := F$$

# The Language Described

Wörter  $w_0, w_1, w_2, \dots \in \Sigma^*$ :

$$P(F) = F + F +$$

$$w_0 := s_0$$

$$w_0 := F$$

$$w_1 := P(w_0)$$

$$w_1 := F + F +$$

# The Language Described

Wörter  $w_0, w_1, w_2, \dots \in \Sigma^*$ :

$$P(F) = F + F +$$

$$w_0 := s_0$$

$$w_0 := F$$

$$w_1 := P(w_0)$$

$$w_1 := F + F +$$

$$w_2 := P(w_1)$$

$$w_2 := F + F + + F + F + +$$

## Definition

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

# The Language Described

Wörter  $w_0, w_1, w_2, \dots \in \Sigma^*$ :

$$P(F) = F + F +$$

$$w_0 := s_0$$

$$w_0 := F$$

$$w_1 := P(w_0)$$

$$w_1 := F + F +$$


$$w_2 := P(w_1)$$

$$w_2 := F + F + + F + F + +$$

$P(F) \quad P(+)$     $P(F) \quad P(+)$

## Definition

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

# The Language Described

Wörter  $w_0, w_1, w_2, \dots \in \Sigma^*$ :

$$P(F) = F + F +$$

$$w_0 := s_0$$

$$w_0 := F$$

$$w_1 := P(w_0)$$

$$w_1 := F + F +$$

$$w_2 := P(w_1)$$

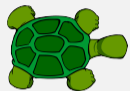
$$w_2 := F + F + + F + F + +$$

## Definition

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

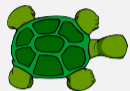
# Turtle Graphics

Turtle with position and direction



# Turtle Graphics

Turtle with position and direction



Turtle understands 3 commands:

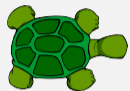
**F**: move one step forwards

**+**: rotate by 90 degrees

**-**: rotate by -90 degrees

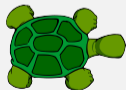
# Turtle Graphics

Turtle with position and direction

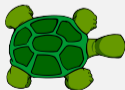


Turtle understands 3 commands:

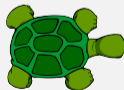
**F**: move one step forwards



**+**: rotate by 90 degrees



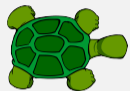
**-**: rotate by -90 degrees





# Turtle Graphics

Turtle with position and direction



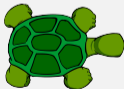
Turtle understands 3 commands:

**F**: move one step  
forwards ✓

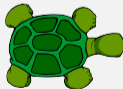
trace



**+**: rotate by 90  
degrees

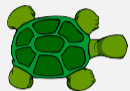


**-**: rotate by -90  
degrees



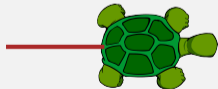
# Turtle Graphics

Turtle with position and direction

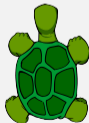


Turtle understands 3 commands:

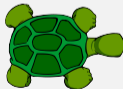
**F**: move one step forwards ✓



**+**: rotate by 90 degrees ✓

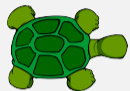


**-**: rotate by -90 degrees



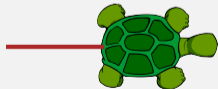
# Turtle Graphics

Turtle with position and direction

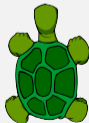


Turtle understands 3 commands:

**F**: move one step forwards ✓



**+**: rotate by 90 degrees ✓

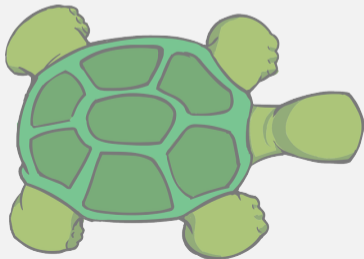


**-**: rotate by  $-90$  degrees ✓



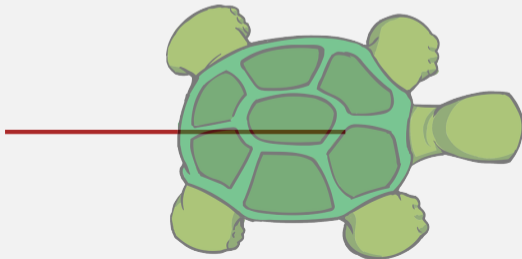
# Draw Words!

$$w_1 = \text{F} + \text{F} +$$



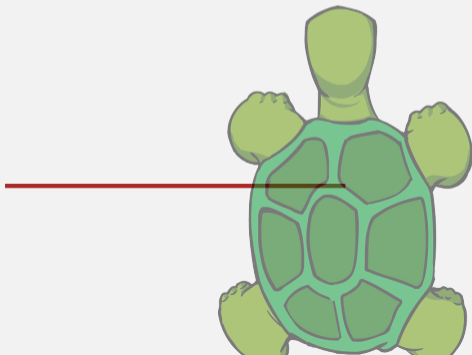
# Draw Words!

$$w_1 = \mathbf{F} + \mathbf{F} +$$

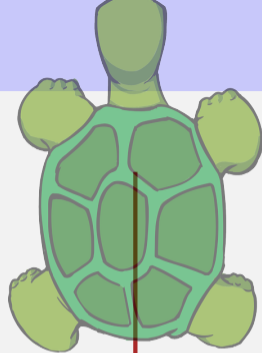


# Draw Words!

$$w_1 = \text{F} + \text{F} +$$

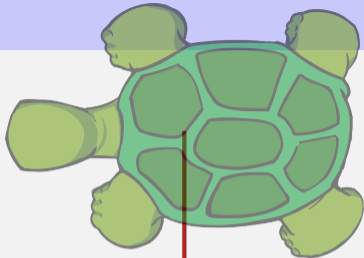


# Draw Words!



$$w_1 = \text{F} + \text{F} +$$

# Draw Words!

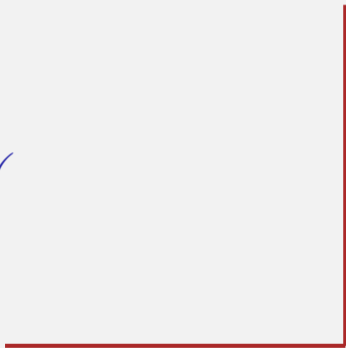


$$w_1 = \text{F} + \text{F} +$$



# Draw Words!

$$w_1 = \text{F} + \text{F} + \checkmark$$



word  $w_0 \in \Sigma^*$ :

```
int main () {
 std::cout << "Maximal Recursion Depth =? ";
 unsigned int n;
 std::cin >> n;

 std::string w = "F"; // w_0
 produce(w,n);

 return 0;
}
```

word  $w_0 \in \Sigma^*$ :

```
int main () {
 std::cout << "Maximal Recursion Depth =? ";
 unsigned int n;
 std::cin >> n;

 std::string w = "F"; // w_0
 produce(w,n);

 return 0;
}
```

$w = w_0 = F$

```
// POST: recursively iterate over the production of the characters
// of a word.
// When recursion limit is reached, the word is "drawn"
void produce(std::string word, int depth){
 if (depth > 0){
 for (unsigned int k = 0; k < word.length(); ++k)
 produce(produce(word[k]), depth-1);
 } else {
 draw_word(word);
 }
}
```

```
// POST: recursively iterate over the production of the characters
// of a word.
// When recursion limit is reached, the word is "drawn"
void produce(std::string word, int depth){
 if (depth > 0){ $w = w_i \rightarrow w = w_{i+1}$
 for (unsigned int k = 0; k < word.length(); ++k)
 produce(produce(word[k]), depth-1);
 } else {
 draw_word(word);
 }
}
```

```
// POST: recursively iterate over the production of the characters
// of a word.
// When recursion limit is reached, the word is "drawn"
void produce(std::string word, int depth){
 if (depth > 0){
 for (unsigned int k = 0; k < word.length(); ++k)
 produce(produce(word[k]), depth-1);
 } else {
 draw_word(word);
 }
}
```

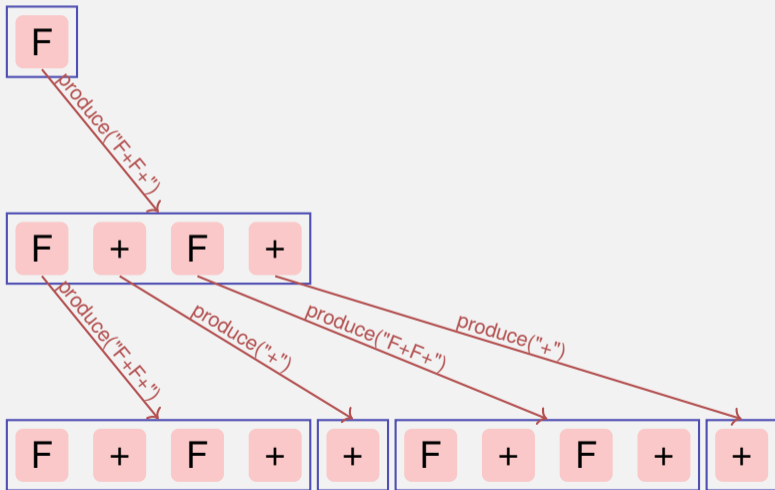
```
// POST: recursively iterate over the production of the characters
// of a word.
// When recursion limit is reached, the word is "drawn"
void produce(std::string word, int depth){
 if (depth > 0){
 for (unsigned int k = 0; k < word.length(); ++k)
 produce(produce(word[k]), depth-1);
 } else {
 draw $w = w_n!$
 draw_word(word);
 }
}
```

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



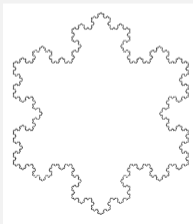
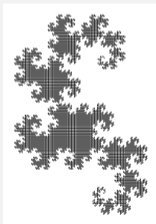
```
// POST: draws the turtle graphic interpretation of word
void draw_word (const std::string& word)
{
 for (unsigned int k = 0; k < word.length(); ++k)
 switch (word[k]) {
 case 'F':
 turtle::forward(); // move one step forward
 break;
 case '+':
 turtle::left(90); // turn counterclockwise by 90 degrees
 break;
 case '-':
 turtle::right(90); // turn clockwise by 90 degrees
 }
}
```

# The Recursion



# L-Systeme: Erweiterungen

- arbitrary symbols without graphical interpretation
- arbitrary angles (snowflake)
- saving and restoring the state of the turtle → plants (bush)



# 15. Recursion 2

Building a Calculator, Formal Grammars, Extended Backus Naur Form (EBNF), Parsing Expressions

# Motivation: Calculator

## Example

Input:  $3 + 5$

Output: 8

- binary Operators  $+$ ,  $-$ ,  $*$ ,  $/$  and numbers

# Motivation: Calculator

## Example

Input: 3 / 5

Output: 0.6

- binary Operators +, -, \*, / and numbers
- floating point arithmetic

# Motivation: Calculator

## Example

Input:  $3 + 5 * 20$

Output: 103

- binary Operators  $+$ ,  $-$ ,  $*$ ,  $/$  and numbers
- floating point arithmetic
- precedences and associativities like in C++

# Motivation: Calculator

## Example

Input:  $(3 + 5) * 20$

Output: 160

- binary Operators  $+$ ,  $-$ ,  $*$ ,  $/$  and numbers
- floating point arithmetic
- precedences and associativities like in C++
- parentheses



# Motivation: Calculator

## Example

Input:  $-(3 + 5) + 20$

Output: 12

- binary Operators  $+$ ,  $-$ ,  $*$ ,  $/$  and numbers
- floating point arithmetic
- precedences and associativities like in 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";
```

# Seems to work...

```
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 1 * 2 * 3 * 4 =
Result 24
```

# Oops, Multiplication first...

```
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:

$$13 + \dots$$

# Analyzing the Problem

## Example

Input:

$$13 + 4 * \dots$$

# Analyzing the Problem

## Example

Input:

$$13 + 4 * (15 - ...$$

# Analyzing the Problem

## Example

Input:

$$13 + 4 * (15 - 7 * ...$$



# Analyzing the Problem

## Example

Input:

$$13 + 4 * (15 - 7 * 3) =$$

Needs to be stored such that  
evaluation can be performed

# Analyzing the Problem

## Example

Result:

$$13 + 4*(15 - 21)$$

# Analyzing the Problem

## Example

Result:

$$13 + 4 * (-6)$$

# Analyzing the Problem

## Example

Result:

$$13 + (-24)$$

# Analyzing the Problem

Example

Result:

-11

# Analyzing the Problem

## Example

Expression:

$$13 + 4 * (15 - 7 * 3)$$

## Example

This

# Analyzing the Problem

## Example

Expression:

$$13 + 4 * (15 - 7 * 3)$$

## Example

This lecture

# Analyzing the Problem

## Example

Expression:

$$13 + 4 * (15 - 7 * 3)$$

## Example

This lecture is



# Analyzing the Problem

## Example

Expression:

$$13 + 4 * (15 - 7 * 3)$$

## Example

This lecture is pretty

# Analyzing the Problem

## Example

Expression:

$$13 + 4 * (15 - 7 * 3)$$

## Example

This lecture is pretty much

# Analyzing the Problem

## Example

Expression:

$$13 + 4 * (15 - 7 * 3)$$

## Example

This lecture is pretty much recursive.

# Analyzing the Problem

$$13 + 4 * (15 - 7 * 3)$$

“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 C++).

# Analyzing the Problem

$$13 + 4 * (15 - 7 * 3)$$

“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 C++).

# Analyzing the Problem

$$13 + 4 * (15 - 7 * 3)$$

“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 C++).

# Analyzing the Problem

$$13 + 4 * (15 - 7 * 3)$$

“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 C++).

# Formal Grammars

- Alphabet: finite set of symbols
- Strings: finite sequences of symbols



# Formal Grammars

- Alphabet: finite set of symbols
- Strings: finite sequences of symbols

A formal grammar defines which strings are valid.

# Formal Grammars

- Alphabet: finite set of symbols
- Strings: finite sequences of symbols

A formal grammar defines which strings are valid.

To describe the formal grammar, we use:

*Extended Backus Naur Form (EBNF)*

## What Can We Do about the Unnecessary Diversity of Notation for Syntactic Definitions?

Niklaus Wirth  
Federal Institute of Technology (ETH), Zürich, and  
Xerox Palo Alto Research Center

**Key Words and Phrases:** syntactic description  
language, extended BNF  
**CR Categories:** 4.20

The population of programming languages is steadily growing, and there is no end of this growth in sight. Many language definitions appear in journals, many are found in technical reports, and perhaps an even greater number remains confined to proprietary circles. After frequent exposure to these definitions, one cannot fail to notice the lack of "common denominators." The only widely accepted fact is that the language structure is defined by a syntax. But even notation for syntactic description eludes any commonly agreed standard form, although the underlying ancestor is invariably the Backus-Naur Form of the Algol 60 report. As variations are often only slight, they become annoying for their very lack of an apparent motivation.

Out of sympathy with the troubled reader who is weary of adapting to a new variant of BNF each time another language definition appears, and without any claim for originality, I venture to submit a simple notation that has proven valuable and satisfactory in use. It has the following properties to recommend it:

Copyright © 1977, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Author's present address: Xerox Corporation, Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

1. The notation distinguishes clearly between meta-, terminal, and nonterminal symbols.
2. It does not exclude characters used as metasympols from use as symbols of the language (as e.g. "|" in BNF).
3. It contains an explicit iteration construct, and thereby avoids the heavy use of recursion for expressing simple repetition.
4. It avoids the use of an explicit symbol for the empty string (such as (empty) or  $\epsilon$ ).
5. It is based on the ASCII character set.

This meta language can therefore conveniently be used to define its own syntax, which may serve here as an example of its use. The word *identifier* is used to denote *nonterminal symbol*, and *literal* stands for *terminal symbol*. For brevity, *identifier* and *character* are not defined in further detail.

```
syntax = {production}.
production = identifier "=" expression " ".
expression = term {"|" term}.
term = factor {factor}.
factor = identifier | literal | "(" expression ")" |
 "[" expression "]" | "{" expression "}".
literal = " " " " character {character} " " " " .
```

Repetition is denoted by curly brackets, i.e. {a} stands for  $\epsilon$  | a | aa | aaa | . . . . Optionality is expressed by square brackets, i.e. [a] stands for  $\epsilon$  | a. Parentheses merely serve for grouping, e.g. (a|b|c) stands for ac | bc. Terminal symbols, i.e. literals, are enclosed in quote marks (and, if a quote mark appears as a literal itself, it is written twice), which is consistent with common practice in programming languages.

# Expressions

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

What do we need in a grammar?

# Expressions

$$- (\underline{3} - (\underline{4} - \underline{5})) * (\underline{3} + \underline{4} * \underline{5}) / \underline{6}$$

What do we need in a grammar?

- Number

# Expressions

$$\underline{-} \left( \underline{3} - \left( \underline{4} - \underline{5} \right) \right) * \left( \underline{3} + \underline{4} * \underline{5} \right) / \underline{6}$$

What do we need in a grammar?

- Number , ( ? )

# Expressions

$$\underline{-} (3 - (4 - 5)) * (3 + 4 * 5) / 6$$

What do we need in a grammar?

- Number , ( ? )  
-Number, -( ? )

# Expressions

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

What do we need in a grammar?

- Number, ( ? )  
-Number, -( ? )
- ? \* ?, ? / ?, ...



# Expressions

$$-(3_{\underline{\quad}} - (4_{\underline{\quad}} - 5)) * (3_{\underline{\quad}} + 4 * 5) / 6$$

What do we need in a grammar?

- Number , ( ? )  
-Number, -( ? )
- ? \* ?, ? / ?, ...
- ? - ?, ? + ?, ...

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

What do we need in a grammar?

- Number , ( ? )  
-Number, -( ? )
- ? \* ?, ? / ?, ...
- ? - ?, ? + ?, ...

Factor

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

What do we need in a grammar?

- Number , ( ? )  
-Number, -( ? )
- Factor \* Factor,  
Factor / Factor , ...
- ? - ?, ? + ?, ...

Factor

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

What do we need in a grammar?

- Number , ( ? )  
-Number, -( ? )
- Factor \* Factor,  
Factor / Factor , ...
- ? - ?, ? + ?, ...

Factor

Term

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

What do we need in a grammar?

- Number , ( ? )  
-Number, -( ? )
- Factor \* Factor, Factor  
Factor / Factor , ...
- ? - ?, ? + ?, ...

Factor

Term

# Expressions

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

What do we need in a grammar?

- Number , ( ? )  
-Number, -( ? )
- Factor \* Factor, Factor  
Factor / Factor , ...
- Term + Term,  
Term - Term, ...

Factor

Term

# Expressions

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

What do we need in a grammar?

- Number , ( ? )  
-Number, -( ? )
- Factor \* Factor, Factor  
Factor / Factor , ...
- Term + Term,  
Term - Term, ...

Factor

Term

Expression

# Expressions

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

What do we need in a grammar?

- Number , ( ? )  
-Number, -( ? )
- Factor \* Factor, Factor  
Factor / Factor , ...
- Term + Term, **Term**  
Term - Term, ...

Factor

Term

Expression



# Expressions

$$-(3 - (4 - 5)) * (3 + 4 * 5) / 6$$

What do we need in a grammar?

- Number , ( Expression )  
-Number, -( Expression )
- Factor \* Factor, Factor  
Factor / Factor , ...
- Term + Term, Term  
Term - Term, ...

Factor

Term

Expression

# The EBNF for Expressions

A factor is

- a number,
- an expression in parentheses or
- a negated factor.

```
factor = number
 | "(" expression ")"
 | "-" factor.
```

# The EBNF for Expressions

A factor is

- a number,
- an expression in parentheses or
- a negated factor.

```
factor = number
 | "(" expression ")"
 | "-" factor.
```

# The EBNF for Expressions

A factor is

- a number,
- an expression in parentheses or
- a negated factor.

```
factor = number
 | "(" expression ")"
 | "-" factor.
```

# The EBNF for Expressions

A factor is

- a number,
- an expression in parentheses or
- a negated factor.

```
factor = number
 | "(" expression ")"
 | "-" factor.
```

# The EBNF for Expressions

A factor is

- a number,
- an expression in parentheses or
- a negated factor.

*non-terminal symbol*

factor = number  
| "(" expression ")"  
| "-" factor.

*terminal symbol*

*alternative*

# The EBNF for Expressions

A term is

- factor,
- factor \* factor, factor / factor,
- factor \* factor \* factor, factor / factor \* factor, ...
- ...

term = factor { "\*" factor | "/" factor }.

# The EBNF for Expressions

A term is

- factor,
- factor \* factor, factor / factor,
- factor \* factor \* factor, factor / factor \* factor, ...
- ...

term = factor { "\*" factor | "/" factor }.



# The EBNF for Expressions

A term is

- factor,
- factor \* factor, factor / factor,
- factor \* factor \* factor, factor / factor \* factor, ...
- ...

term = factor { "\*" factor | "/" factor }.

# The EBNF for Expressions

A term is

- factor,
- factor \* factor, factor / factor,
- factor \* factor \* factor, factor / factor \* factor, ...
- ...

term = factor { "\*" factor | "/" factor }.

# The EBNF for Expressions

A term is

- factor,
- factor \* factor, factor / factor,
- factor \* factor \* factor, factor / factor \* factor, ...
- ...

term = factor { "\*" factor | "/" factor } .

*optional repetition*

# The EBNF for Expressions

factor = number  
| "(" expression ")"  
| "-" factor.

term = factor { "\*" factor | "/" factor }.

expression = term { "+" term | "-" term }.

# Numbers

An *integer* comprises at least one digit, followed by an arbitrary number of digits.

```
number = digit { digit }.
```

```
digit = '0' | '1' | '2' | ... | '9'.
```

# Numbers

An integer *comprises at least one digit*, followed by an arbitrary number of digits.

```
number = digit { digit }.
digit = '0' | '1' | '2' | ... | '9'.
```

# Numbers

An integer comprises at least one digit, *followed by an arbitrary number of digits.*

number = digit { digit }.

digit = '0' | '1' | '2' | ... | '9'.

# Parsing

- **Parsing:** Check if a string is valid according to the EBNF.



# Parsing

- **Parsing:** Check if a string is valid according to the EBNF.
- **Parser:** A program for parsing.

# Parsing

- **Parsing:** Check if a string is valid according to the EBNF.
- **Parser:** A program for parsing.
- **Useful:** From the EBNF we can (nearly) automatically generate a parser

# Construct a Parser

- Rules become functions
- Alternatives and options become `if`-statements.
- Nonterminal symbols on the right hand side become function calls
- Optional repetitions become `while`-statements

# Rules (except number)

factor = number  
| "(" expression ")"  
| "-" factor.

term = factor { "\*" factor | "/" factor }.

expression = term { "+" term | "-" term }.

Expression is read from an input stream.

```
// POST: returns true if and only if is = factor ...
// and in this case extracts factor from is
bool factor (std::istream& is);
```

```
// POST: returns true if and only if is = term ...,
// and in this case extracts all factors from is
bool term (std::istream& is);
```

```
// POST: returns true if and only if is = expression ...,
// and in this case extracts all terms from is
bool expression (std::istream& is);
```

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 input, and the first non-whitespace character
// input returned (0 if there input no such character)
char lookahead (std::istream& input)
{
 input >> std::ws; // skip whitespaces
 if (input.eof())
 return 0; // end of stream
 else
 return input.peek(); // next character in input
}
```

# Cherry-Picking

... to extract the desired character.

```
// POST: if ch matches the next lookahead then consume it and return true
// otherwise return false
bool consume (std::istream& input, char c)
{
 if (lookahead (input) == c) {
 input >> c;
 return true;
 } else
 return false ;
}
```



# Evaluating Factors

```
double factor (std::istream& input)
{
 double value;
 if (consume (input, '(')) {
 value = expression (input); // "(" expression
 consume (input, ')'); // ")"
 } else if (consume (input, '-'))
 value = -factor (input); // - factor
 else
 value = number(input);
 return value;
}
```

```
factor = "(" expression ")"
 | "-" factor
 | number.
```

# Evaluating Terms

```
double term (std::istream& input)
{
 double value = factor (input); // factor
 while (true) {
 if (consume (input, '*')) // "*" factor
 value *= factor (input);
 else if (consume (input, '/')) // "/" factor
 value /= factor (input);
 else
 return value;
 }
}
```

term = factor { "\*" factor | "/" factor }.

# Evaluating Expressions

```
double expression (std::istream& input)
{
 double value = term (input); // term
 while (true) {
 if (consume (input, '+'))
 value += term (input); // "+" term
 else if (consume (input, '-'))
 value -= term (input); // "-" term
 else
 return value;
 }
}
```

expression = term { "+" term | "-" term }.

# Digits ...

```
// POST: returns the digit that could be consumed from a stream
// (0 if no digit available)
// digit = '0' | '1' | ... | '9'.
char digit(std::istream& input){
 char ch = input.peek(); // one symbol lookahead
 if (input.eof()) return 0; // nothing available on the stream
 if (ch >= '0' && ch <= '9'){
 input >> ch; // consume
 return ch;
 }
 return 0;
}
```

## ... and Numbers

```
// POST: returns an unsigned integer consumed from the stream
// number = digit {digit}.
unsigned int number (std::istream& input){
 input >> std::skipws;// skip whitespaces before the first digit
 char ch = digit(input);
 input >> std::noskipws; // no whitespaces allowed within a number
 unsigned int num = 0;
 while(ch > 0){ // skip remaining digits
 num = num * 10 + ch - '0';
 ch = digit(input);
 }
 return num;
}
```

# Recursion!

number

factor

term

expression

# Recursion!

number

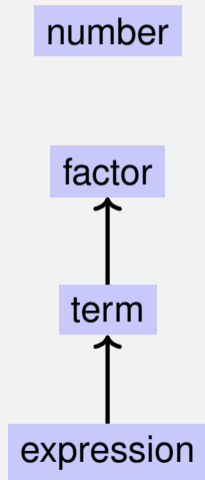
factor

term

expression

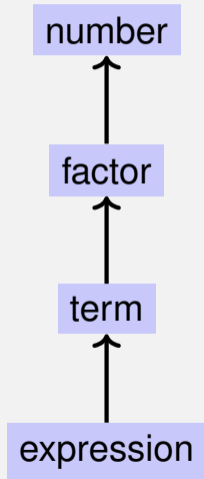
```
graph BT; expression --> term;
```

# Recursion!

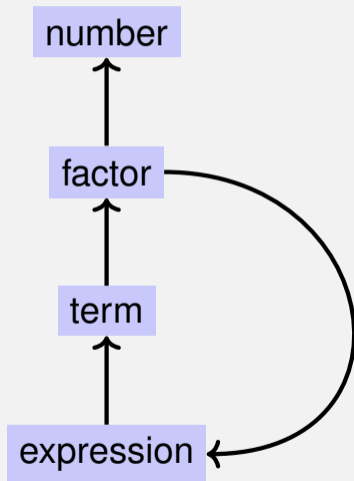




# Recursion!



# Recursion!



# EBNF — and it works!

EBNF (calculator.cpp, Evaluation from left to right):

```
factor = number
 | "(" expression ")"
 | "-" factor.
```

```
term = factor { "*" factor | "/" factor }.
```

```
expression = term { "+" term | "-" term }.
```

```
std::stringstream input ("1-2-3");
std::cout << expression (input) << "\n"; // -4
```

# 16. Structs

Rational Numbers, Struct Definition, Function- and Operator Overloading

# Calculating with Rational Numbers

- Rational numbers ( $\mathbb{Q}$ ) are of the form  $\frac{n}{d}$  with  $n$  and  $d$  in  $\mathbb{Z}$
- C++ does not provide a built-in type for rational numbers

# Calculating with Rational Numbers

- Rational numbers ( $\mathbb{Q}$ ) are of the form  $\frac{n}{d}$  with  $n$  and  $d$  in  $\mathbb{Z}$
- C++ does not provide a built-in type for rational numbers

## Goal

We build a C++-type for rational numbers ourselves! 😊

# Vision

```
// 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

```
struct rational {
 int n;
 int d; // INV: d != 0
};
```



# A First Struct

```
struct rational {
 int n; ← member variable (numerator)
 int d; // INV: d != 0
};
 ← member variable (denominator)
```

# A First Struct

```
struct rational {
 int n; ← member variable
 int d; // INV: d != 0
};
 ← member variable
```

- struct defines a new *type*

# A First Struct

```
struct rational {
 int n; ← member variable
 int d; // INV: d != 0
};
 ← member variable
```

- struct defines a new *type*
- formal range of values: *cartesian product* of the value ranges of existing types

# A First Struct

```
struct rational {
 int n; ← member variable
 int d; // INV: d != 0
};
 ← member variable
```

- 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;
}
```

$$\frac{r_n}{r_d} := \frac{a_n}{a_d} + \frac{b_n}{b_d} = \frac{a_n \cdot b_d + a_d \cdot b_n}{a_d \cdot b_d}$$

# 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: 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: Initialization and Assignment

```
rational s; ← member variables are uninitialized
```

```
rational t = {1,5};
```

```
rational u = t;
```

```
t = u;
```

```
rational v = add (u,t);
```

# Structs: Initialization and Assignment

```
rational s;
```

```
rational t = {1,5}; ← member-wise initialization:
t.n = 1, t.d = 5
```

```
rational u = t;
```

```
t = u;
```

```
rational v = add (u,t);
```

# Structs: Initialization and Assignment

```
rational s;
```

```
rational t = {1,5};
```

```
rational u = t; ← member-wise copy
```

```
t = u;
```

```
rational v = add (u,t);
```

# Structs: Initialization and Assignment

```
rational s;
```

```
rational t = {1,5};
```

```
rational u = t;
```

```
t = u; ← member-wise copy
```

```
rational v = add (u,t);
```

# Structs: Initialization and Assignment

```
rational s;
```

```
rational t = {1,5};
```

```
rational u = t;
```

```
t = u;
```

```
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?

# 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...



# 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,  $\frac{2}{3} \neq \frac{4}{6}$

# User Defined Operators

Instead of

```
rational t = add(r, s);
```

we would rather like to write

```
rational t = r + s;
```

# 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*.

# 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

```
std::cout << sq (3);
std::cout << sq (1.414);
std::cout << pow (2);
std::cout << pow (3,3);
```

# 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

```
std::cout << sq (3);
std::cout << sq (1.414);
std::cout << pow (2);
std::cout << pow (3,3);
```

# 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

```
std::cout << sq (3);
std::cout << sq (1.414);
std::cout << pow (2);
std::cout << pow (3,3);
```

# 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

```
std::cout << sq (3); // compiler chooses f2
std::cout << sq (1.414);
std::cout << pow (2);
std::cout << pow (3,3);
```

# 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

```
std::cout << sq (3); // compiler chooses f2
std::cout << sq (1.414); // compiler chooses f1
std::cout << pow (2);
std::cout << pow (3,3);
```



# 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

```
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);
```

# 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

```
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
```

# 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;
```

# 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

# 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 = operator+ (r, s);
```

↑  
equivalent but less handy: functional notation

# Unary Minus

Only one argument:

```
// POST: return value is $-a$
rational operator- (rational a)
{
 a.n = $-a.n$;
 return a;
}
```



# Comparison Operators

can be defined such that they do the right thing:

# Comparison Operators

can be defined such that they do the right thing:

```
// POST: returns true iff a == b
bool operator==(rational a, rational b)
{
 return a.n * b.d == a.d * b.n;
}
```

# Comparison Operators

can be defined such that they do the right thing:

```
// 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 +=

```
rational& operator+= (rational& a, rational b)
{
 a.n = a.n * b.d + a.d * b.n;
 a.d *= b.d;
 return a;
}
```

# Operator +=

```
rational& operator+= (rational& a, rational b)
{
 a.n = a.n * b.d + a.d * b.n;
 a.d *= b.d;
 return a;
}
```

- The L-value `a` is increased by the value of `b` and returned as L-value

# 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;
}
```



# 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";
```

# 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 <<

# 17. Classes

Encapsulation, Classes, Member Functions, Constructors

# 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



# Thought Experiment

The three core missions of ETH:

- research
- education

# Thought Experiment

The three core missions of ETH:

- research
- education
- technology transfer

# Thought Experiment

The three core missions of ETH:

- research
- education
- technology transfer

We found a startup: RAT PACK<sup>®</sup>!

# Thought Experiment

The three core missions of ETH:

- research
- education
- **technology transfer**

We found a startup: RAT PACK<sup>®</sup>!

- Selling the `rational` library to customers
- ongoing development according to customer's demands

## The Customer is Happy

*“Buying RAT PACK<sup>®</sup> has been a game-changing move to put us on the forefront of cutting-edge technology in social media engineering.”*

B. Labla, CEO

# The Customer is Happy

...and programs busily using `rational`.

# The Customer is Happy

...and programs busily using `rational`.

- output as double-value ( $\frac{3}{5} \rightarrow 0.6$ )

# 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?”

# 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;
};
```



```
struct rational {
 unsigned int n;
 unsigned int d;
 bool is_positive;
};
```

# New Version of RAT PACK®



*It sucks, nothing works any more!*



# New Version of RAT PACK®



*It sucks, nothing works any more!*

- What is the problem?



# New Version of RAT PACK<sup>®</sup>



*It sucks, nothing works any more!*

■ What is the problem?



*$-\frac{3}{5}$  is sometimes 0.6, this cannot be true!*



# New Version of RAT PACK<sup>®</sup>



*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.



# 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;
}
```



# Liability Discussion

```
// POST: double approximation of r
double to_double (rational r){
 double result = r.n;
 return result / r.d;
}
```

correct using...

```
struct rational {
 int n;
 int d;
};
```

# Liability Discussion

```
// POST: double approximation of r
double to_double (rational r){
 double result = r.n;
 return result / r.d;
}
```

correct using...

```
struct rational {
 int n;
 int d;
};
```

...not correct using

```
struct rational {
 unsigned int n;
 unsigned int d;
 bool is_positive;
};
```

# 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`)

# 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.

# 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.

# 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<sup>®</sup> is history...

# Idea of Encapsulation (Information Hiding)

- A type is uniquely defined by its *value range* and its *functionality*



# Idea of Encapsulation (Information Hiding)

- A type is uniquely defined by its *value range* and its *functionality*
- The *representation* should *not be visible*.

# Idea of Encapsulation (Information Hiding)

- A type is uniquely defined by its *value range* and its *functionality*
- The *representation* should *not be visible*.
- $\Rightarrow$  The customer is not provided with *representation* but with *functionality*!

# Idea of Encapsulation (Information Hiding)

- A type is uniquely defined by its *value range* and its *functionality*
- The *representation* should *not be visible*.
- $\Rightarrow$  The customer is not provided with *representation* but with *functionality!*



```
str.length(),
v.push_back(1),...
```

# Classes

- provide the concept for encapsulation in C++

# Classes

- provide the concept for encapsulation in C++
- are a variant of structs

# 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”

# 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



# Encapsulation: public / private

```
class rational {
 int n;
 int d; // INV: d != 0
};
```

## 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
```

# Encapsulation: public / private

```
class rational {
 int n;
 int d; // INV: d != 0
};
```

Good news: `r.d = 0` cannot happen any more by accident.

## 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
```

# 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 ...

## 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
```

# 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 ...

## 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
```

...and we can't, either.  
(no operator+,...)

# Member Functions: Declaration

```
class rational {
public:
 // POST: return value is the numerator of this instance
 int numerator () const {
 return n;
 }
 // POST: return value is the denominator of this instance
 int denominator () const {
 return d;
 }
private:
 int n;
 int d; // INV: d!= 0
};
```

# Member Functions: Declaration

```
class rational {
public:
 // POST: return value is the numerator of this instance
 int numerator () const {
 return n;
 }
 // POST: return value is the denominator of this instance
 int denominator () const {
 return d;
 }
private:
 int n;
 int d; // INV: d!= 0
};
```

public area

# Member Functions: Declaration

```
class rational {
public:
 // POST: return value is the numerator of this instance
 int numerator () const { member function
 return n;
 }
 // POST: return value is the denominator of this instance
 int denominator () const {
 return d;
 }
private:
 int n;
 int d; // INV: d!= 0
};
```

public area

# Member Functions: Declaration

```
class rational {
public:
 // POST: return value is the numerator of this instance
 int numerator () const {
 return n;
 }
 // POST: return value is the denominator of this instance
 int denominator () const {
 return d;
 }
private:
 int n;
 int d; // INV: d!= 0
};
```

public area

member function

member functions have access to private data



# Member Functions: Call

```
// Definition des Typs
class rational {
 ...
};
...
// Variable des Typs
rational r; member access

int n = r.numerator(); // Zaehler
int d = r.denominator(); // Nenner
```

# Member Functions: Definition

```
// POST: returns numerator of this instance
int numerator () const
{
 return n;
}
```

# Member Functions: Definition ???

```
// POST: returns numerator of this instance
int numerator () const
{
 return n;
}
```

# Member Functions: Definition

```
// POST: returns numerator of this instance
int numerator () const
{
 return n; r.numerator()
}
```

- A member function is called **for** an expression of the class.

# Member Functions: Definition

```
// POST: returns numerator of this instance
int numerator () const
{
 return n;
}
```

**r.numerator()**



- A member function is called **for** an expression of the class. in the function, **this** ~~is the name of this~~ **implicit argument**.

# Member Functions: Definition

```
// POST: returns numerator of this instance
int numerator () const
{
 return n; r.numerator()
}
```

- A member function is called **for** an expression of the class. in the function, **this** is the name of this **implicit argument**.
- **const** refers to the instance **this**

# Member Functions: Definition

```
// POST: returns numerator of this instance
int numerator () const
{
 return n;
}
```

`r.numerator()`

- A member function is called **for** an expression of the class. in the function, **this** is the name of this **implicit argument**.
- **const** refers to the instance **this**
- **n** is the shortcut for **this->n** (precise explanation of “->” next week)

# const and Member Functions

```
class rational {
public:
 int numerator () const
 { return n; }
 void set_numerator (int N)
 { n = N;}
 ...
}
```

```
rational x;
x.set_numerator(10); // ok;
const rational y = x;
int n = y.numerator(); // ok;
y.set_numerator(10); // error;
```

The `const` at a member function is to promise that an instance cannot be changed via this function.

`const` items can only call `const` member functions.



# Comparison

```
class rational {
 int n;
 ...
public:
 int numerator () const
 {
 return n;
 }
};

rational r;
...
std::cout << r.numerator();
```

# Comparison

```
class rational {
 int n;
 ...
public:
 int numerator () const
 {
 return this->n;
 }
};

rational r;
...
std::cout << r.numerator();
```

# Comparison

**Roughly** like this it were ...

```
class rational {
 int n;
 ...
public:
 int numerator () const
 {
 return this->n;
 }
};

rational r;
...
std::cout << r.numerator();
```

# Comparison

**Roughly** like this it were ...

```
class rational {
 int n;
 ...
public:
 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

```
class rational {
 int n;
 ...
public:
 int numerator () const
 {
 return n;
 }

};
```

- No separation between declaration and definition (bad for libraries)

# Member-Definition: In-Class vs. Out-of-Class

```
class rational {
 int n;
 ...
public:
 int numerator () const
 {
 return n;
 }

};
```

- No separation between declaration and definition (bad for libraries)

```
class rational {
 int n;
 ...
public:
 int numerator () const;
 ...
};

int rational::numerator () const
{
 return n;
}
```

- This also works.

# Initialisation? Constructors!

```
class rational
{
public:
 rational (int num, int den)
 : n (num), d (den)
 {
 assert (den != 0);
 }
 ...
};
...
rational r (2,3); // r = 2/3
```

# Initialisation? Constructors!

```
class rational
{
public:
 rational (int num, int den)
 : n (num), d (den) ← Initialization of the
 member variables
 {
 assert (den != 0); ← function body.
 }
 ...
};
...
rational r (2,3); // r = 2/3
```



# Initialisation “rational = int”?

```
class rational
{
public:
 rational (int num)
 : n (num), d (1)
 {}

 ...
};

...
rational r (2); // explicit initialization with 2
rational s = 2; // implicit conversion
```

# 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
```

# The Default Constructor

```
class rational
{
public:
 ...
 rational () ← empty list of arguments
 : n (0), d (1)
 {}
 ...
};
...
rational r; // r = 0
```

# The Default Constructor

```
class rational
{
public:
 ...
 rational () ← empty list of arguments
 : n (0), d (1)
 {}
 ...
};
...
rational r; // r = 0
```

⇒ There are no uninitialized variables of type rational any more!

# Alternatively: Deleting a Default Constructor

```
class rational
{
public:
 ...
 rational () = delete;
 ...
};
...
rational r; // error: use of deleted function 'rational::rational()'

```

⇒ There are no uninitialized variables of type rational any more!

# RAT PACK<sup>®</sup> 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();
}
```

# RAT PACK<sup>®</sup> 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<sup>®</sup> Reloaded ...

before

```
class rational {
 ...
private:
 int n;
 int d;
};
```

---



# RAT PACK<sup>®</sup> Reloaded ...

before

```
class rational {
 ...
private:
 int n;
 int d;
};
```

```
int numerator () const
{
 return n;
}
```

---

# RAT PACK<sup>®</sup> 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;
};
```

# RAT PACK<sup>®</sup> 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<sup>®</sup> 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;
 }
}
```

# RAT PACK<sup>®</sup> 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

# RAT PACK<sup>®</sup> 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 Incomplete

Customer's point of view (rational.h):

```
class rational {
public:
 // POST: returns numerator of *this
 int numerator () const;
 ...
private:
 // none of my business
};
```

# Encapsulation still Incomplete

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`



# Encapsulation still Incomplete

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:
 using integer = long int; // might change
 // POST: returns numerator of *this
 integer numerator () const;
```

# Fix: “our” type `rational::integer`

Customer’s point of view (`rational.h`):

```
public:
 using integer = long int; // might change
 // POST: returns numerator of *this
 integer numerator () const;
```

- We provide an additional type!

# Fix: “our” type `rational::integer`

Customer’s point of view (`rational.h`):

```
public:
 using integer = long int; // might change
 // POST: returns numerator of *this
 integer numerator () const;
```

- We provide an additional type!
- Determine only **Functionality**, e.g:
  - implicit conversion `int`  $\rightarrow$  `rational::integer`

# Fix: “our” type `rational::integer`

Customer’s point of view (`rational.h`):

```
public:
 using integer = long int; // might change
 // POST: returns numerator of *this
 integer numerator () const;
```

- We provide an additional type!
- Determine only **Functionality**, e.g:
  - implicit conversion `int`  $\rightarrow$  `rational::integer`
  - function `double to_double (rational::integer)`

# RAT PACK<sup>®</sup> 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);
}
```

# 18. Dynamic Data Structures I

Dynamic Memory, Addresses and Pointers, Const-Pointer Arrays,  
Array-based Vectors

## Recap: `vector<T>`

- Can be initialised with arbitrary size `n`



# Recap: `vector<T>`

- Can be initialised with arbitrary size `n`
- Supports various operations:

```
e = v[i]; // Get element
v[i] = e; // Set element
l = v.size(); // Get size
v.push_front(e); // Prepend element
v.push_back(e); // Append element
...
```

# Recap: `vector<T>`

- Can be initialised with arbitrary size `n`
- Supports various operations:

```
e = v[i]; // Get element
v[i] = e; // Set element
l = v.size(); // Get size
v.push_front(e); // Prepend element
v.push_back(e); // Append element
...
```

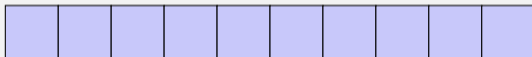
- A vector is a *dynamic data structure*, whose size may change at runtime

# Our Own Vector!

- Today, we'll implement our own vector: `vec`
- Step 1: `vec<int>` (today)
- Step 2: `vec<T>` (later, only superficially)

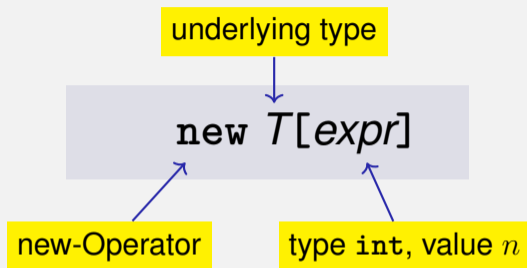
# Vectors in Memory

Already known: A vector has a *contiguous* memory layout



**Question:** How to *allocate* a chunk of memory of *arbitrary* size during runtime, i.e. *dynamically*?

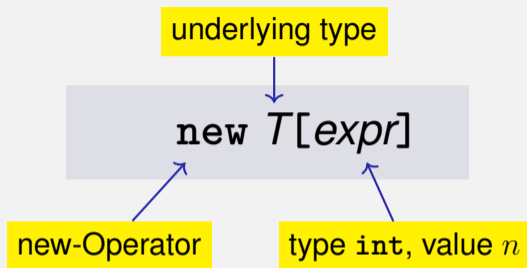
# new for Arrays



- **Effect:** new contiguous chunk of memory  $n$  elements of type  $T$  is allocated



# new for Arrays

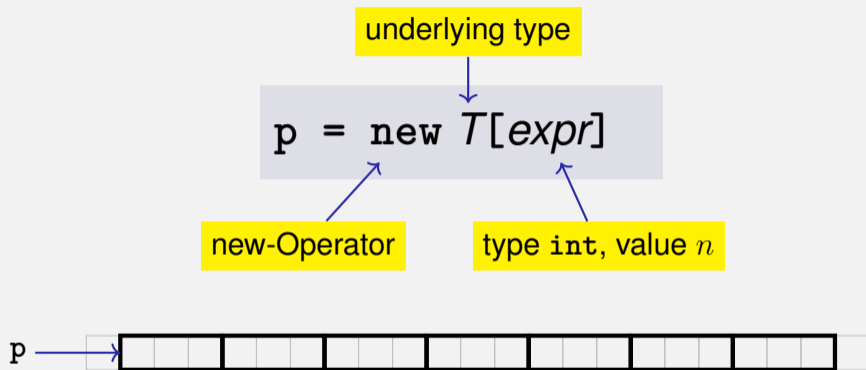


- **Effect:** new contiguous chunk of memory  $n$  elements of type  $T$  is allocated



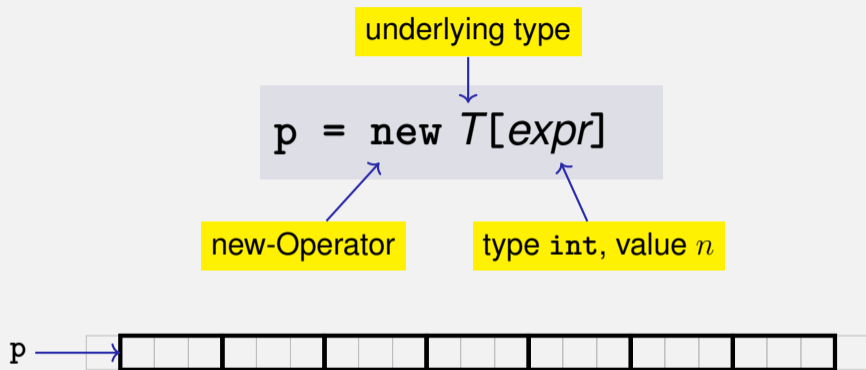
- This chunk of memory is called an *array* (of length  $n$ )

# new for Arrays



- **Type:** A pointer  $T^*$  (more soon)

# new for Arrays



- **Type:** A pointer  $T^*$  (more soon)
- **Value:** the starting address of the memory chunk



# Outlook: new and delete

```
new T[expr]
```

- So far: memory (local variables, function arguments) “lives” only inside a function call

# Outlook: new and delete

```
new T[expr]
```

- So far: memory (local variables, function arguments) “lives” only inside a function call
- But now: memory chunk inside vector must not “die” before the vector itself

# Outlook: new and delete

```
new T[expr]
```

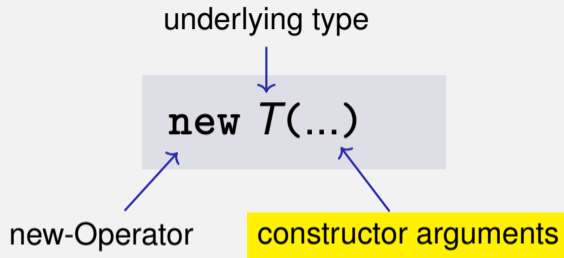
- So far: memory (local variables, function arguments) “lives” only inside a function call
- But now: memory chunk inside vector must not “die” before the vector itself
- Memory allocated with `new` is *not* automatically deallocated (= released)

# Outlook: new and delete

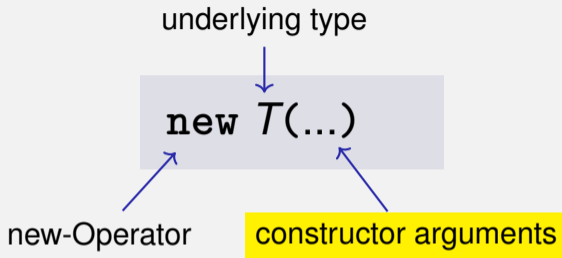
```
new T[expr]
```

- So far: memory (local variables, function arguments) “lives” only inside a function call
- But now: memory chunk inside vector must not “die” before the vector itself
- Memory allocated with `new` is *not* automatically deallocated (= released)
- Every `new` must have a matching `delete` that releases the memory explicitly → **in two weeks**

# new (Without Arrays)

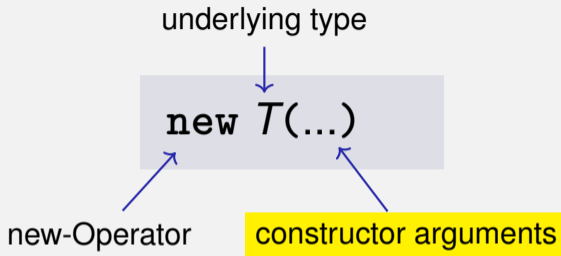


# new (Without Arrays)



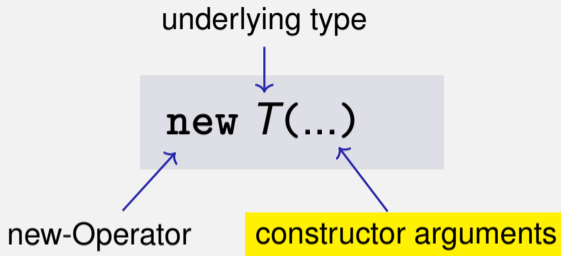
- **Effect:** memory for a new object of type  $T$  is allocated . . .

# new (Without Arrays)



- **Effect:** memory for a new object of type  $T$  is allocated ...
- ... and initialized by means of the matching constructor

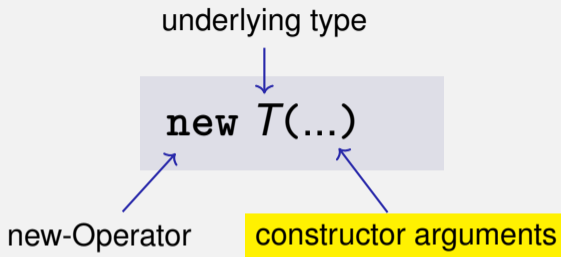
# new (Without Arrays)



- **Effect**: memory for a new object of type  $T$  is allocated ...
- ... and initialized by means of the matching constructor
- **Value**: address of the new  $T$  object, **Type**: Pointer  $T^*$



# new (Without Arrays)



- **Effect:** memory for a new object of type  $T$  is allocated . . .
- . . . and initialized by means of the matching constructor
- **Value:** address of the new  $T$  object, **Type:** Pointer  $T^*$
- Also true here: object “lives” until deleted explicitly (usefulness will become clearer later)

# Pointer Types

**T\***      Pointer type for base type T

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

# Pointer Types

**T\*** Pointer type for base type T

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

```
int* p; // Pointer to an int
std::string* q; // Pointer to a std::string
```

# Pointer Types

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

# Pointer Types

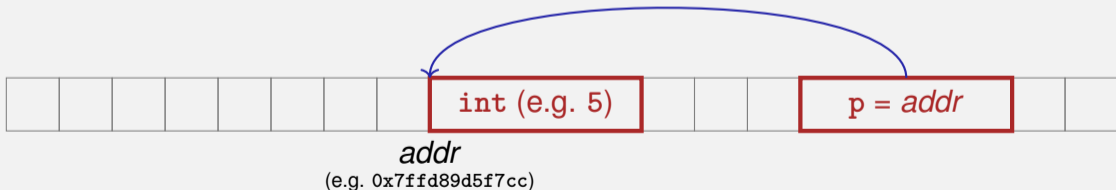
*Value* of a pointer to T is the *address* of an object of type T

```
int* p = ...;
std::cout << p; // e.g. 0x7ffd89d5f7cc
```

# Pointer Types

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

```
int* p = ...;
std::cout << p; // e.g. 0x7ffd89d5f7cc
```



# Address Operator

*Question:* How to obtain an object's address?

- 1 Directly, when creating a new object via `new`

# Address Operator

*Question:* How to obtain an object's address?

- 1 Directly, when creating a new object via `new`
- 2 For existing objects: via the *address operator* `&`

`&expr` ← `expr: l-value of type  $T$`



# Address Operator

*Question:* How to obtain an object's address?

- 1 Directly, when creating a new object via `new`
- 2 For existing objects: via the *address operator* `&`

`&expr` ← `expr: l-value of type  $T$`

- **Value** of the expression: the *address* of object (l-value) `expr`

# Address Operator

*Question:* How to obtain an object's address?

- 1 Directly, when creating a new object via `new`
- 2 For existing objects: via the *address operator* `&`

`&expr` ← `expr: l-value of type  $T$`

- **Value** of the expression: the *address* of object (l-value) `expr`
- **Type** of the expression: A pointer  $T^*$  (of type  $T$ )

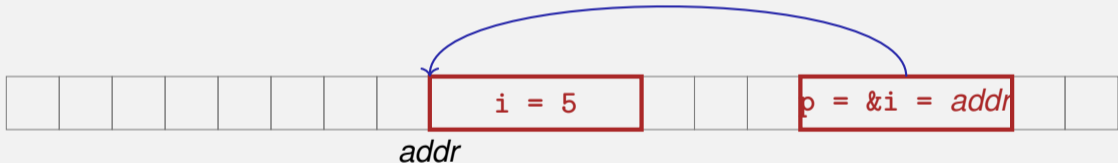
# Address Operator

```
int i = 5; // i initialised with 5
!1int* p = &i;
```



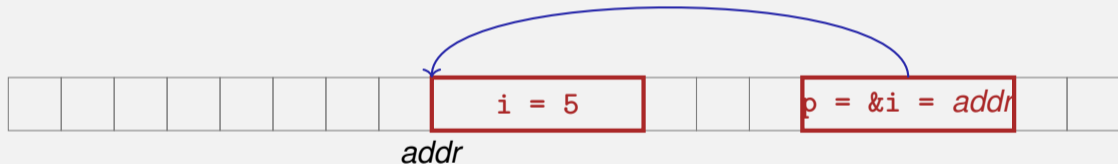
# Address Operator

```
int i = 5; // i initialised with 5
!1int* p = &i; // p initialised with address of i
```



# Address Operator

```
int i = 5; // i initialised with 5
!1int* p = &i; // p initialised with address of i
```



*Next question:* How to “follow” a pointer?

# Dereference Operator

*Answer:* by using the *dereference operator* \*

`*expr` ← `expr: r-value of type  $T^*$`

# Dereference Operator

*Answer:* by using the *dereference operator* \*

*\*expr* ← expr: r-value of type  $T^*$

- **Value** of the expression: the *value* of the object located at the address denoted by *expr*

# Dereference Operator

*Answer:* by using the *dereference operator* \*



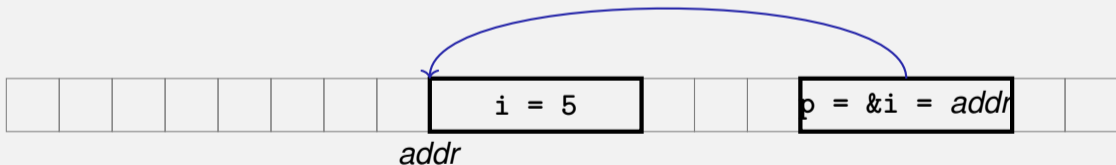
*\*expr* ← expr: r-value of type  $T^*$

- **Value** of the expression: the *value* of the object located at the address denoted by *expr*
- **Type** of the expression:  $T$



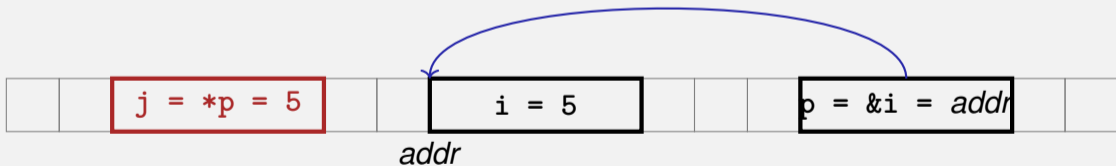
# Dereference Operator

```
int i = 5;
int* p = &i; // p = address of i
!1int j = *p;
```

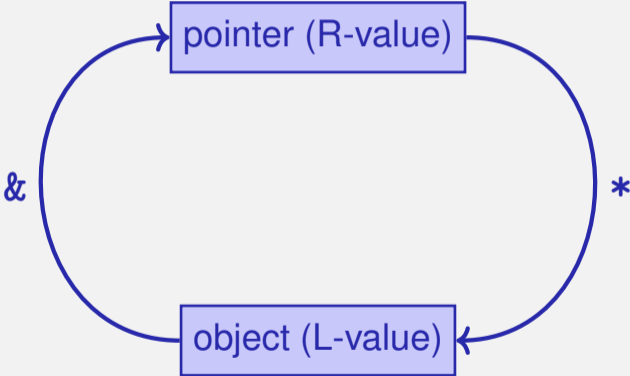


# Dereference Operator

```
int i = 5;
int* p = &i; // p = address of i
!1int j = *p; // j = 5
```



# Address and Dereference Operator



# Pointer Types

A  $T^*$  must actually point to a  $T$

```
int* p = ...; // p points to an int
double* q = p; // but q to a double → compiler
 error!
```

# Mnemonic Trick

The declaration

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

# Mnemonic Trick

The declaration

```
T* p; // p is of the type “pointer to T”
```

can be read as

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

# Null-Pointer

- Special pointer value that signals that no object is pointed to
- represented by the literal `nullptr` (convertible to `T*`)

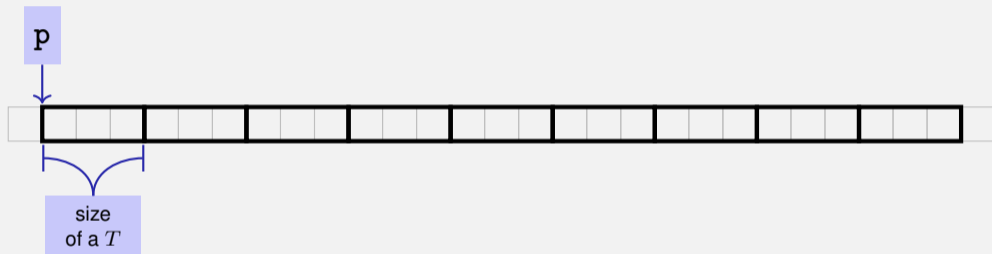
```
int* p = nullptr;
```

- Cannot be dereferenced (runtime error)
- Exists to avoid undefined behaviour

```
int* p; // p could point to anything
int* q = nullptr; // q explicitly points nowhere
```

# Pointer Arithmetic: Pointer plus int

```
 $T^* p = \text{new } T[n];$ // p points to first array element
```

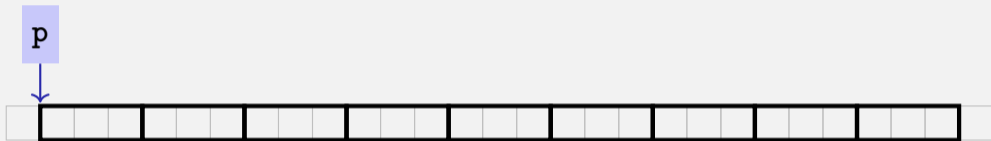


How to point to rear elements?



# Pointer Arithmetic: Pointer plus int

```
T* p = new T[n]; // p points to first array element
```



How to point to rear elements? → *Pointer arithmetic*:

# Pointer Arithmetic: Pointer plus int

```
T* p = new T[n]; // p points to first array element
```

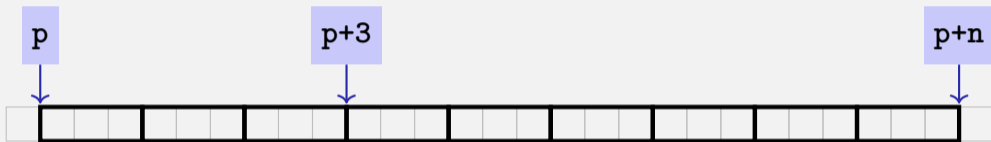


How to point to rear elements? → *Pointer arithmetic*:

- $p$  yields the *value* of the *first* array element,  $*p$  its *value*

# Pointer Arithmetic: Pointer plus int

```
T* p = new T[n]; // p points to first array element
```



How to point to rear elements? → *Pointer arithmetic*:

- $p$  yields the *value* of the *first* array element,  $*p$  its *value*
- $*(p + i)$  yields the value of the  $i$ th array element, for  $0 \leq i < n$

# Pointer Arithmetic: Pointer plus int

```
T* p = new T[n]; // p points to first array element
```



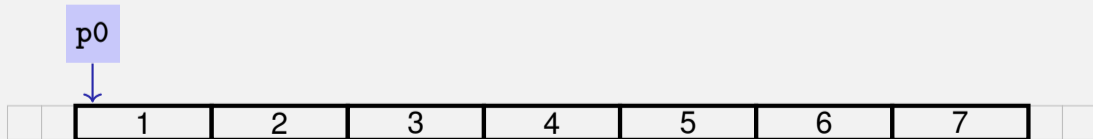
How to point to rear elements? → *Pointer arithmetic*:

- $p$  yields the *value* of the *first* array element,  $*p$  its *value*
- $*(p + i)$  yields the value of the  $i$ th array element, for  $0 \leq i < n$
- $*p$  is equivalent to  $*(p + 0)$

# Pointer Arithmetic: Pointer plus int

```
int* p0 = new int[7]{1,2,3,4,5,6,7}; // p0 points to
 1st element
!1int* p3 = p0 + 3;
!1-2*(p3 + 2) = 600;

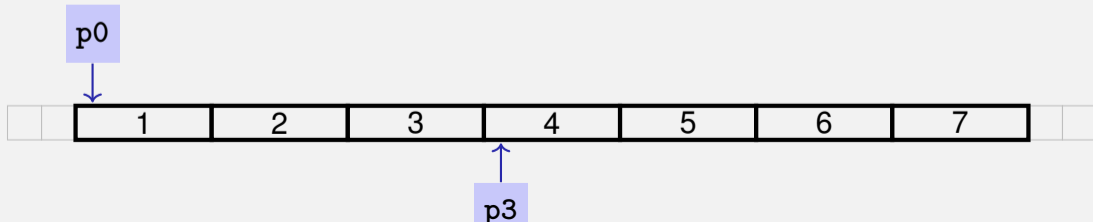
!1-3std::cout << *(p0 + 5);
```



# Pointer Arithmetic: Pointer plus int

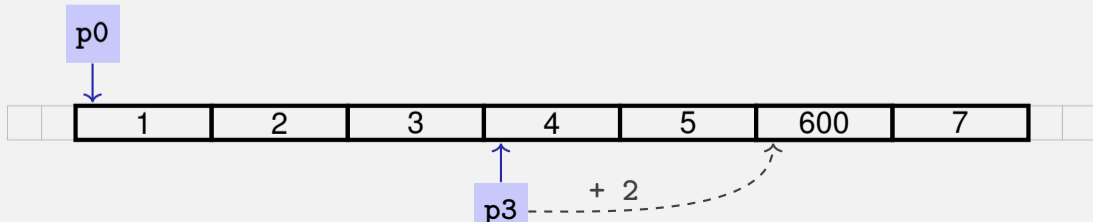
```
int* p0 = new int[7]{1,2,3,4,5,6,7}; // p0 points to
1st element
!1int* p3 = p0 + 3; // p3 points to 4th element
!1-2*(p3 + 2) = 600;

!1-3std::cout << *(p0 + 5);
```



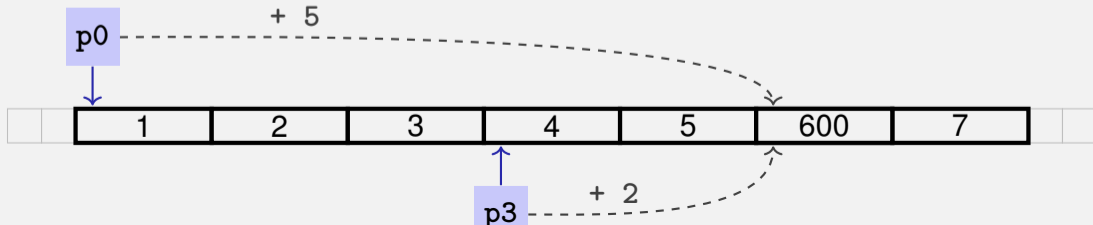
# Pointer Arithmetic: Pointer plus int

```
int* p0 = new int[7]{1,2,3,4,5,6,7}; // p0 points to
1st element
!1int* p3 = p0 + 3; // p3 points to 4th element
!1-2*(p3 + 2) = 600; // set value of 6th element to
600
!1-3std::cout << *(p0 + 5);
```



# Pointer Arithmetic: Pointer plus int

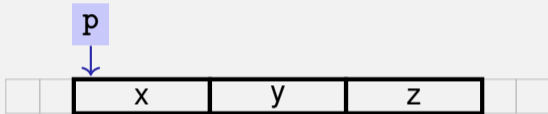
```
int* p0 = new int [7]{1,2,3,4,5,6,7}; // p0 points to
1st element
!1int* p3 = p0 + 3; // p3 points to 4th element
!1-2*(p3 + 2) = 600; // set value of 6th element to
600
!1-3std::cout << *(p0 + 5); // output 6th element's
value (i.e. 600)
```





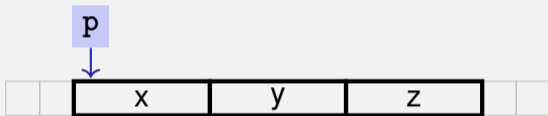
# Sequential Pointer Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```



# Sequential Pointer Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```

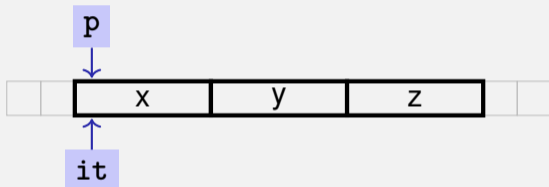


```
for (3char* it = p;
 4,7,10,13it != p + 3;
 ++it) {

 std::cout << *it << ' ';
}
```

# Sequential Pointer Iteration

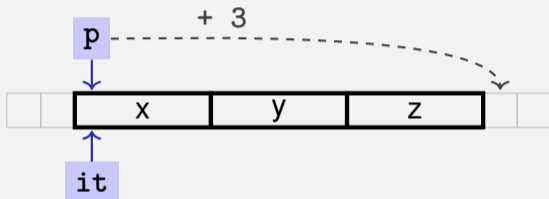
```
char* p = new char[3]{'x', 'y', 'z'};
```



```
for (3char* it = p; it points to first element it != p + 3; ++it) {
 std::cout << *it << ' ';
```

# Sequential Pointer Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```

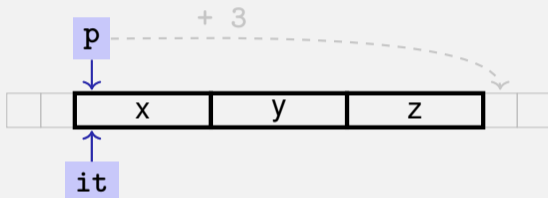


```
for (3char* it = p;
 4,7,10,13it != p + Abort if end reached
 ++it) {

 std::cout << *it << ' ';
```

# Sequential Pointer Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```



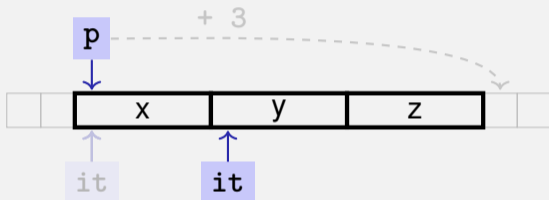
```
for (3char* it = p;
 4,7,10,13it != p + 3;
 ++it) {
```

```
std::cout << *it << ' ';
```

← Output current element: 'x'

# Sequential Pointer Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```



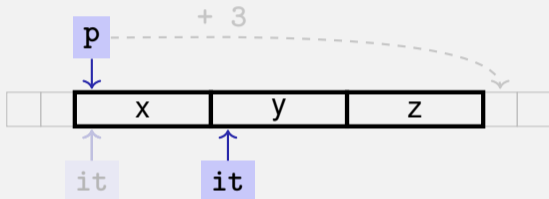
```
for (3char* it = p;
 4,7,10,13it != p + 3;
 ++it) ← {
```

Advance pointer element-wise

```
std::cout << *it << ' '; // x
```

# Sequential Pointer Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```

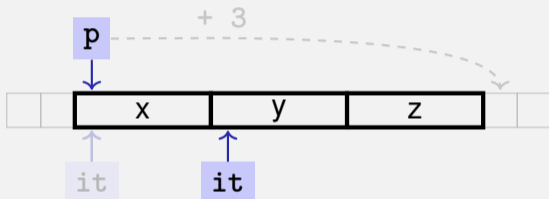


```
for (3char* it = p;
 4,7,10,13it != p + 3;
 ++it) {

 std::cout << *it << ' '; // x
```

# Sequential Pointer Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```



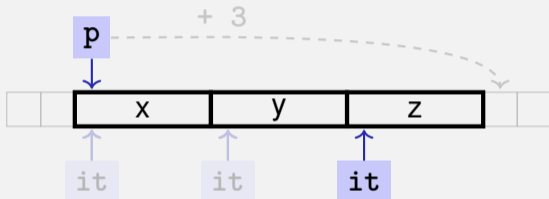
```
for (3char* it = p;
 4,7,10,13it != p + 3;
 ++it) {

 std::cout << *it << ' '; // x y
```



# Sequential Pointer Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```

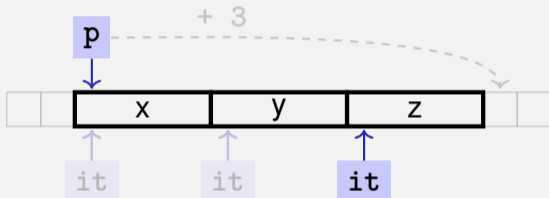


```
for (3char* it = p;
 4,7,10,13it != p + 3;
 ++it) {

 std::cout << *it << ' '; // x y
```

# Sequential Pointer Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```

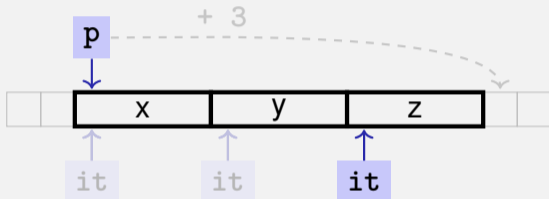


```
for (3char* it = p;
 4,7,10,13it != p + 3;
 ++it) {

 std::cout << *it << ' '; // x y
```

# Sequential Pointer Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```

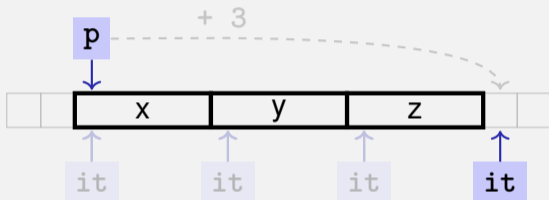


```
for (3char* it = p;
 4,7,10,13it != p + 3;
 ++it) {

 std::cout << *it << ' '; // x y z
```

# Sequential Pointer Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```

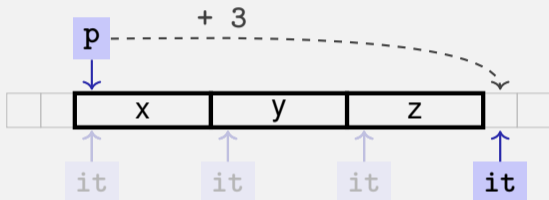


```
for (3char* it = p;
 4,7,10,13it != p + 3;
 ++it) {

 std::cout << *it << ' '; // x y z
}
```

# Sequential Pointer Iteration

```
char* p = new char[3]{'x', 'y', 'z'};
```

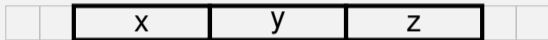


```
for (3char* it = p;
 4,7,10,13it != p + 3;
 ++it) {

 std::cout << *it << ' '; // x y z
}
```

# Random Access to Arrays

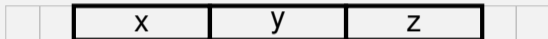
```
char* p = new char[3]{'x', 'y', 'z'};
```



- The expression `*(p + i)`
- can also be written as `p[i]`

# Random Access to Arrays

```
char* p = new char[3]{'x', 'y', 'z'};
```



- The expression `*(p + i)`
- can also be written as `p[i]`
- E.g. `p[1] == *(p + 1) == 'y'`

# Random Access to Arrays

iteration over an array via indices and *random access*:

```
char* p = new char[3]{'x', 'y', 'z'};

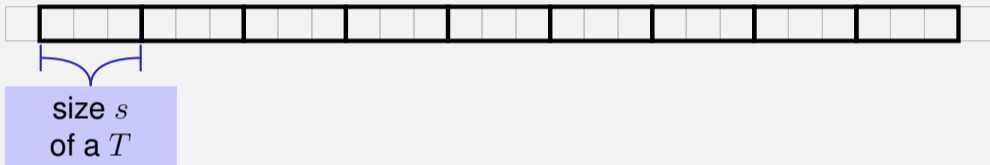
for (int i = 0; i < 3; ++i)
 std::cout << p[i] << ' ';
```

*But:* this is less *efficient* than the previously shown *sequential* access via pointer iteration



# Random Access to Arrays

```
 $T^* p = \text{new } T[n];$
```



# Random Access to Arrays

```
 $T^* \text{ p} = \text{new } T[\text{n}];$
```



- Access  $p[i]$ , i.e.  $*(p + i)$ , “costs” computation  $p + i \cdot s$

# Random Access to Arrays

```
T* p = new T[n];
```



- Access  $p[i]$ , i.e.  $*(p + i)$ , “costs” computation  $p + i \cdot s$
- Iteration via *random access* ( $p[0], p[1], \dots$ ) costs one addition and one multiplication per access

# Random Access to Arrays

```
T* p = new T[n];
```



- Access  $p[i]$ , i.e.  $*(p + i)$ , “costs” computation  $p + i \cdot s$
- Iteration via *random access* ( $p[0], p[1], \dots$ ) costs one addition and one multiplication per access
- Iteration via *sequential access* ( $++p, ++p, \dots$ ) costs only one addition per access

# Random Access to Arrays

```
T* p = new T[n];
```



- Access  $p[i]$ , i.e.  $*(p + i)$ , “costs” computation  $p + i \cdot s$
- Iteration via *random access* ( $p[0], p[1], \dots$ ) costs one addition and one multiplication per access
- Iteration via *sequential access* ( $++p, ++p, \dots$ ) costs only one addition per access
- Sequential access is thus to be preferred for iterations

# Reading a book ... with random access

## 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
- ....

## Random Access

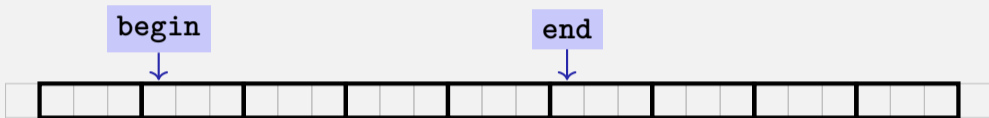
- open book on page 1
- close book
- open book on pages 2-3
- close book
- open book on pages 4-5
- close book
- ....

## Sequential Access

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

# Arrays in Functions

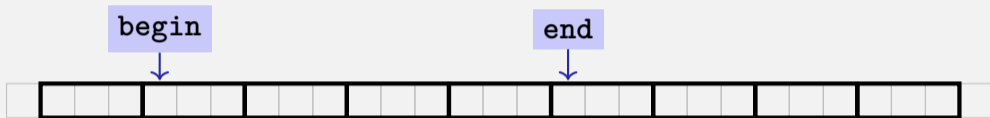
C++ *covention*: arrays (or a segment of it) are passed using two pointers





# Arrays in Functions

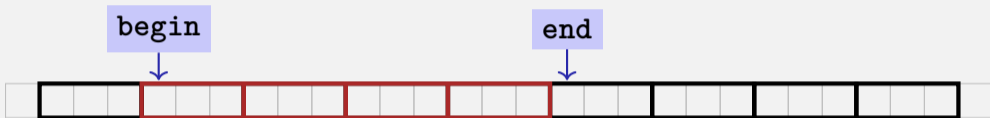
C++ *covention*: arrays (or a segment of it) are passed using two pointers



- **begin**: Pointer to the first element
- **end**: Pointer *past* the last element

# Arrays in Functions

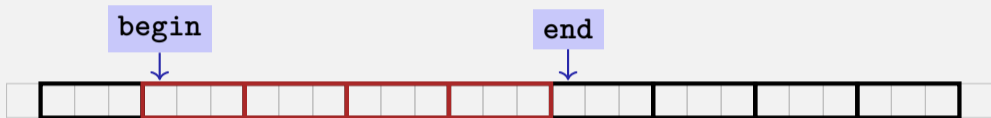
C++ *covention*: arrays (or a segment of it) are passed using two pointers



- **begin**: Pointer to the first element
- **end**: Pointer *past* the last element
- **[begin, end)** Designates the elements of the segment of the array

# Arrays in Functions

C++ *covention*: arrays (or a segment of it) are passed using two pointers



- **begin**: Pointer to the first element
- **end**: Pointer *past* the last element
- `[begin, end)` Designates the elements of the segment of the array
- `[begin, end)` is empty if `begin == end`
- `[begin, end)` must be a *valid range*, i.e. a (pot. empty) array segment

# Arrays in (mutating) Functions: fill

```
// PRE: [begin, end) is a valid range
// POST: Every element within [begin, end) was set to
 value
void fill(1-int* begin, 1-int* end, int value) {
 for (int* p = begin; p != end; ++p)
 *p = value;
}
```

# Arrays in (mutating) Functions: fill

```
// PRE: [begin, end) is a valid range
// POST: Every element within [begin, end) was set to
// value
void fill(1-int* begin, 1-int* end, int value) {
 for (int* p = begin; p != end; ++p)
 *p = value;
}

...
int* p = new int[5];
fill(2-p, 2-p+5, 1); // Array at p
// becomes {1, 1, 1, 1, 1}
```

# Functions with/without Effect

- Pointers can (like references) be used for functions with effect.  
Example: `fill`

# Functions with/without Effect

- Pointers can (like references) be used for functions with effect.  
Example: `fill`
- But many functions don't have an effect, they only read the data
- $\Rightarrow$  Use of `const`

# Functions with/without Effect

- Pointers can (like references) be used for functions with effect.  
Example: `fill`
- But many functions don't have an effect, they only read the data
- $\Rightarrow$  Use of `const`
- So far, for example:

```
const int zero = 0;
const int& nil = zero;
```



# Positioning of Const

`const T` is equivalent to `T const` (and can be written like this):

```
const int zero = ... \iff int const zero = ...
const int& nil = ... \iff int const& nil = ...
```

# Positioning of Const

`const T` is equivalent to `T const` (and can be written like this):

```
const int zero = ... \iff int const zero = ...
const int& nil = ... \iff int const& nil = ...
```

Both keyword orders are used in praxis

# Const and Pointers

Read the declaration from right to left

```
int const p;
```

p is a constant integer

# Const and Pointers

Read the declaration from right to left

```
int const p;
```

p is a constant integer

```
int const* p;
```

p is a pointer to a constant integer

# Const and Pointers

Read the declaration from right to left

```
int const p;
```

p is a constant integer

```
int const* p;
```

p is a pointer to a constant integer

```
int* const p;
```

p is a constant pointer to an integer

# Const and Pointers

Read the declaration from right to left

```
int const p;
```

p is a constant integer

```
int const* p;
```

p is a pointer to a constant integer

```
int* const p;
```

p is a constant pointer to an integer

```
int const* const p;
```

p is a constant pointer to a constant integer

# Non-mutating Functions: print

```
// PRE: [begin, end) is a valid range
// POST: The values in [begin, end) were printed
void print(
 2-int const* const begin,
 2-const int* const end) {

 for (3-int const* p = begin; p != end; ++p)
 std::cout << *p << ' ';
}
```

# Non-mutating Functions: print

```
// PRE: [begin, end) is a valid range
// POST: The values in [begin, end) were printed
void print(
 2-int const* const begin,
 2-const int* const end) {
 for (3-int const* p = begin; p != end; ++p)
 std::cout << *p << ' ';
}
```

Const pointer to const int

Likewise (but different keyword order)



# Non-mutating Functions: print

```
// PRE: [begin, end) is a valid range
// POST: The values in [begin, end) were printed
void print(
 2-int const* const begin,
 2-const int* const end) {
 for (3-int const* p = begin; p != end; ++p)
 std::cout << *p << ' ';
}
```

Const pointer to const int

Likewise (but different keyword order)

Pointer, *not const*, to const int

# Arrays, new, Pointer: Conclusion

- Arrays are contiguous chunks of memory of statically unknown size

# Arrays, new, Pointer: Conclusion

- Arrays are contiguous chunks of memory of statically unknown size
- `new T[n]` allocates a  $T$ -array of size  $n$

# Arrays, new, Pointer: Conclusion

- Arrays are contiguous chunks of memory of statically unknown size
- `new T[n]` allocates a  $T$ -array of size  $n$
- `T* p = new T[n]`: pointer `p` points to the first array element

# Arrays, new, Pointer: Conclusion

- Arrays are contiguous chunks of memory of statically unknown size
- `new T[n]` allocates a  $T$ -array of size  $n$
- `T* p = new T[n]`: pointer `p` points to the first array element
- Pointer arithmetic enables accessing rear array elements

# Arrays, new, Pointer: Conclusion

- Arrays are contiguous chunks of memory of statically unknown size
- `new T[n]` allocates a  $T$ -array of size  $n$
- `T* p = new T[n]`: pointer `p` points to the first array element
- Pointer arithmetic enables accessing rear array elements
- Sequentially iterating over arrays via pointers is more efficient than random access

# Arrays, new, Pointer: Conclusion

- Arrays are contiguous chunks of memory of statically unknown size
- `new T[n]` allocates a  $T$ -array of size  $n$
- `T* p = new T[n]`: pointer `p` points to the first array element
- Pointer arithmetic enables accessing rear array elements
- Sequentially iterating over arrays via pointers is more efficient than random access
- `new T` allocates memory for (and initialises) a single  $T$ -object, and yields a pointer to it

# Arrays, new, Pointer: Conclusion

- Arrays are contiguous chunks of memory of statically unknown size
- `new T[n]` allocates a  $T$ -array of size  $n$
- `T* p = new T[n]`: pointer `p` points to the first array element
- Pointer arithmetic enables accessing rear array elements
- Sequentially iterating over arrays via pointers is more efficient than random access
- `new T` allocates memory for (and initialises) a single  $T$ -object, and yields a pointer to it
- Pointers can point to something (not) `const`, and they can be (not) `const` themselves



# Arrays, `new`, Pointer: Conclusion

- Arrays are contiguous chunks of memory of statically unknown size
- `new T[n]` allocates a  $T$ -array of size  $n$
- `T* p = new T[n]`: pointer `p` points to the first array element
- Pointer arithmetic enables accessing rear array elements
- Sequentially iterating over arrays via pointers is more efficient than random access
- `new T` allocates memory for (and initialises) a single  $T$ -object, and yields a pointer to it
- Pointers can point to something (not) `const`, and they can be (not) `const` themselves
- Memory allocated by `new` is *not* automatically released (more on this soon)

# Arrays, new, Pointer: Conclusion

- Arrays are contiguous chunks of memory of statically unknown size
- `new T[n]` allocates a  $T$ -array of size  $n$
- `T* p = new T[n]`: pointer `p` points to the first array element
- Pointer arithmetic enables accessing rear array elements
- Sequentially iterating over arrays via pointers is more efficient than random access
- `new T` allocates memory for (and initialises) a single  $T$ -object, and yields a pointer to it
- Pointers can point to something (not) `const`, and they can be (not) `const` themselves
- Memory allocated by `new` is *not* automatically released (more on this soon)
- Pointers and references are related, both “link” to objects in memory. See also additional the slides `pointers.pdf`)

# Array-based Vector


- Vectors . . . that somehow rings a bell



## Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<T>` (später, nur kurz angeschnitten)


# Array-based Vector

- Vectors . . . that somehow rings a bell  

- Now we know how to allocate memory chunks of arbitrary size . . .

## Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<T>` (später, nur kurz angeschnitten)


# Array-based Vector

- Vectors . . . that somehow rings a bell  

- Now we know how to allocate memory chunks of arbitrary size . . .
- . . . we can implement a vector, based on such a chunk of memory

## Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<T>` (später, nur kurz angeschnitten)

# Array-based Vector

- Vectors ... that somehow rings a bell  

- Now we know how to allocate memory chunks of arbitrary size ...
- ... we can implement a vector, based on such a chunk of memory
- `avec` – an array-based vector of `int` elements

## Unser eigener Vektor!

- Wir implementieren unseren eigenen Vektor: `vec`
- Schritt 1: `vec<int>` (heute)
- Schritt 2: `vec<T>` (später, nur kurz angeschnitten)

# Array-based Vector `avec`: Class Signature

```
class avec {
 // Private (internal) state:
 1int* elements;
 2unsigned int count;
```



Pointer to first element

```
}
```

# Array-based Vector avec: Class Signature

```
class avec {
 // Private (internal) state:
 1int* elements; // Pointer to first element
 2unsigned int count; ← Number of elements

}
```



# Array-based Vector avec: Class Signature

```
class avec {
 // Private (internal) state:
 1int* elements; // Pointer to first element
 2unsigned int count; // Number of elements

 public: // Public interface:
 3avec(unsigned int size); ← Constructor
 4unsigned int size() const;
 5int& operator [] (int i);
 6void print(std::ostream& sink) const;
}
```

# Array-based Vector avec: Class Signature

```
class avec {
 // Private (internal) state:
 1int* elements; // Pointer to first element
 2unsigned int count; // Number of elements

 public: // Public interface:
 3avec(unsigned int size); // Constructor
 4unsigned int size() const; ← Size of vector
 5int& operator[] (int i);
 6void print(std::ostream& sink) const;
}
```

# Array-based Vector avec: Class Signature

```
class avec {
 // Private (internal) state:
 1int* elements; // Pointer to first element
 2unsigned int count; // Number of elements

 public: // Public interface:
 3avec(unsigned int size); // Constructor
 4unsigned int size() const; // Size of vector
 5int& operator[](int i); ← Access an element
 6void print(std::ostream& sink) const;
}
```

# Array-based Vector avec: Class Signature

```
class avec {
 // Private (internal) state:
 1int* elements; // Pointer to first element
 2unsigned int count; // Number of elements

 public: // Public interface:
 3avec(unsigned int size); // Constructor
 4unsigned int size() const; // Size of vector
 5int& operator[](int i); // Access an element
 6void print(std::ostream& sink) const; ←
}
```

Output elements

# Array-based Vector avec: Class Signature

```
class avec {
 // Private (internal) state:
 1int* elements; // Pointer to first element
 2unsigned int count; // Number of elements

 public: // Public interface:
 3avec(unsigned int size); // Constructor
 4unsigned int size() const; // Size of vector
 5int& operator[](int i); // Access an element
 6void print(std::ostream& sink) const; // Output elems.
}
```

# Constructor avec::avec()

```
avec::avec(unsigned int size)
 : 1count(size) ← {
 2elements = new int[size];
}
```

Save size

# Constructor avec::avec()

```
avec::avec(unsigned int size)
 : 1count(size) {

 2elements = new int[size];
}
```

Allocate memory



# Constructor avec::avec()

```
avec::avec(unsigned int size)
 : 1count(size) {

 2elements = new int[size];
}
```

Side remark: vector is not initialised with a default value



# Excursion: Accessing Member Variables

```
avec::avec(unsigned int size): count(size) {
 elements = new int[size];
}
```

- `elements` is a member variable of our `avec` instance

# Excursion: Accessing Member Variables

```
avec::avec(unsigned int size): count(size) {
 elements = new int[size];
}
```

- `elements` is a member variable of our `avec` instance
- That instance can be accessed via the *pointer* `this`

# Excursion: Accessing Member Variables

```
avec::avec(unsigned int size): count(size) {
 (*this).elements = new int[size];
}
```

- `elements` is a member variable of our `avec` instance
- That instance can be accessed via the *pointer* `this`
- `elements` is a shorthand for `(*this).elements`

# Excursion: Accessing Member Variables

```
avec::avec(unsigned int size): count(size) {
 this->elements = new int[size];
}
```

- `elements` is a member variable of our `avec` instance
- That instance can be accessed via the *pointer* `this`
- `elements` is a shorthand for `(*this).elements`
- Equivalent, but shorter: `this->elements`

# Excursion: Accessing Member Variables

```
avec::avec(unsigned int size): count(size) {
 this->elements = new int[size];
}
```

- `elements` is a member variable of our `avec` instance
- That instance can be accessed via the *pointer* `this`
- `elements` is a shorthand for `(*this).elements`
- Equivalent, but shorter: `this->elements`
- Mnemonic trick: “Follow the pointer to the member variable”

# Function `avec::size()`

```
int avec::size() const ← {
 return this->count;
}
```

Doesn't modify the vector

# Function `avec::size()`

```
int avec::size() const {
 return this->count; }
}
```



Return size

Usage example:

```
avec v = avec(7);
assert(v.size() == 7); // ok
```

# Function avec::operator []

```
int& avec::operator [] (int i) {
 return this->elements[i];
}
```



Return ith element



## Function avec ::operator []

```
int& avec::operator [] (int i) {
 return this->elements[i];
}
```

Element access with index check:

```
int& avec::at(int i) const {
 assert(0 <= i && i < this->count);

 return this->elements[i];
}
```

# Function avec::operator []

```
int& avec::operator [] (int i) {
 return this->elements[i];
}
```

Usage example:

```
avec v = avec(7);
std::cout << v[6]; // Outputs a "random" value
v[6] = 0;
std::cout << v[6]; // Outputs 0
```

# Function avec::print()

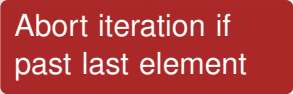
Output elements using sequential access:

```
void avec::print(std::ostream& sink) const {
 for (1int* p = this->elements; ← Pointer to first element
 2p != this->elements + this->count;
 3++p)
 {
 4sink << *p << ' ';
 }
}
```

# Function `avec::print()`

Output elements using sequential access:


```
void avec::print(std::ostream& sink) const {
 for (int* p = this->elements;
 p != this->elements + this->count;
 ++p)
 {
 sink << *p << ' ';
 }
}
```



Abort iteration if  
past last element

# Function `avec::print()`

Output elements using sequential access:

```
void avec::print(std::ostream& sink) const {
 for (int* p = this->elements;
 p != this->elements + this->count;
 p++) 
 {
 sink << *p << ' ';
 }
}
```

Advance pointer element-wise

# Function `avec::print()`

Output elements using sequential access:

```
void avec::print(std::ostream& sink) const {
 for (int* p = this->elements;
 p != this->elements + this->count;
 p++) ← Advance pointer element-wise
 {
 sink << *p << ' '; ← Output current element
 }
}
```

# Function `avec::print()`

Finally: overload output operator:

```
_____ operator<<(_____ sink,
 _____ vec) {
 vec.print(sink);
 return _____;
}
```

# Function `avec::print()`

Finally: overload output operator:

```
std::ostream& operator<<(std::ostream& sink,
 const avec& vec) {
 vec.print(sink);
 return sink;
}
```



# Function `avec::print()`

Finally: overload output operator:

```
std::ostream& operator<<(std::ostream& sink,
 const avec& vec) {
 vec.print(sink);
 return sink;
}
```

Observations:

- Constant reference to `vec`, since unchanged

# Function `avec::print()`

Finally: overload output operator:

```
std::ostream& operator<<(std::ostream& sink,
 const avec& vec) {
 vec.print(sink);
 return sink;
}
```

Observations:

- Constant reference to `vec`, since unchanged
- But not to `sink`: Outputting elements equals change

# Function `avec::print()`

Finally: overload output operator:

```
std::ostream& operator<<(std::ostream& sink,
 const avec& vec) {
 vec.print(sink);
 return sink;
}
```

Observations:

- Constant reference to `vec`, since unchanged
- But not to `sink`: Outputting elements equals change
- `sink` is returned to enable output chaining, e.g.

```
std::cout << v << '\n'
```

# Further Functions?

```
class avec {
 ...
 void push_front(int e) // Prepend e to vector
 void push_back(int e) // Append e to vector
 void remove(unsigned int i) // Cut out ith element
 ...
}
```

# Further Functions?

```
class avec {
 ...
 void push_front(int e) // Prepend e to vector
 void push_back(int e) // Append e to vector
 void remove(unsigned int i) // Cut out ith element
 ...
}
```

Commonalities: such operations need to change the vector's *size*

# Resizing arrays

An allocated block of memory (e.g. `new int [3]`) cannot be resized later on

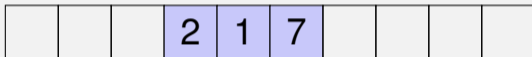
# Resizing arrays

An allocated block of memory (e.g. `new int [3]`) cannot be resized later on



# Resizing arrays

An allocated block of memory (e.g. `new int [3]`) cannot be resized later on



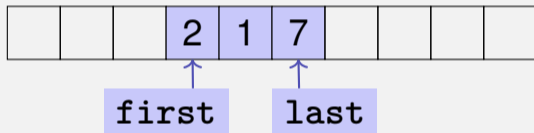
Possibility:

- Allocate more memory than initially necessary



# Resizing arrays

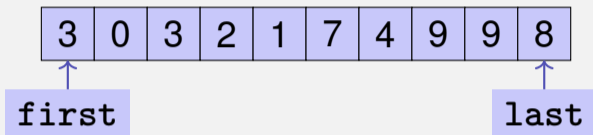
An allocated block of memory (e.g. `new int [3]`) cannot be resized later on



Possibility:

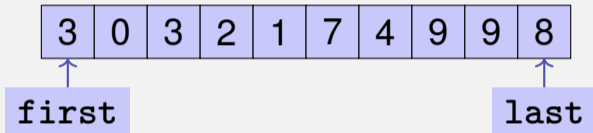
- Allocate more memory than initially necessary
- Fill from inside out, with pointers to first and last element

# Resizing arrays



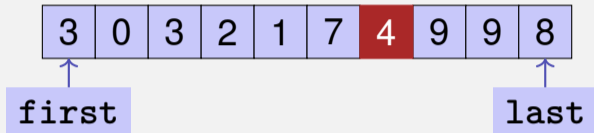
- But eventually, all slots will be in use

# Resizing arrays



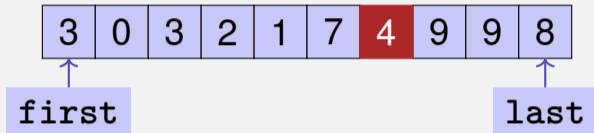
- But eventually, all slots will be in use
- Then unavoidable: Allocate larger memory block and copy data over

# Resizing arrays

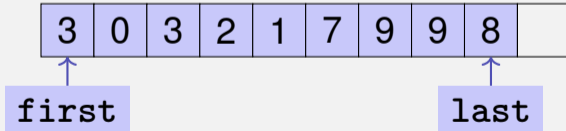


Deleting elements requires shifting (by copying) all preceding or following elements

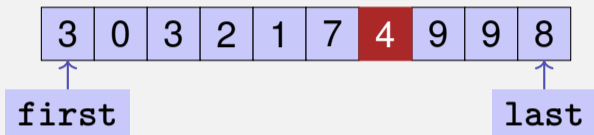
# Resizing arrays



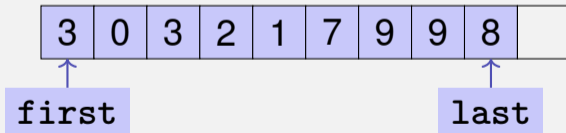
Deleting elements requires shifting (by copying) all preceding or following elements



# Resizing arrays



Deleting elements requires shifting (by copying) all preceding or following elements



Similar: inserting at arbitrary position

# 19. Dynamic Data Structures II

Linked Lists, Vectors as Linked Lists

# Different Memory Layout: Linked List

- *No* contiguous area of memory and *no* random access





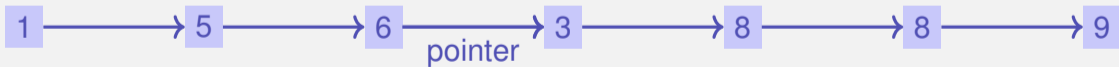
# Different Memory Layout: Linked List

- *No* contiguous area of memory and *no* random access
- Each element points to its successor



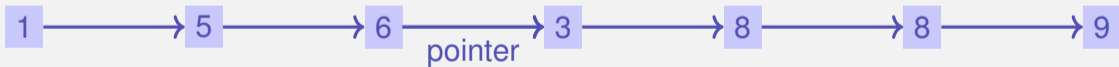
# Different Memory Layout: Linked List

- *No* contiguous area of memory and *no* random access
- Each element points to its successor



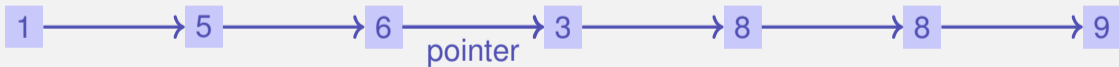
# Different Memory Layout: Linked List

- *No* contiguous area of memory and *no* random access
- Each element points to its successor
- Insertion and deletion of *arbitrary* elements is simple



# Different Memory Layout: Linked List

- *No* contiguous area of memory and *no* random access
- Each element points to its successor
- Insertion and deletion of *arbitrary* elements is simple



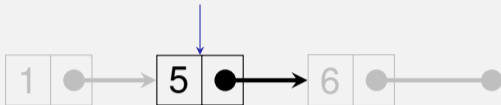
⇒ Our vector can be implemented as a linked list

# Linked List: Zoom



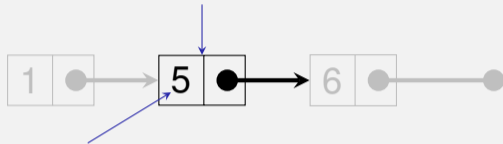
# Linked List: Zoom

element (type struct llnode)



# Linked List: Zoom

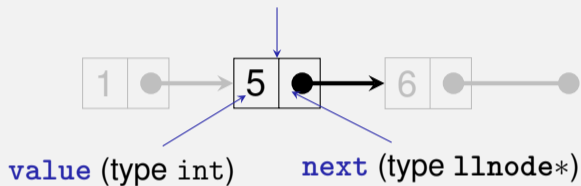
`element (type struct llnode)`



`value (type int)`

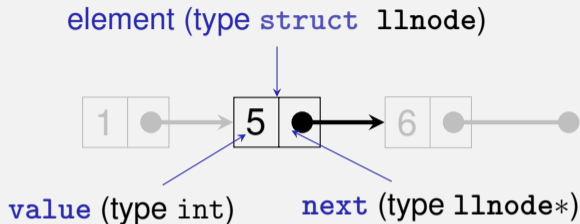
# Linked List: Zoom

element (type struct llnode)





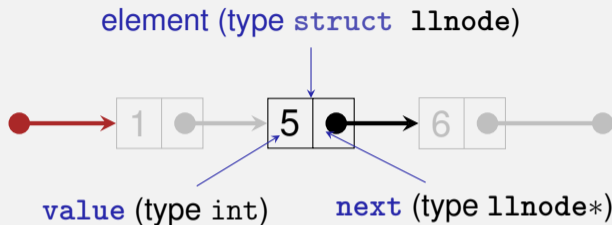
# Linked List: Zoom



```
struct llnode {
 int value;
 llnode* next;

 llnode(int v, llnode* n): value(v), next(n) {} //
 Constructor
};
```


# Vector = Pointer to the First Element



```
class llvec {
 llnode* head;
public:
 // Public interface identical to avec's
 llvec(unsigned int size);
 unsigned int size() const;
 ...
}
```

# Function `llvec::print()`

```
struct llnode {
 int value;
 llnode* next;
 ...
};
```

```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;  Pointer to first element
 n != nullptr;
 n = n->next)
 {
 sink << n->value << ' ';
 }
}
```


# Function `llvec::print()`

```
struct llnode {
 int value;
 llnode* next;
 ...
};
```

```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;
 2n != nullptr; ← Abort if end reached
 3n = n->next)
 {
 4sink << n->value << ' ' ;
 }
}
```

# Function `llvec::print()`

```
struct llnode {
 int value;
 llnode* next;
 ...
};
```

```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;
 2n != nullptr;
 3n = n->next) 
 {
 4sink << n->value << ' ';
 }
}
```

Advance pointer element-wise

# Function `llvec::print()`

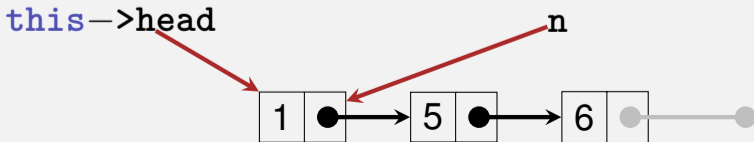
```
struct llnode {
 int value;
 llnode* next;
 ...
};
```

```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;
 2n != nullptr;
 3n = n->next)
 {
 4sink << n->value << ' '; ←
 }
}
```

Output current element

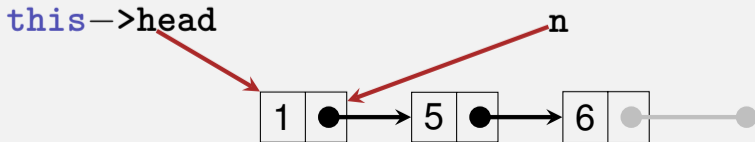
# Function `llvec::print()`

```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;
 258n != nullptr;
 n = n->next)
 {
 sink << n->value << ' ';
 }
}
```



# Function `llvec::print()`

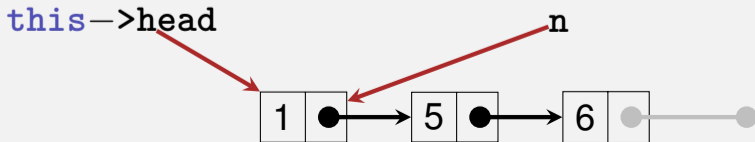
```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;
 n != nullptr;
 n = n->next)
 {
 sink << n->value << ' ';
 }
}
```





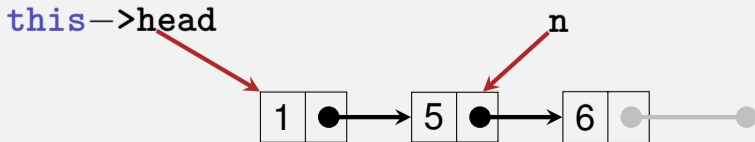
# Function `llvec::print()`

```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;
 258n != nullptr;
 n = n->next)
 {
 sink << n->value << ' '; // 1
 }
}
```



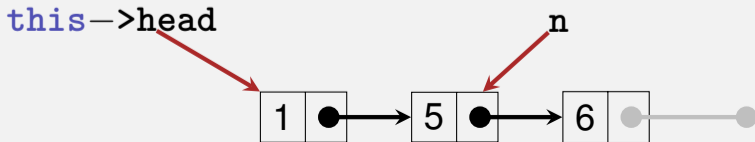
# Function `llvec::print()`

```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;
 n != nullptr;
 n = n->next)
 {
 sink << n->value << ' '; // 1
 }
}
```



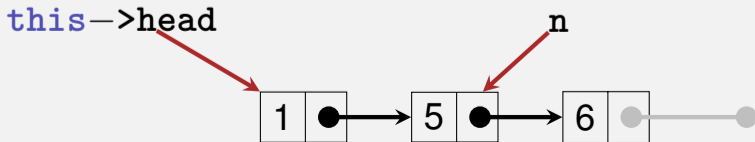
# Function `llvec::print()`

```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;
 n != nullptr;
 n = n->next)
 {
 sink << n->value << ' '; // 1
 }
}
```



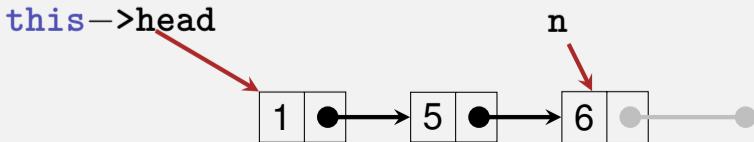
# Function `llvec::print()`

```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;
 n != nullptr;
 n = n->next)
 {
 sink << n->value << ' '; // 1 5
 }
}
```



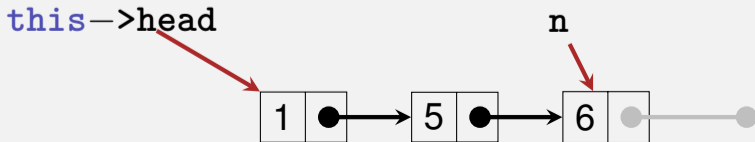
# Function `llvec::print()`

```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;
 258n != nullptr;
 n = n->next)
 {
 sink << n->value << ' '; // 1 5
 }
}
```



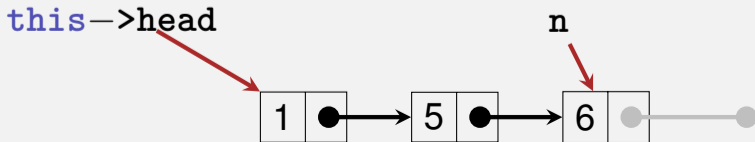
# Function `llvec::print()`

```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;
 258n != nullptr;
 n = n->next)
 {
 sink << n->value << ' '; // 1 5
 }
}
```



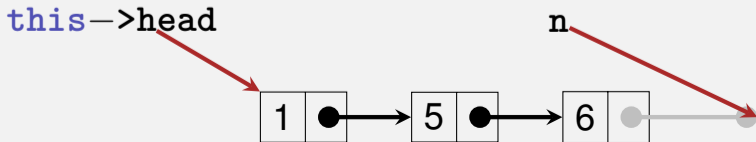
# Function `llvec::print()`

```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;
 n != nullptr;
 n = n->next)
 {
 sink << n->value << ' '; // 1 5 6
 }
}
```



# Function `llvec::print()`

```
void llvec::print(std::ostream& sink) const {
 for (llnode* n = this->head;
 258n != nullptr;
 n = n->next)
 {
 sink << n->value << ' '; // 1 5 6
 }
}
```





# Function `llvec::operator []`

Accessing *i*th Element is implemented similarly to `print()`:

```
int& llvec::operator [] (unsigned int i) {
 1llnode* n = this->head; ← Pointer to first element

 2for (; 0 < i; --i)
 2n = n->next;

 3return n->value;
}
```

# Function `llvec::operator []`

Accessing  $i$ th Element is implemented similarly to `print()`:

```
int& llvec::operator [] (unsigned int i) {
 1llnode* n = this->head;

 2for (; 0 < i; --i) |
 2n = n->next; |

 3return n->value;
}
```




Step to  $i$ th element

# Function `llvec::operator []`

Accessing *i*th Element is implemented similarly to `print()`:

```
int& llvec::operator [] (unsigned int i) {
 1llnode* n = this->head;

 2for (; 0 < i; --i)
 2n = n->next;

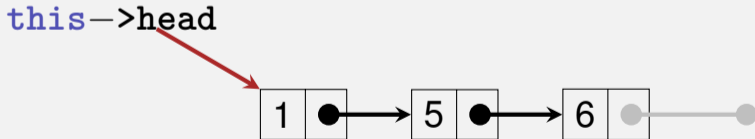
 3return n->value; 
}
```

Return *i*th element

# Function `llvec::push_front()`

Advantage `llvec`: Prepending elements is very easy:

```
void llvec::push_front(int e) {
 this->head =
 new llnode{e, this->head};
}
```

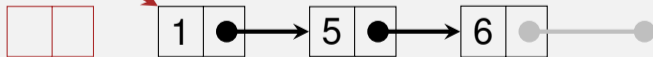


# Function `llvec::push_front()`

Advantage `llvec`: Prepending elements is very easy:

```
void llvec::push_front(int e) {
 this->head =
 new llnode{e, this->head};
}
```

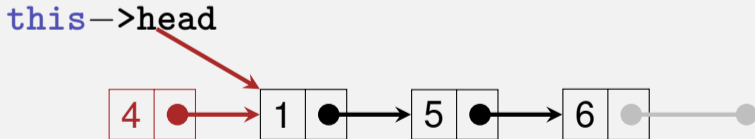
`this->head`



# Function `llvec::push_front()`

Advantage `llvec`: Prepending elements is very easy:

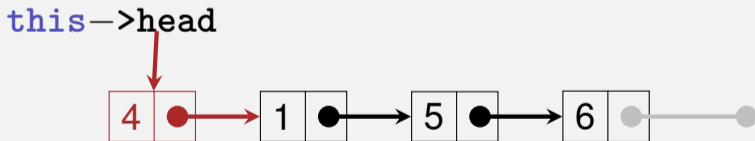
```
void llvec::push_front(int e) {
 this->head =
 new llnode{e, this->head};
}
```



# Function `llvec::push_front()`

Advantage `llvec`: Prepending elements is very easy:

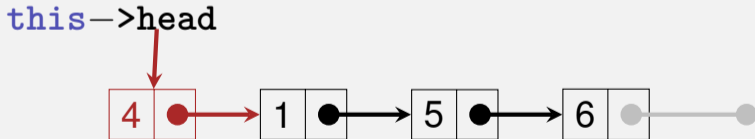
```
void llvec::push_front(int e) {
 this->head =
 new llnode{e, this->head};
}
```



# Function `llvec::push_front()`

Advantage `llvec`: Prepending elements is very easy:

```
void llvec::push_front(int e) {
 this->head =
 new llnode{e, this->head};
}
```



Attention: If the new `llnode` weren't allocated *dynamically*, then it would be deleted (= memory deallocated) as soon as `push_front` terminates



# Function `llvec::llvec()`

Constructor can be implemented using `push_front()`:

```
llvec::llvec(unsigned int size) {
 1this->head = nullptr; ← head initially points to nowhere

 2for (; 0 < size; --size)
 2this->push_front(0);
}
```

# Function `llvec::llvec()`

Constructor can be implemented using `push_front()`:

```
llvec::llvec(unsigned int size) {
 1this->head = nullptr;

 2for (; 0 < size; --size)
 2this->push_front(0);
}
```

← Prepend 0 size times

# Function `llvec::llvec()`

Constructor can be implemented using `push_front()`:

```
llvec::llvec(unsigned int size) {
 1this->head = nullptr;



 2for (; 0 < size; --size)
 2this->push_front(0);
}
```

Use case:

```
llvec v = llvec(3);
std::cout << v; // 0 0 0
```

# Function `llvec::push_back()`

Simple, but inefficient: traverse linked list to its end and append new element

```
void llvec::push_back(int e) {
 1llnode* n = this->head;   Start at first element ...

 2for (; n->next != nullptr; n = n->next);

 3n->next =
 3new llnode{e, nullptr};
}
```

# Function `llvec::push_back()`

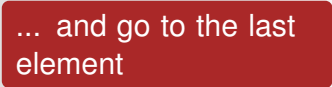
Simple, but inefficient: traverse linked list to its end and append new element

```
void llvec::push_back(int e) {
 1llnode* n = this->head;

 2for (; n->next != nullptr; n = n->next);

 3n->next =
 3new llnode{e, nullptr};
}
```

... and go to the last element



# Function `llvec::push_back()`

Simple, but inefficient: traverse linked list to its end and append new element

```
void llvec::push_back(int e) {
 llnode* n = this->head;
```

```
 for (; n->next != nullptr; n = n->next);
```

```
 n->next =
 new llnode{e, nullptr};
```

```
}
```

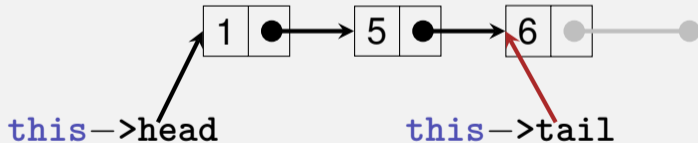


Append new element to currently last

# Function `llvec::push_back()`

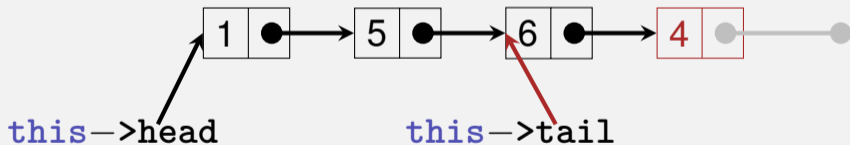
- More efficient, but also slightly more complex:

- 1 Second pointer, pointing to the last element: `this->tail`



# Function `llvec::push_back()`

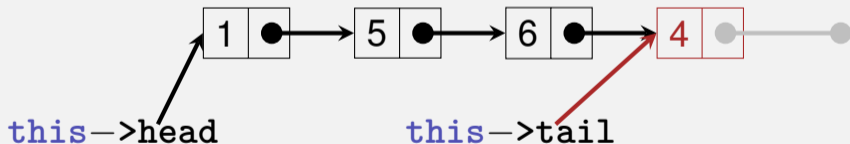
- More efficient, but also slightly more complex:
  - 1 Second pointer, pointing to the last element: `this->tail`
  - 2 Using this pointer, it is possible to append to the end directly





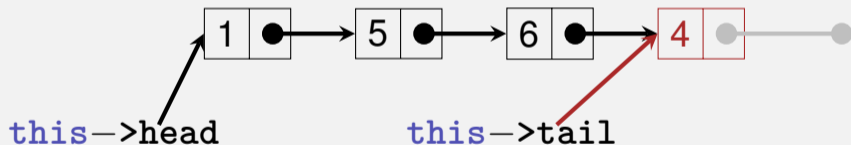
# Function `llvec::push_back()`

- More efficient, but also slightly more complex:
  - 1 Second pointer, pointing to the last element: `this->tail`
  - 2 Using this pointer, it is possible to append to the end directly



# Function `llvec::push_back()`


- More efficient, but also slightly more complex:
  - 1 Second pointer, pointing to the last element: `this->tail`
  - 2 Using this pointer, it is possible to append to the end directly



- **But:** Several corner cases, e.g. vector still empty, must be accounted for

# Function `llvec::size()`

Simple, but inefficient: *compute* size by counting

```
unsigned int llvec::size() const {
 unsigned int c = 0; 

 for (llnode* n = this->head;
 n != nullptr;
 n = n->next)
 ++c;

 return c;
}
```

# Function `llvec::size()`

Simple, but inefficient: *compute* size by counting

```
unsigned int llvec::size() const {
 unsigned int c = 0;

 for (llnode* n = this->head;
 n != nullptr;
 n = n->next)
 ++c;

 return c;
}
```




Count linked-list length

# Function `llvec::size()`

Simple, but inefficient: *compute* size by counting

```
unsigned int llvec::size() const {
 unsigned int c = 0;

 for (llnode* n = this->head;
 n != nullptr;
 n = n->next)
 ++c;

 return c; 
```

Return count

# Function `llvec::size()`

More efficient, but also slightly more complex: *maintain* size as member variable

- 1 Add member variable `unsigned int` `count` to class `llvec`

# Function `llvec::size()`

More efficient, but also slightly more complex: *maintain* size as member variable

- 1 Add member variable `unsigned int` `count` to class `llvec`
- 2 `this->count` must now be updated *each* time an operation (such as `push_front`) affects the vector's size

# Efficiency: Arrays vs. Linked Lists

- Memory: our `avec` requires roughly  $n$  ints (vector size  $n$ ), our `llvec` roughly  $3n$  ints (a pointer typically requires 8 byte)



# Efficiency: Arrays vs. Linked Lists

- Memory: our `avec` requires roughly  $n$  ints (vector size  $n$ ), our `llvec` roughly  $3n$  ints (a pointer typically requires 8 byte)
- Runtime (with `avec = std::vector`, `llvec = std::list`):

```
prepending (insert at front) [100,000x]:
 ▶ avec: 675 ms
 ▶ llvec: 10 ms
appending (insert at back) [100,000x]:
 ▶ avec: 2 ms
 ▶ llvec: 9 ms
removing first [100,000x]:
 ▶ avec: 675 ms
 ▶ llvec: 4 ms
removing last [100,000x]:
 ▶ avec: 0 ms
 ▶ llvec: 4 ms

removing randomly [10,000x]:
 ▶ avec: 3 ms
 ▶ llvec: 113 ms
inserting randomly [10,000x]:
 ▶ avec: 16 ms
 ▶ llvec: 117 ms
fully iterate sequentially (5000 elements) [5,000x]:
 ▶ avec: 354 ms
 ▶ llvec: 525 ms
```

# 20. Containers, Iterators and Algorithms

Containers, Sets, Iterators, const-Iterators, Algorithms, Templates

# Vectors are Containers

- Viewed abstractly, a vector is
  - 1 A collection of elements
  - 2 Plus operations on this collection

# Vectors are Containers

- Viewed abstractly, a vector is
  - 1 A collection of elements
  - 2 Plus operations on this collection
- In C++, `vector<T>` and similar data structures are called *container*

# Vectors are Containers

- Viewed abstractly, a vector is
  - 1 A collection of elements
  - 2 Plus operations on this collection
- In C++, `vector<T>` and similar data structures are called *container*
- Called *collections* in some other languages, e.g. Java

# Container properties

- Each container has certain *characteristic properties*
- For an array-based vector, these include:

# Container properties

- Each container has certain *characteristic properties*
- For an array-based vector, these include:
  - Efficient index-based access ( $v[i]$ )
  - Efficient use of memory: Only the elements themselves require space (plus element count)

# Container properties

- Each container has certain *characteristic properties*
- For an array-based vector, these include:
  - Efficient index-based access ( $v[i]$ )
  - Efficient use of memory: Only the elements themselves require space (plus element count)
  - Inserting at/removing from arbitrary index is potentially inefficient
  - Looking for a specific element is potentially inefficient



# Container properties

- Each container has certain *characteristic properties*
- For an array-based vector, these include:
  - Efficient index-based access ( $v[i]$ )
  - Efficient use of memory: Only the elements themselves require space (plus element count)
  - Inserting at/removing from arbitrary index is potentially inefficient
  - Looking for a specific element is potentially inefficient
  - Can contain the same element more than once
  - Elements are in insertion order (ordered but not sorted)

# Containers in C++

- Nearly every application requires maintaining and manipulating arbitrarily many data records

# Containers in C++

- Nearly every application requires maintaining and manipulating arbitrarily many data records
- But with different requirements (e.g. only append elements, hardly ever remove, often search elements, . . .)

# Containers in C++

- Nearly every application requires maintaining and manipulating arbitrarily many data records
- But with different requirements (e.g. only append elements, hardly ever remove, often search elements, ...)
- That's why C++'s standard library includes several containers with different properties, see <https://en.cppreference.com/w/cpp/container>

# Containers in C++

- Nearly every application requires maintaining and manipulating arbitrarily many data records
- But with different requirements (e.g. only append elements, hardly ever remove, often search elements, ...)
- That's why C++'s standard library includes several containers with different properties, see <https://en.cppreference.com/w/cpp/container>
- Many more are available from 3rd-party libraries, e.g. [https://www.boost.org/doc/libs/1\\_68\\_0/doc/html/container.html](https://www.boost.org/doc/libs/1_68_0/doc/html/container.html), <https://github.com/abseil/abseil-cpp>

## Example Container: `std::unordered_set<T>`

- A *mathematical set* is an unordered, duplicate-free collection of elements:

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- In C++: `std::unordered_set<T>`

# Example Container: `std::unordered_set<T>`

- A *mathematical set* is an unordered, duplicate-free collection of elements:

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- In C++: `std::unordered_set<T>`
- Properties:
  - Cannot contain the same element twice
  - Elements are not in any particular order

# Example Container: `std::unordered_set<T>`

- A *mathematical set* is an unordered, duplicate-free collection of elements:

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- In C++: `std::unordered_set<T>`
- Properties:
  - Cannot contain the same element twice
  - Elements are not in any particular order
  - Does not provide index-based access (`s[i]` undefined)



# Example Container: `std::unordered_set<T>`

- A *mathematical set* is an unordered, duplicate-free collection of elements:

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- In C++: `std::unordered_set<T>`
- Properties:
  - Cannot contain the same element twice
  - Elements are not in any particular order
  - Does not provide index-based access (`s[i]` undefined)
  - Efficient “element contained?” check
  - Efficient insertion and removal of elements

# Example Container: `std::unordered_set<T>`

- A *mathematical set* is an unordered, duplicate-free collection of elements:

$$\{1, 2, 1\} = \{1, 2\} = \{2, 1\}$$

- In C++: `std::unordered_set<T>`
- Properties:
  - Cannot contain the same element twice
  - Elements are not in any particular order
  - Does not provide index-based access (`s[i]` undefined)
  - Efficient “element contained?” check
  - Efficient insertion and removal of elements
- Side remark: implemented as a hash table

# Use Case `std::unordered_set<T>`

Problem:

- given a sequence of pairs (*name*, *percentage*) of Code Expert submissions ...

```
// Input: file submissions.txt
```

```
Friedrich 90
```

```
Schwerhoff 10
```

```
Lehner 20
```

```
Schwerhoff 11
```

# Use Case `std::unordered_set<T>`

Problem:

- given a sequence of pairs (*name*, *percentage*) of Code Expert submissions ...

```
// Input: file submissions.txt
Friedrich 90
Schwerhoff 10
Lehner 20
Schwerhoff 11
```

- ... determine the submitters that achieved at least 50%

```
// Output
Friedrich
```

## Use Case `std::unordered_set<T>`

```
1std::ifstream in("submissions.txt"); ← Open submissions.t
2std::unordered_set<std::string> names;

3std::string name;
3unsigned int score;

while (4in >> name >> score) {
 5if (50 <= score)
 5names.insert(name);
}

6std::cout << "Unique submitters: "
 6<< names << '\n';
```

## Use Case `std::unordered_set<T>`

```
1std::ifstream in("submissions.txt");
```

```
2std::unordered_set<std::string> names; ← Set of names, initially
```

```
3std::string name;
```

```
3unsigned int score;
```

```
while (4in >> name >> score) {
```

```
5if (50 <= score)
```

```
6names.insert(name);
```

```
}
```

```
6std::cout << "Unique submitters: "
```

```
6<< names << '\n';
```

# Use Case `std::unordered_set<T>`

```
1std::ifstream in("submissions.txt");
2std::unordered_set<std::string> names;
```

```
3std::string name;
3unsigned int score;
```



Pair (*name, score*)


```
while (4in >> name >> score) {
 5if (50 <= score)
 5names.insert(name);
}
```

```
6std::cout << "Unique submitters: "
 6<< names << '\n';
```

## Use Case `std::unordered_set<T>`

```
1std::ifstream in("submissions.txt");
2std::unordered_set<std::string> names;
```

```
3std::string name;
3unsigned int score;
```

```
while (4in >> name >> score) { ←  Input next pair
 5if (50 <= score)
 5names.insert(name);
}
```

```
6std::cout << "Unique submitters: "
 6<< names << '\n';
```



## Use Case `std::unordered_set<T>`

```
1std::ifstream in("submissions.txt");
2std::unordered_set<std::string> names;
```

```
3std::string name;
3unsigned int score;
```

```
while (4in >> name >> score) {
 5if (50 <= score)
 5names.insert(name);
}
```

Record name if score  
fices

```
6std::cout << "Unique submitters: "
 6<< names << '\n';
```

## Use Case `std::unordered_set<T>`

```
1std::ifstream in("submissions.txt");
2std::unordered_set<std::string> names;

3std::string name;
3unsigned int score;

while (4in >> name >> score) {
 5if (50 <= score)
 5names.insert(name);
}

6std::cout << "Unique submitters: "
 6<< names << '\n';
```

← Output recorded name

## Example Container: `std::set<T>`

- Nearly equivalent to `std::unordered_set<T>`, but the elements are *ordered*

$$\{1, 2, 1\} = \{1, 2\} \neq \{2, 1\}$$

## Example Container: `std::set<T>`

- Nearly equivalent to `std::unordered_set<T>`, but the elements are *ordered*

$$\{1, 2, 1\} = \{1, 2\} \neq \{2, 1\}$$

- Element look-up, insertion and removal are still efficient (better than for `std::vector<T>`), but less efficient than for `std::unordered_set<T>`

## Example Container: `std::set<T>`

- Nearly equivalent to `std::unordered_set<T>`, but the elements are *ordered*

$$\{1, 2, 1\} = \{1, 2\} \neq \{2, 1\}$$

- Element look-up, insertion and removal are still efficient (better than for `std::vector<T>`), but less efficient than for `std::unordered_set<T>`
- That's because maintaining the order does not come for free

## Example Container: `std::set<T>`

- Nearly equivalent to `std::unordered_set<T>`, but the elements are *ordered*

$$\{1, 2, 1\} = \{1, 2\} \neq \{2, 1\}$$

- Element look-up, insertion and removal are still efficient (better than for `std::vector<T>`), but less efficient than for `std::unordered_set<T>`
- That's because maintaining the order does not come for free
- Side remark: implemented as a red-black tree

## Use Case `std::set<T>`

```
std::ifstream in("submissions.txt");
```

```
1std::set<std::string> names; ←
```

set instead of `unordered_set`

```
std::string name;
```

```
unsigned int score;
```

```
while (in >> name >> score) {
```

```
 if (50 <= score)
```

```
 names.insert(name);
```

```
}
```

```
2std::cout << "Unique submitters: "
```

```
 ? << names << '\n';
```

## Use Case `std::set<T>`

```
std::ifstream in("submissions.txt");
1std::set<std::string> names;
```

```
std::string name;
unsigned int score;
```

```
while (in >> name >> score) {
 if (50 <= score)
 names.insert(name);
}
```

```
2std::cout << "Unique submitters: "
 << names << '\n';
```

← ... and the output is in alphabetical order



# Printing Containers

- Recall: `avec::print()` and `l1vec::print()`

# Printing Containers

- Recall: `avec::print()` and `l1vec::print()`
- What about printing `set`, `unordered_set`, ...?

# Printing Containers

- Recall: `avec::print()` and `l1vec::print()`
- What about printing `set`, `unordered_set`, ...?
- Commonality: iterate over container elements and print them

# Similar Functions

- Lots of other useful operations can be implemented by iterating over a container:
- `contains(c, e)`: true iff container `c` contains element `e`

# Similar Functions

- Lots of other useful operations can be implemented by iterating over a container:
- `contains(c, e)`: true iff container `c` contains element `e`
- `min/max(c)`: Returns the smallest/largest element

# Similar Functions

- Lots of other useful operations can be implemented by iterating over a container:
- `contains(c, e)`: true iff container `c` contains element `e`
- `min/max(c)`: Returns the smallest/largest element
- `sort(c)`: Sorts `c`'s elements

# Similar Functions

- Lots of other useful operations can be implemented by iterating over a container:
- `contains(c, e)`: true iff container `c` contains element `e`
- `min/max(c)`: Returns the smallest/largest element
- `sort(c)`: Sorts `c`'s elements
- `replace(c, e1, e2)`: Replaces each `e1` in `c` with `e2`

# Similar Functions

- Lots of other useful operations can be implemented by iterating over a container:
- `contains(c, e)`: true iff container `c` contains element `e`
- `min/max(c)`: Returns the smallest/largest element
- `sort(c)`: Sorts `c`'s elements
- `replace(c, e1, e2)`: Replaces each `e1` in `c` with `e2`
- `sample(c, n)`: Randomly chooses `n` elements from `c`



# Similar Functions

- Lots of other useful operations can be implemented by iterating over a container:
- `contains(c, e)`: true iff container `c` contains element `e`
- `min/max(c)`: Returns the smallest/largest element
- `sort(c)`: Sorts `c`'s elements
- `replace(c, e1, e2)`: Replaces each `e1` in `c` with `e2`
- `sample(c, n)`: Randomly chooses `n` elements from `c`
- ...

# Recall: Iterating With Pointers

- Iteration over an *array*:



# Recall: Iterating With Pointers

- Iteration over an *array*:
  - Point to start element: `p = this->arr`



# Recall: Iterating With Pointers

- Iteration over an *array*:
  - Point to start element: `p = this->arr`
  - Access current element: `*p`



# Recall: Iterating With Pointers

- Iteration over an *array*:
  - Point to start element: `p = this->arr`
  - Access current element: `*p`
  - Check if end reached: `p == p + size`



# Recall: Iterating With Pointers

- Iteration over an *array*:
  - Point to start element: `p = this->arr`
  - Access current element: `*p`
  - Check if end reached: `p == p + size`
  - Advance pointer: `p = p + 1`



# Recall: Iterating With Pointers

- Iteration over an *array*:
  - Point to start element:  $p = \text{this} \rightarrow \text{arr}$
  - Access current element:  $*p$
  - Check if end reached:  $p == p + \text{size}$
  - Advance pointer:  $p = p + 1$
  
- Iteration over a *linked list*:



# Recall: Iterating With Pointers

## ■ Iteration over an *array*:

- Point to start element:  $p = \text{this} \rightarrow \text{arr}$
- Access current element:  $*p$
- Check if end reached:  $p == p + \text{size}$
- Advance pointer:  $p = p + 1$



## ■ Iteration over a *linked list*:

- Point to start element:  $p = \text{this} \rightarrow \text{head}$





# Recall: Iterating With Pointers

## ■ Iteration over an *array*:

- Point to start element:  $p = \text{this} \rightarrow \text{arr}$
- Access current element:  $*p$
- Check if end reached:  $p == p + \text{size}$
- Advance pointer:  $p = p + 1$



## ■ Iteration over a *linked list*:

- Point to start element:  $p = \text{this} \rightarrow \text{head}$
- Access current element:  $p \rightarrow \text{value}$



# Recall: Iterating With Pointers

## ■ Iteration over an *array*:

- Point to start element: `p = this->arr`
- Access current element: `*p`
- Check if end reached: `p == p + size`
- Advance pointer: `p = p + 1`



## ■ Iteration over a *linked list*:

- Point to start element: `p = this->head`
- Access current element: `p->value`
- Check if end reached: `p == nullptr`



# Recall: Iterating With Pointers

## ■ Iteration over an *array*:

- Point to start element:  $p = \text{this} \rightarrow \text{arr}$
- Access current element:  $*p$
- Check if end reached:  $p == p + \text{size}$
- Advance pointer:  $p = p + 1$



## ■ Iteration over a *linked list*:

- Point to start element:  $p = \text{this} \rightarrow \text{head}$
- Access current element:  $p \rightarrow \text{value}$
- Check if end reached:  $p == \text{nullptr}$
- Advance pointer:  $p = p \rightarrow \text{next}$



# Iterators

- Iteration requires only the previously shown four operations
- But their implementation depends on the container

# Iterators

- Iteration requires only the previously shown four operations
- But their implementation depends on the container
- $\Rightarrow$  Each C++-container implements their own *iterator*

# Iterators

- Iteration requires only the previously shown four operations
- But their implementation depends on the container
- $\Rightarrow$  Each C++-container implements their own *Iterator*
- Given a container `c`:
  - `it = c.begin()`: Iterator pointing to the first element

# Iterators

- Iteration requires only the previously shown four operations
- But their implementation depends on the container
- $\Rightarrow$  Each C++-container implements their own *Iterator*
- Given a container `c`:
  - `it = c.begin()`: Iterator pointing to the first element
  - `it = c.end()`: Iterator pointing *behind* the last element

# Iterators

- Iteration requires only the previously shown four operations
- But their implementation depends on the container
- $\Rightarrow$  Each C++ container implements their own *Iterator*
- Given a container `c`:
  - `it = c.begin()`: Iterator pointing to the first element
  - `it = c.end()`: Iterator pointing *behind* the last element
  - `*it`: Access current element

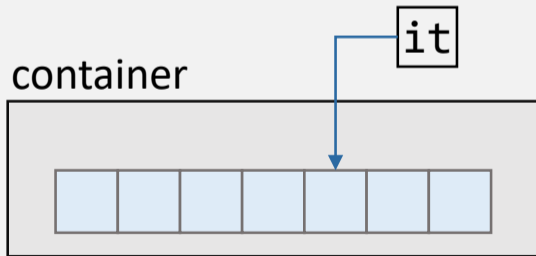


# Iterators

- Iteration requires only the previously shown four operations
- But their implementation depends on the container
- $\Rightarrow$  Each C++ container implements their own *Iterator*
- Given a container `c`:
  - `it = c.begin()`: Iterator pointing to the first element
  - `it = c.end()`: Iterator pointing *behind* the last element
  - `*it`: Access current element
  - `++it`: Advance iterator by one element
- Iterators are essentially pimped pointers

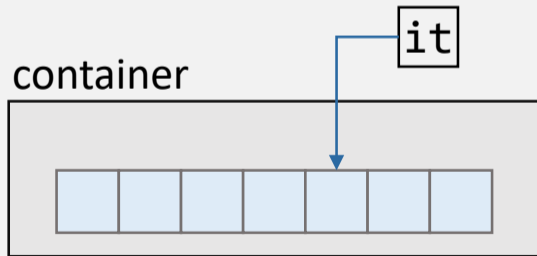
# Iterators

- Iterators allow accessing different containers in a *uniform* way: `*it`, `++it`, etc.



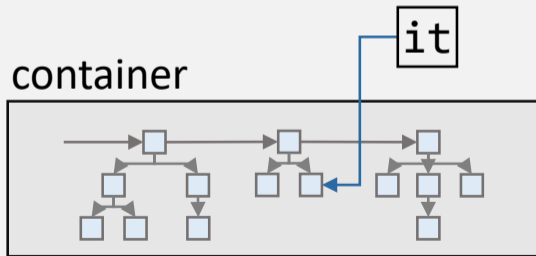
# Iterators

- Iterators allow accessing different containers in a *uniform* way: `*it`, `++it`, etc.
- Users remain independent of the container implementation



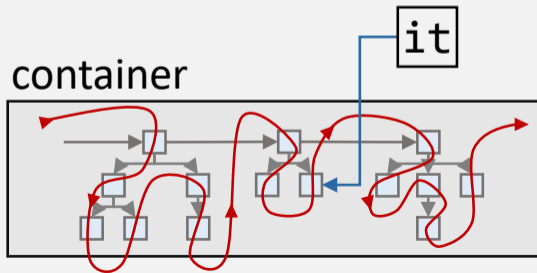
# Iterators

- Iterators allow accessing different containers in a *uniform* way: `*it`, `++it`, etc.
- Users remain independent of the container implementation



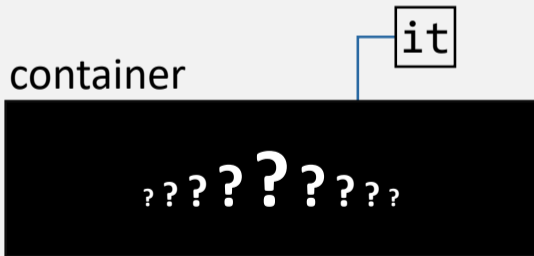
# Iterators

- Iterators allow accessing different containers in a *uniform* way: `*it`, `++it`, etc.
- Users remain independent of the container implementation
- Iterator knows how to iterate over the elements of “its” container



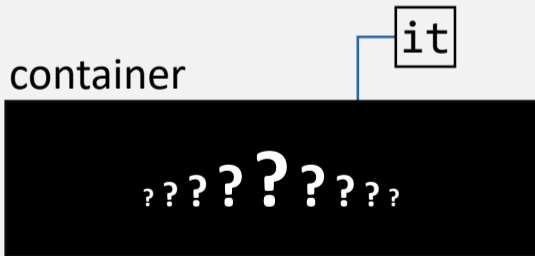
# Iterators

- Iterators allow accessing different containers in a *uniform* way: `*it`, `++it`, etc.
- Users remain independent of the container implementation
- Iterator knows how to iterate over the elements of “its” container
- Users don't need to and also shouldn't know internal details



# Iterators

- Iterators allow accessing different containers in a *uniform* way: `*it`, `++it`, etc.
- Users remain independent of the container implementation
- Iterator knows how to iterate over the elements of “its” container
- Users don't need to and also shouldn't know internal details
- ⇒



# Example: Iterate over `std::vector`

`it` is an iterator specific to `std::vector`

```
std::vector<int> v = {1, 2, 3};

for (std::vector<int>::iterator it = v.begin();
 it != v.end();
 ++it) {

 *it = -*it;
}

std::cout << v; // -1 -2 -3
```



# Example: Iterate over `std::vector`

```
std::vector<int> v = {1, 2, 3};
```

```
for (std::vector<int>::iterator it = v.begin();
 it != v.end();
 ++it) {

 *it = -*it;
}
```

```
std::cout << v; // -1 -2 -3
```

it initially points to the first element



# Example: Iterate over `std::vector`

```
std::vector<int> v = {1, 2, 3};

for (std::vector<int>::iterator it = v.begin();
 it != v.end(); ++it) {
 *it = -*it;
}

std::cout << v; // -1 -2 -3
```

← Abort if it reached the end

# Example: Iterate over `std::vector`

```
std::vector<int> v = {1, 2, 3};
```

```
for (std::vector<int>::iterator it = v.begin();
 it != v.end();
 ++it) {
```

Advance it element-wise



```
 *it = -*it;
}
```

```
std::cout << v; // -1 -2 -3
```

# Example: Iterate over `std::vector`

```
std::vector<int> v = {1, 2, 3};
```

```
for (std::vector<int>::iterator it = v.begin();
 it != v.end();
 ++it) {
```

```
 *it = -*it;  Negate current element ($e \rightarrow -e$)
}
```

```
std::cout << v; // -1 -2 -3
```

## Example: Iterate over `std::vector`

```
std::vector<int> v = {1, 2, 3};

for (std::vector<int>::iterator it = v.begin();
 it != v.end();
 ++it) {

 *it = -*it;
}

std::cout << v; // -1 -2 -3
```

## Example: Iterate over `std::vector`

Recall: type aliases can be used to shorten often-used type names

```
using ivit = std::vector<int>::iterator; // int-
 vector iterator

for (ivit it = v.begin();
 ...
```

# Negate as a Function

```
void neg(std::vector<int>& v) {
 for (std::vector<int>::iterator it = v.begin();
 it != v.end();
 ++it) {

 *it = -*it;
 }
}

// in main():
std::vector<int> v = {1, 2, 3};
neg(v); // v = {-1, -2, -3}
```

# Negate as a Function

```
void neg(std::vector<int>& v) {
 for (std::vector<int>::iterator it = v.begin();
 it != v.end();
 ++it) {

 *it = -*it;
 }
}

// in main():
std::vector<int> v = {1, 2, 3};
neg(v); // v = {-1, -2, -3}
```



# Negate as a Function

*Better: negate inside a specific range (interval)*

```
void neg(1std::vector<int>::iterator begin;
 1std::vector<int>::iterator end) {

 for (std::vector<int>::iterator it = 1begin;
 it != 1end;
 ++it) {

 *it = -*it;
 }
}
```

Negate elem  
interval [beg

# Negate as a Function

*Better: negate inside a specific range (interval)*

```
void neg(std::vector<int>::iterator start;
 std::vector<int>::iterator end);
```

```
// in main():
```

```
std::vector<int> v = {1, 2, 3};
```

```
1neg(v.begin(), v.begin() + (v.size() / 2)); ← Negate first half
```

# Algorithms Library in C++

- The C++ standard library includes lots of useful algorithms (functions) that work on iterator-defined intervals [*begin*, *end*)

# Algorithms Library in C++

- The C++ standard library includes lots of useful algorithms (functions) that work on iterator-defined intervals [*begin*, *end*)
- For example `find`, `fill` and `sort`

# Algorithms Library in C++

- The C++ standard library includes lots of useful algorithms (functions) that work on iterator-defined intervals [*begin*, *end*)
- For example `find`, `fill` and `sort`
- See also <https://en.cppreference.com/w/cpp/algorithm>

# An iterator for `l1vec`

We need:

- 1 An `l1vec`-specific iterator with at least the following functionality:
  - Access current element: `operator*`
  - Advance iterator: `operator++`
  - End-reached check: `operator!=` (or `operator==`)

# An iterator for `llvec`

We need:

- 1 An `llvec`-specific iterator with at least the following functionality:
  - Access current element: `operator*`
  - Advance iterator: `operator++`
  - End-reached check: `operator!=` (or `operator==`)
- 2 Member functions `begin()` and `end()` for `llvec` to get an iterator to the beginning and past the end, respectively

## Iterator avec ::iterator (Step 1/2)

```
1class l1vec {
 ...
public:
 1class iterator {
 1 ...
 1};

 ...
}
```

- The iterator belongs to our vector, that's why `iterator` is a public *inner class* of `l1vec`



# Iterator avec ::iterator (Step 1/2)

```
1class l1vec {
 ...
public:
 1class iterator {
 1 ...
 1};

 ...
}
```

- The iterator belongs to our vector, that's why `iterator` is a public *inner class* of `l1vec`
- Instances of our iterator are of type `l1vec::iterator`

# Iterator `llvec::iterator` (Step 1/2)

```
class iterator {
 1llnode* node; ← Pointer to current vector element

public:
 2iterator(llnode* n);
 3iterator& operator++();
 4int& operator*() const;
 5bool operator!=(const iterator& other) const;
};
```

# Iterator `llvec::iterator` (Step 1/2)

```
class iterator {
 1llnode* node;

public:
 2iterator(llnode* n); ← Create iterator to specific element
 3iterator& operator++();
 4int& operator*() const;
 5bool operator!=(const iterator& other) const;
};
```

## Iterator `llvec::iterator` (Step 1/2)

```
class iterator {
 1llnode* node;

public:
 2iterator(llnode* n);
 3iterator& operator++(); ← Advance iterator by one element
 4int& operator*() const;
 5bool operator!=(const iterator& other) const;
};
```

## Iterator `llvec::iterator` (Step 1/2)

```
class iterator {
 llnode* node;

public:
 2iterator(llnode* n);
 3iterator& operator++();
 4int& operator*() const; ← Access current element
 5bool operator!=(const iterator& other) const;
};
```

## Iterator `llvec::iterator` (Step 1/2)

```
class iterator {
 1llnode* node;

public:
 2iterator(llnode* n);
 3iterator& operator++();
 4int& operator*() const;
 5bool operator!=(const iterator& other) const;
};
```

Compare with other iterator



## Iterator `llvec::iterator` (Step 1/2)

```
// Constructor
llvec::iterator::iterator(llnode* n): 2node(n) {}

// Pre-increment
llvec::iterator& llvec::iterator::operator++() {
 assert(this->node != nullptr);

 4this->node = this->node->next;

 5return *this;
}
```

# Iterator `llvec::iterator` (Step 1/2)

```
// Constructor
```

```
llvec::iterator::iterator(llnode* n): 2node(n) ← {}
```

Let iterator point to `n` initially

```
// Pre-increment
```

```
llvec::iterator& llvec::iterator::operator++() {
```

```
 assert(this->node != nullptr);
```

```
 4this->node = this->node->next;
```

```
 5return *this;
```

```
}
```



## Iterator `llvec::iterator` (Step 1/2)

```
// Constructor
llvec::iterator::iterator(llnode* n): 2node(n) {}

// Pre-increment
llvec::iterator& llvec::iterator::operator++() {
 assert(this->node != nullptr);

 4this->node = this->node->next;

 5return *this;
}
```

## Iterator `llvec::iterator` (Step 1/2)

```
// Constructor
```

```
llvec::iterator::iterator(llnode* n): 2node(n) {}
```

```
// Pre-increment
```

```
llvec::iterator& llvec::iterator::operator++() {
 assert(this->node != nullptr);
```

```
 4this->node = this->node->next; ← Advance iterator by one element
```

```
 5return *this;
```

```
}
```

# Iterator `llvec::iterator` (Step 1/2)

```
// Constructor
llvec::iterator::iterator(llnode* n): 2node(n) {}

// Pre-increment
llvec::iterator& llvec::iterator::operator++() {
 assert(this->node != nullptr);

 4this->node = this->node->next;

 5return *this; ← Return reference to advanced iterator
}
```

## Iterator `llvec::iterator` (Step 1/2)

```
// Element access
int& llvec::iterator::operator*() const {
 2return this->node->value;
}

// Comparison
bool llvec::iterator::operator!=(const llvec::
 iterator& other) const {
 4return this->node != other.node;
}
```

## Iterator `llvec::iterator` (Step 1/2)

```
// Element access
```

```
int& llvec::iterator::operator*() const {
```

```
 2return this->node->value; ← Access current element
```

```
}
```

```
// Comparison
```

```
bool llvec::iterator::operator!=(const llvec::
 iterator& other) const {
```

```
 4return this->node != other.node;
```

```
}
```

## Iterator `llvec::iterator` (Step 1/2)

```
// Element access
int& llvec::iterator::operator*() const {
 2return this->node->value;
}

// Comparison
bool llvec::iterator::operator!=(const llvec::
 iterator& other) const {
 4return this->node != other.node;
}
```

## Iterator `llvec::iterator` (Step 1/2)

```
// Element access
int& llvec::iterator::operator*() const {
 2return this->node->value;
}

// Comparison
bool llvec::iterator::operator!=(const llvec::
 iterator& other) const {
 4return this->node != other.node;
}
```

this iterator different from other if they  
point to different element

# An iterator for `l1vec` (Repetition)

We need:

1 An `l1vec`-specific iterator with at least the following functionality:

- Access current element: `operator*`
- Advance iterator: `operator++`
- End-reached check: `operator!=` (or `operator==`)



2 Member functions `begin()` and `end()` for `l1vec` to get an iterator to the beginning and past the end, respectively



## Iterator avec ::iterator (Step 2/2)

```
1class l1vec {
 ...
public:
 class iterator {...};

 iterator begin();
 iterator end();

 ...
}
```

`l1vec` needs member functions to issue iterators pointing *to the beginning* and *past the end*, respectively, of the vector

## Iterator `llvec::iterator` (Step 2/2)

```
llvec::iterator llvec::begin() {
 1return llvec::iterator(this->head);
}
```



Iterator to first vector element

```
llvec::iterator llvec::end() {
 2return llvec::iterator(nullptr);
}
```

## Iterator `llvec::iterator` (Step 2/2)

```
llvec::iterator llvec::begin() {
 1return llvec::iterator(this->head);
}
```

```
llvec::iterator llvec::end() {
 2return llvec::iterator(nullptr);
}
```



Iterator past last vector element

# Const-Iterators

- In addition to `iterator`, every container should also provide a *const-iterator* `const_iterator`
- Const-iterators grant only read access to the underlying Container
- For example for `llvec`:

```
llvec::1-const_iterator llvec::1-cbegin() const;
llvec::1-const_iterator llvec::1-cend() const;

1-const int& llvec::const_iterator::operator*()
 const;
...
```

# Const-Iterators

- In addition to `iterator`, every container should also provide a *const-iterator* `const_iterator`
- Const-iterators grant only read access to the underlying Container
- For example for `llvec`:

```
llvec::1-const_iterator llvec::1-cbegin() const;
llvec::1-const_iterator llvec::1-cend() const;

1-const int& llvec::const_iterator::operator*()
 const;
...
```

- Therefore not possible (compiler error): `*(v.cbegin()) = 0`

# Const-Iterators

Const-Iterator *can* be used to allow only reading:

```
llvec v = ...;
for (llvec::const_iterator it = v.cbegin(); ...)
 std::cout << *it;
```

It would also be possible to use the non-const `iterator` here

# Const-Iterators

Const-Iterator *must* be used if the vector is const:

```
1const l1vec v = ...;
for (l1vec::1const_iterator it = 1v.cbegin(); ...)
 std::cout << *it;
```

It is not possible to use `iterator` here (compiler error)

# Excursion: Templates

- **Goal:** A *generic* output operator `<<` for *iterable Containers*: `llvec`, `avec`, `std::vector`, `std::set`, ...



# Excursion: Templates

- **Goal:** A *generic* output operator `<<` for *iterable Containers*: `llvec`, `avec`, `std::vector`, `std::set`, ...
- I.e. `std::cout << c << '\n'` should work for any such container `c`

# Excursion: Templates

*Templates* enable *type-generic* functions and classes:

- Templates enable the use of *types as arguments*

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container);
```

# Excursion: Templates

*Templates* enable *type-generic* functions and classes:

- Templates enable the use of *types as arguments*

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container);
```

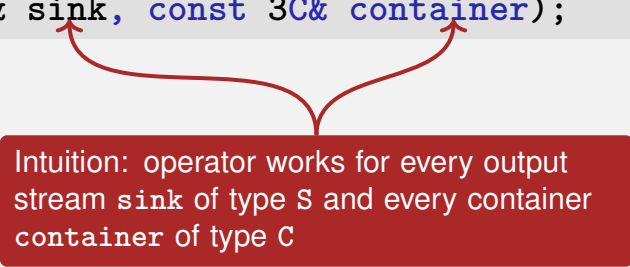
We already know the pointy brackets from vectors. Vectors are also implemented as templates.

# Excursion: Templates

*Templates* enable *type-generic* functions and classes:

- Templates enable the use of *types as arguments*

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container);
```



Intuition: operator works for every output stream sink of type S and every container container of type C

# Excursion: Templates

*Templates* enable *type-generic* functions and classes:

- Templates enable the use of *types as arguments*

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container);
```

- The compiler *infers* suitable types from the call arguments

```
std::set<int> s = ...;
std::cout << s << '\n';
```

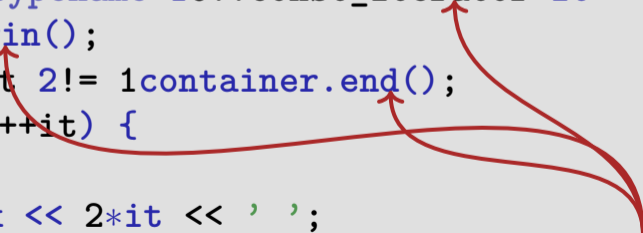
← S = std::ostream, C = std::set<int>

# Excursion: Templates

Implementation of `<< constrains S and C` (Compiler errors if not satisfied):

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container) {
 for (typename C::const_iterator it = container.
 begin();
 it != container.end();
 ++it) {

 sink << *it << ' ';
 }
}
```

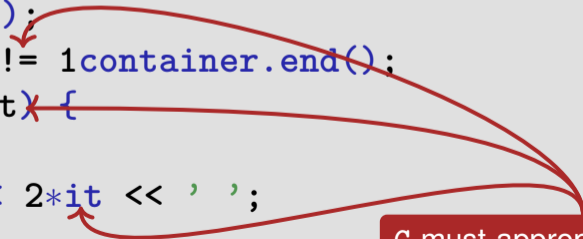


C must appropriate iterators

# Excursion: Templates

Implementation of `<< constrains S and C` (Compiler errors if not satisfied):

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container) {
 for (typename C::const_iterator it = container.
 begin();
 it != container.end();
 ++it) {
 sink << *it << ' ';
 }
}
```



C must appropriate iterators  
– with appropriate functions

# Excursion: Templates

Implementation of `<< constrains S and C` (Compiler errors if not satisfied):

```
template <typename S, typename C>
S& operator<<(S& sink, const C& container) {
 for (typename C::const_iterator it = container.
 begin();
 it != container.end();
 ++it) {
 1sink << *it << ' ' ; ←
 }
}
```

S must support outputting elements (\*it) and characters ( ' ' )



# **21. Dynamic Datatypes and Memory Management**

# Problem

Last week: dynamic data type

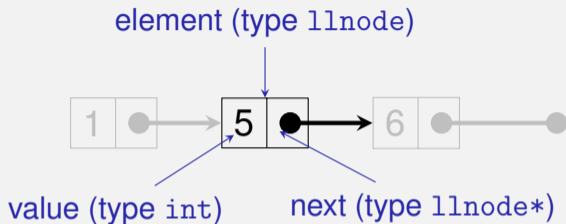
Have allocated dynamic memory, but not released it again. In particular: no functions to remove elements from `llvec`.

Today: correct memory management!

# Goal: class stack with memory management

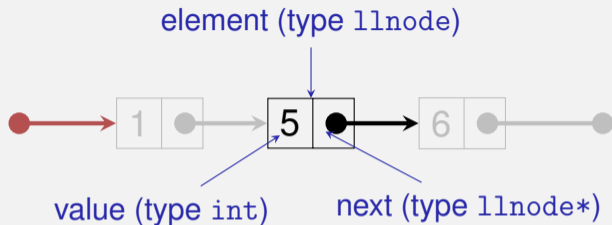
```
class stack{
public:
 // post: Push an element onto the stack
 void push(int value);
 // pre: non-empty stack
 // post: Delete top most element from the stack
 void pop();
 // pre: non-empty stack
 // post: return value of top most element
 int top() const;
 // post: return if stack is empty
 bool empty() const;
 // post: print out the stack
 void print(std::ostream& out) const;
 ...
};
```

# Recall the Linked List



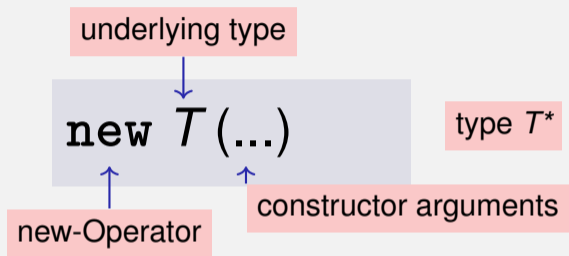
```
struct llnode {
 int value;
 llnode* next;
 // constructor
 llnode (int v, llnode* n) : value (v), next (n) {}
};
```

# Stack = Pointer to the Top Element



```
class stack {
public:
 void push (int value);
 ...
private:
 llnode* topn;
};
```

# Recall 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

# The new Expression

push(4)

```
void stack::push(int value){
 topn = new llnode (value, topn);
}
```

topn



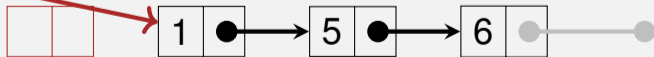
# The new Expression

push(4)

- **Effect:** new object of type  $T$  is allocated in memory ...

```
void stack::push(int value){
 topn = new llnode (value, topn);
}
```

topn





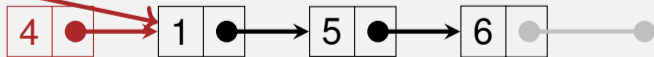
# The new Expression

push(4)

- **Effect:** new object of type  $T$  is allocated in memory ...
- ... and initialized by means of the matching constructor

```
void stack::push(int value){
 topn = new llnode (value, topn);
}
```

topn

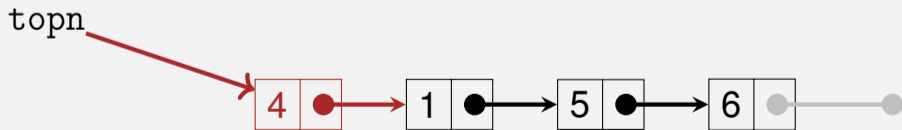


# The new Expression

push(4)

- **Effect:** new object of type  $T$  is allocated in memory ...
- ... and initialized by means of the matching constructor
- **Value:** address of the new object

```
void stack::push(int value){
 topn = new llnode (value, topn);
}
```

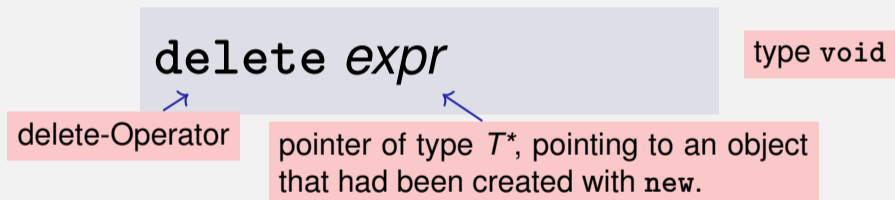


# The delete Expression

Objects generated with `new` have *dynamic storage duration*: they “live” until they are explicitly *deleted*

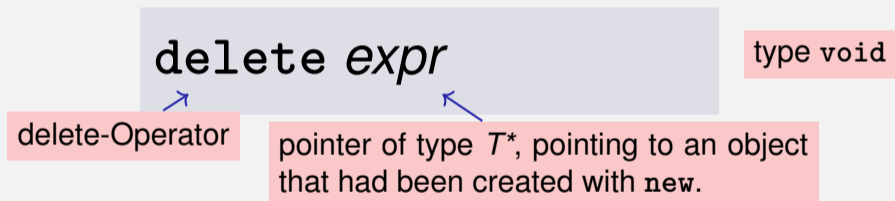
# The delete Expression

Objects generated with `new` have *dynamic storage duration*: they “live” until they are explicitly *deleted*



# The delete Expression

Objects generated with `new` have *dynamic storage duration*: they “live” until they are explicitly *deleted*



- **Effect:** object is *deconstructed* (explanation below)  
... and *memory is released*.

# Who is born must die...

## Guideline “Dynamic Memory”

For each `new` there is a matching `delete`!

# Who is born must die...

## Guideline “Dynamic Memory”

For each `new` there is a matching `delete`!

Non-compliance leads to memory leaks

# 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...



# 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**)

# Careful with new and delete!

```
rational* t = new rational;
rational* s = t;
delete s;
int nominator = (*t).denominator();
```

# Careful with new and delete!

```
rational* t = new rational; ←———— memory for t is allocated
rational* s = t;
delete s;
int nominator = (*t).denominator();
```

# Careful with new and delete!

```
rational* t = new rational; ←————— memory for t is allocated
rational* s = t; ←————— other pointers may point to the same object
delete s;
int nominator = (*t).denominator();
```

# Careful with new and delete!

```
rational* t = new rational; ←———— memory for t is allocated
rational* s = t; ←———— other pointers may point to the same object
delete s; ←———— ... and used for releasing the object
int nominator = (*t).denominator();
```

# Careful with `new` and `delete`!

```
rational* t = new rational; ← memory for t is allocated
rational* s = t; ← other pointers may point to the same object
delete s; ← ... and used for releasing the object
int nominator = (*t).denominator(); ← error: memory released!
```

↑  
Dereferencing of „dangling pointers”

- Pointer to released objects: *dangling pointers*

# Careful with new and delete!

```
rational* t = new rational; ← memory for t is allocated
rational* s = t; ← other pointers may point to the same object
delete s; ← ... and used for releasing the object
int nominator = (*t).denominator(); ← error: memory released!
```

↑  
Dereferencing of „dangling pointers”

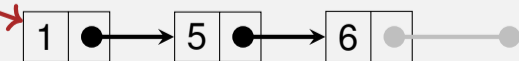
- Pointer to released objects: *dangling pointers*
- Releasing an object more than once using `delete` is a similar severe error

# Stack Continued:

pop()

```
void stack::pop(){
 assert (!empty());
 llnode* p = topn;
 topn = topn->next;
 delete p;
}
```

topn

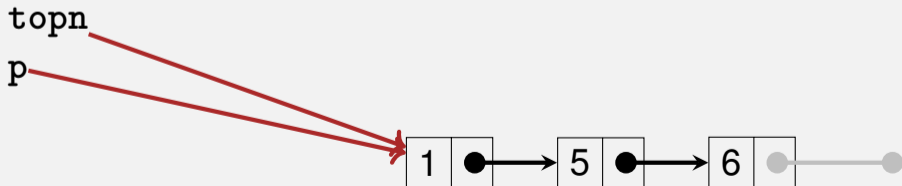




# Stack Continued:

pop()

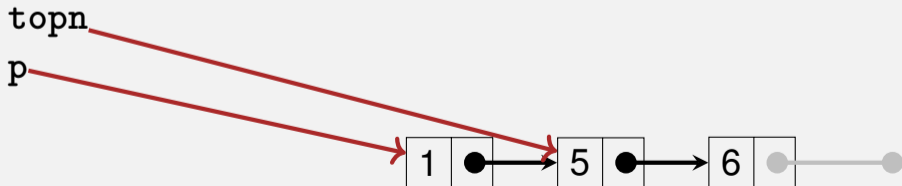
```
void stack::pop(){
 assert (!empty());
 llnode* p = topn;
 topn = topn->next;
 delete p;
}
```



# Stack Continued:

pop()

```
void stack::pop(){
 assert (!empty());
 llnode* p = topn;
 topn = topn->next;
 delete p;
}
```

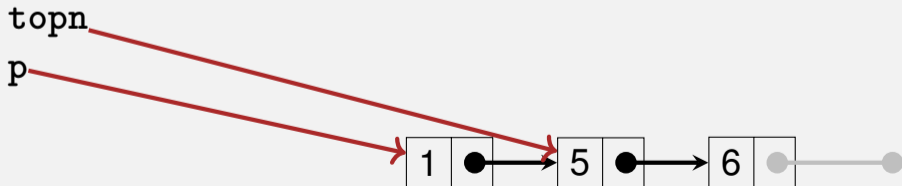


# Stack Continued:

pop()

```
void stack::pop(){
 assert (!empty());
 llnode* p = topn;
 topn = topn->next;
 delete p;
}
```

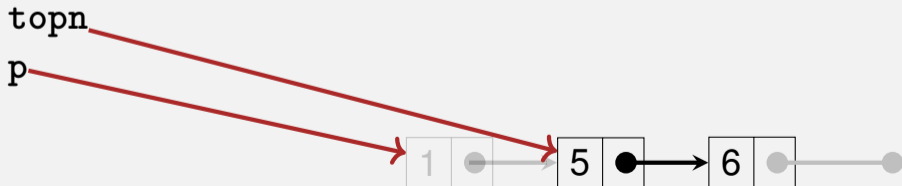
reminder: shortcut for (\*topn).next



# Stack Continued:

pop()

```
void stack::pop(){
 assert (!empty());
 llnode* p = topn;
 topn = topn->next;
 delete p;
}
```

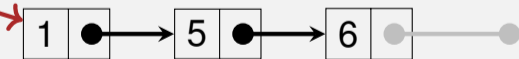


# Print the Stack

print()

```
void stack::print (std::ostream& out) const {
 for(const llnode* p = topn; p != nullptr; p = p->next)
 out << p->value << " ";
}
```

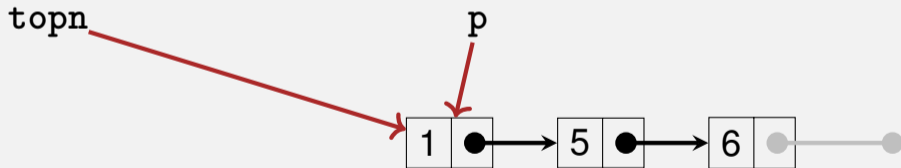
topn



# Print the Stack

print()

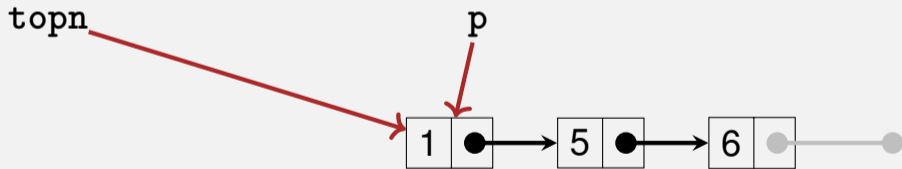
```
void stack::print (std::ostream& out) const {
 for(const llnode* p = topn; p != nullptr; p = p->next)
 out << p->value << " ";
}
```



# Print the Stack

print()

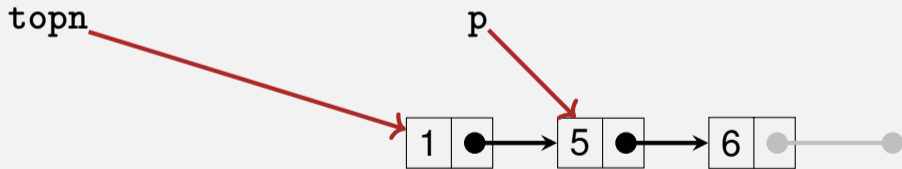
```
void stack::print (std::ostream& out) const {
 for(const llnode* p = topn; p != nullptr; p = p->next)
 out << p->value << " "; // 1
}
```



# Print the Stack

print()

```
void stack::print (std::ostream& out) const {
 for(const llnode* p = topn; p != nullptr; p = p->next)
 out << p->value << " "; // 1
}
```

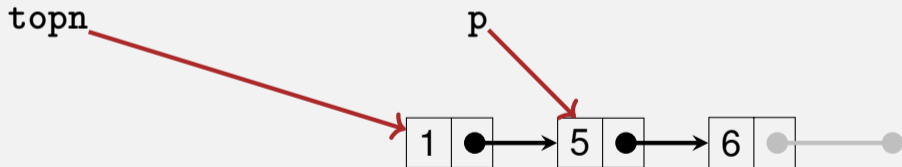




# Print the Stack

print()

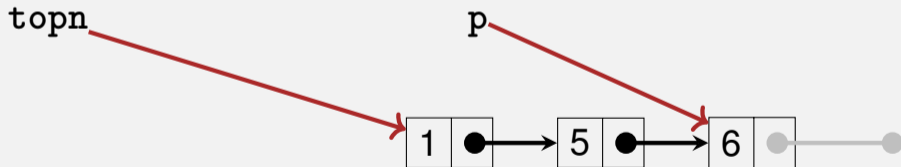
```
void stack::print (std::ostream& out) const {
 for(const llnode* p = topn; p != nullptr; p = p->next)
 out << p->value << " "; // 1 5
}
```



# Print the Stack

print()

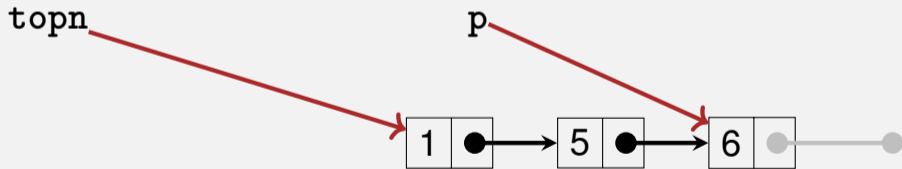
```
void stack::print (std::ostream& out) const {
 for(const llnode* p = topn; p != nullptr; p = p->next)
 out << p->value << " "; // 1 5
}
```



# Print the Stack

print()

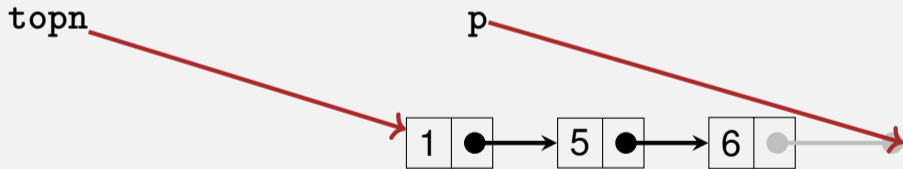
```
void stack::print (std::ostream& out) const {
 for(const llnode* p = topn; p != nullptr; p = p->next)
 out << p->value << " "; // 1 5 6
}
```



# Print the Stack

print()

```
void stack::print (std::ostream& out) const {
 for(const llnode* p = topn; p != nullptr; p = p->next)
 out << p->value << " "; // 1 5 6
}
```



# Output Stack:

operator<<

```
class stack {
public:
 void push (int value);
 void pop();
 void print (std::ostream& o) const;
 ...
private:
 llnode* topn;
};

// POST: s is written to o
std::ostream& operator<< (std::ostream& o, const stack& s){
 s.print (o);
 return o;
}
```

# empty(), top()

```
bool stack::empty() const {
 return top == nullptr;
}
```

```
int stack::top() const {
 assert(!empty());
 return topn->value;
}
```

# Empty Stack

```
class stack{
public:
 stack() : topn (nullptr) {} // default constructor

 void push(int value);
 void pop();
 void print(std::ostream& out) const;
 int top() const;
 bool empty() const;
private:
 llnode* topn;
}
```

# 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)...
```



# 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)!

# 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 – i.e. when `delete` is called on an object of type  $T^*$  or when the enclosing scope of an object of type  $T$  ends.
- If no destructor is declared, it is automatically generated and calls the destructors for the member variables (pointers `topn`, no effect – reason for zombie elements)

# Using a Destructor, it Works

```
// POST: the dynamic memory of *this is deleted
stack::~~stack(){
 while (topn != nullptr){
 llnode* t = topn;
 topn = t->next;
 delete t;
 }
}
```

# Using a Destructor, it Works

```
// POST: the dynamic memory of *this is deleted
stack::~~stack(){
 while (topn != nullptr){
 llnode* t = topn;
 topn = t->next;
 delete t;
 }
}
```

- automatically deletes all stack elements when the stack is being released

# Using a Destructor, it Works

```
// POST: the dynamic memory of *this is deleted
stack::~~stack(){
 while (topn != nullptr){
 llnode* t = topn;
 topn = t->next;
 delete t;
 }
}
```

- automatically deletes all stack elements when the stack is being released
- Now our stack class seems to follow the guideline “dynamic memory” (?)

# Stack Done?

```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n";
```

```
stack s2 = s1;
std::cout << s2 << "\n";
```

```
s1.pop ();
std::cout << s1 << "\n";
```

```
s2.pop ();
```

# Stack Done?

```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n";
```

```
stack s2 = s1;
std::cout << s2 << "\n";
```

```
s1.pop ();
std::cout << s1 << "\n";
```

```
s2.pop ();
```



# Stack Done?

```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n";
```

```
stack s2 = s1;
std::cout << s2 << "\n";
```

```
s1.pop ();
std::cout << s1 << "\n";
```

```
s2.pop ();
```

# Stack Done?

```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n";
```

```
stack s2 = s1;
std::cout << s2 << "\n";
```

```
s1.pop ();
std::cout << s1 << "\n";
```

```
s2.pop ();
```

# Stack Done?

```
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";
```

```
s1.pop ();
std::cout << s1 << "\n";
```

```
s2.pop ();
```

# Stack Done?

```
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";
```

```
s1.pop ();
std::cout << s1 << "\n";
```

```
s2.pop ();
```

# Stack Done?

```
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";
```

```
s2.pop ();
```

# Stack Done?

```
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";
```

```
s2.pop ();
```

# Stack Done?

```
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 ();
```

# Stack Done?

```
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 ();
```



# Stack Done?

```
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!
```

# Stack Done?

Obviously not...

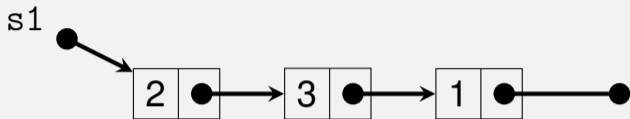
```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n"; // 2 3 1
```

```
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?



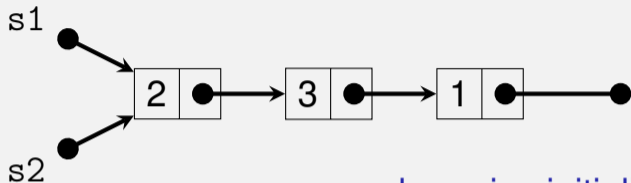
...

```
stack s2 = s1;
std::cout << s2 << "\n";
```

```
s1.pop ();
std::cout << s1 << "\n";
```

```
s2.pop ();
```

# What has gone wrong?



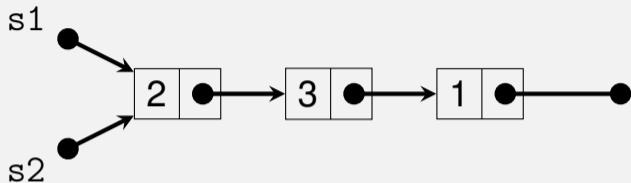
member-wise initialization: copies the  
topn pointer only.

```
...
stack s2 = s1;
std::cout << s2 << "\n";
```

```
s1.pop ();
std::cout << s1 << "\n";
```

```
s2.pop ();
```

# What has gone wrong?



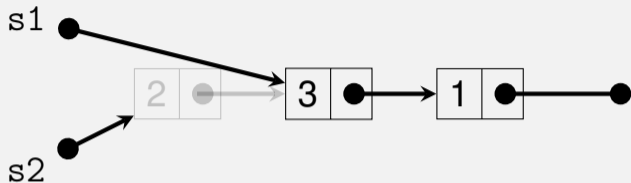
...

```
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
std::cout << s1 << "\n";
```

```
s2.pop ();
```

# What has gone wrong?



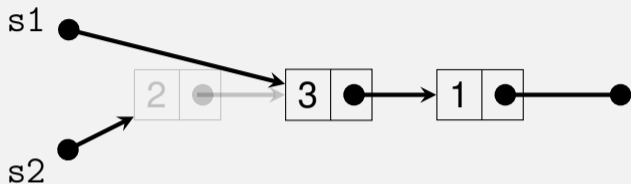
...

```
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
std::cout << s1 << "\n";
```

```
s2.pop ();
```

# What has gone wrong?



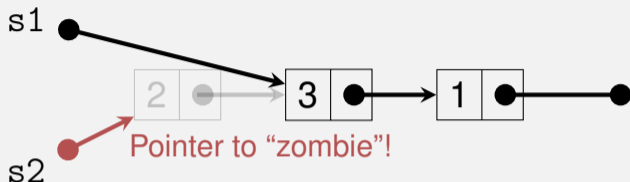
...

```
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```

# What has gone wrong?



...

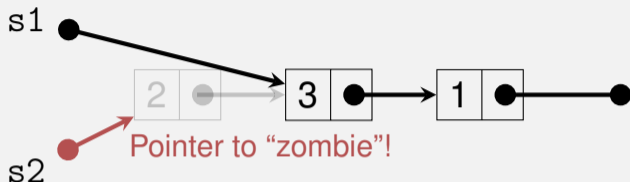
```
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```



# What has gone wrong?



...

```
stack s2 = s1;
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop (); // Oops, crash!
```

# The actual problem

Already this goes wrong:

```
{
 stack s1;
 s1.push(1);
 stack s2 = s1;
}
```

When leaving the scope, both stacks are deconstructed. But both stacks try to delete the same data, because both stacks have *access to the same pointer*.

# Possible solutions

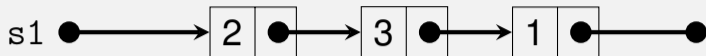
Smart-Pointers (we will not go into details here):

- Count the number of pointers referring to the same objects and delete only when that number goes down to 0  
`std::shared_pointer`
- Make sure that not more than one pointer can point to an object:  
`std::unique_pointer.`

or:

- We make a real copy of all data – as discussed below.

# We make a real copy



...

```
stack s2 = s1;
```

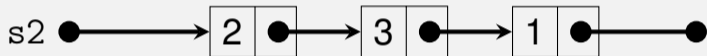
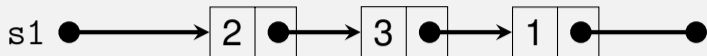
```
std::cout << s2 << "\n";
```

```
s1.pop ();
```

```
std::cout << s1 << "\n";
```

```
s2.pop ();
```

# We make a real copy



...

```
stack s2 = s1;
```

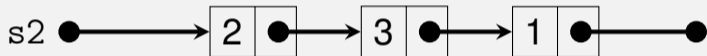
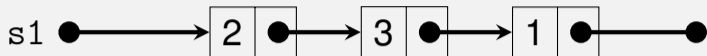
```
std::cout << s2 << "\n";
```

```
s1.pop ();
```

```
std::cout << s1 << "\n";
```

```
s2.pop ();
```

# We make a real copy



...

```
stack s2 = s1;
```

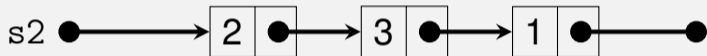
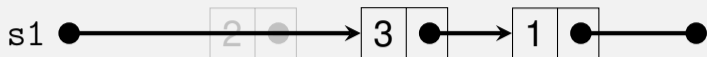
```
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
```

```
std::cout << s1 << "\n";
```

```
s2.pop ();
```

# We make a real copy



...

```
stack s2 = s1;
```

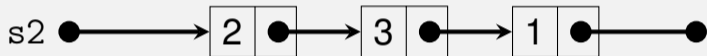
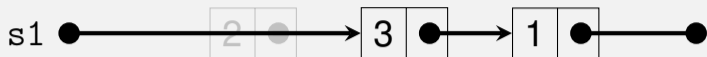
```
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
```

```
std::cout << s1 << "\n";
```

```
s2.pop ();
```

# We make a real copy



...

```
stack s2 = s1;
```

```
std::cout << s2 << "\n"; // 2 3 1
```

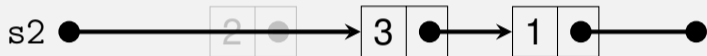
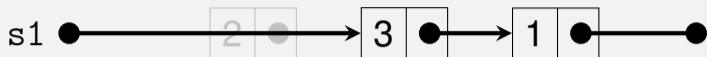
```
s1.pop ();
```

```
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```



# We make a real copy



...

```
stack s2 = s1;
```

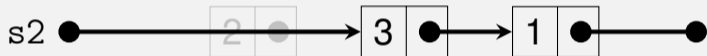
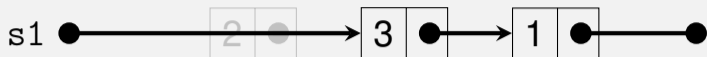
```
std::cout << s2 << "\n"; // 2 3 1
```

```
s1.pop ();
```

```
std::cout << s1 << "\n"; // 3 1
```

```
s2.pop ();
```

# We make 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

$$T(\text{const } T\& x);$$

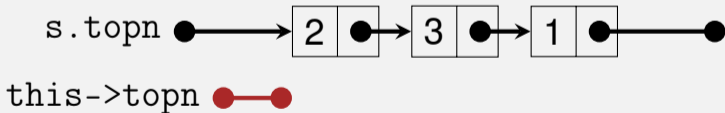
- is automatically called when values of type  $T$  are initialized with values of type  $T$

$$T\ x = t; \quad (\text{t of type } T)$$
$$T\ x(t);$$

- 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

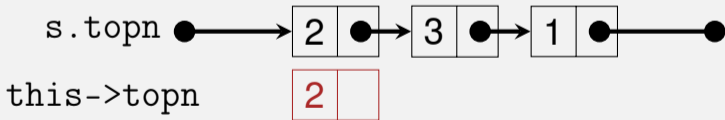
```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
 if (s.topn == nullptr) return;
 topn = new llnode(s.topn->value, nullptr);
 llnode* prev = topn;
 for(llnode* n = s.topn->next; n != nullptr; n = n->next){
 llnode* copy = new llnode(n->value, nullptr);
 prev->next = copy;
 prev = copy;
 }
}
```



prev

# It works with a Copy Constructor

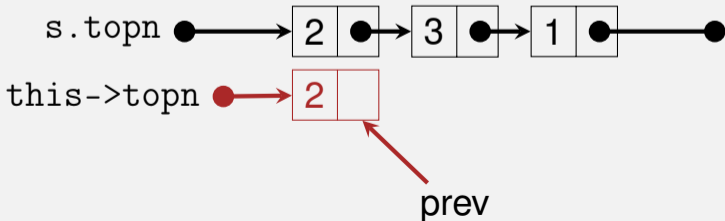
```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
 if (s.topn == nullptr) return;
 topn = new llnode(s.topn->value, nullptr);
 llnode* prev = topn;
 for(llnode* n = s.topn->next; n != nullptr; n = n->next){
 llnode* copy = new llnode(n->value, nullptr);
 prev->next = copy;
 prev = copy;
 }
}
```



prev

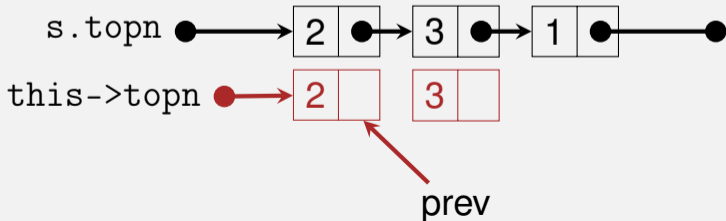
# It works with a Copy Constructor

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
 if (s.topn == nullptr) return;
 topn = new llnode(s.topn->value, nullptr);
 llnode* prev = topn;
 for(llnode* n = s.topn->next; n != nullptr; n = n->next){
 llnode* copy = new llnode(n->value, nullptr);
 prev->next = copy;
 prev = copy;
 }
}
```



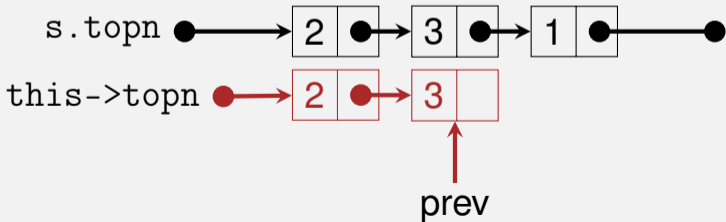
# It works with a Copy Constructor

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
 if (s.topn == nullptr) return;
 topn = new llnode(s.topn->value, nullptr);
 llnode* prev = topn;
 for(llnode* n = s.topn->next; n != nullptr; n = n->next){
 llnode* copy = new llnode(n->value, nullptr);
 prev->next = copy;
 prev = copy;
 }
}
```



# It works with a Copy Constructor

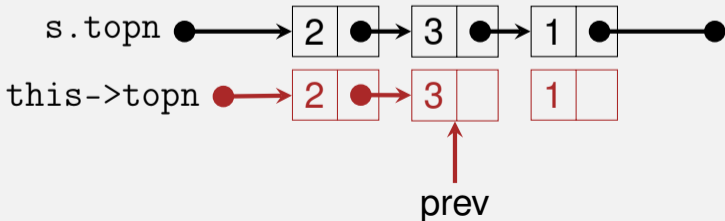
```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
 if (s.topn == nullptr) return;
 topn = new llnode(s.topn->value, nullptr);
 llnode* prev = topn;
 for(llnode* n = s.topn->next; n != nullptr; n = n->next){
 llnode* copy = new llnode(n->value, nullptr);
 prev->next = copy;
 prev = copy;
 }
}
```





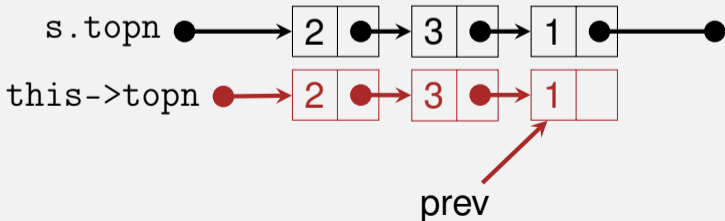
# It works with a Copy Constructor

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
 if (s.topn == nullptr) return;
 topn = new llnode(s.topn->value, nullptr);
 llnode* prev = topn;
 for(llnode* n = s.topn->next; n != nullptr; n = n->next){
 llnode* copy = new llnode(n->value, nullptr);
 prev->next = copy;
 prev = copy;
 }
}
```



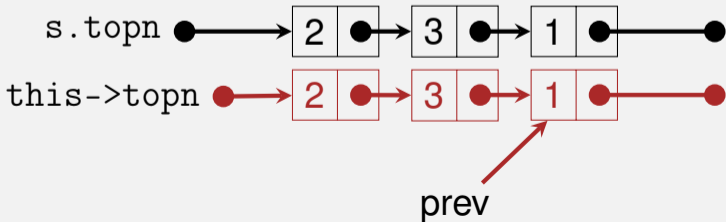
# It works with a Copy Constructor

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
 if (s.topn == nullptr) return;
 topn = new llnode(s.topn->value, nullptr);
 llnode* prev = topn;
 for(llnode* n = s.topn->next; n != nullptr; n = n->next){
 llnode* copy = new llnode(n->value, nullptr);
 prev->next = copy;
 prev = copy;
 }
}
```



# It works with a Copy Constructor

```
// POST: *this is initialized with a copy of s
stack::stack (const stack& s) : topn (nullptr) {
 if (s.topn == nullptr) return;
 topn = new llnode(s.topn->value, nullptr);
 llnode* prev = topn;
 for(llnode* n = s.topn->next; n != nullptr; n = n->next){
 llnode* copy = new llnode(n->value, nullptr);
 prev->next = copy;
 prev = copy;
 }
}
```



## Aside: copy recursively

```
llnode* copy (node* that){
 if (that == nullptr) return nullptr;
 return new llnode(that->value, copy(that->next));
}
```

Elegant, isn't it?

---

<sup>6</sup>not an overflow of the stack that we are implementing but the call stack

## Aside: copy recursively

```
llnode* copy (node* that){
 if (that == nullptr) return nullptr;
 return new llnode(that->value, copy(that->next));
}
```

Elegant, isn't it? Why did we not do it like this?

---

<sup>6</sup>not an overflow of the stack that we are implementing but the call stack

## Aside: copy recursively

```
llnode* copy (node* that){
 if (that == nullptr) return nullptr;
 return new llnode(that->value, copy(that->next));
}
```

Elegant, isn't it? Why did we not do it like this?

Reason: linked lists can become very long. `copy` could then lead to stack overflow<sup>6</sup>. Stack memory is usually smaller than heap memory.

---

<sup>6</sup>not an overflow of the stack that we are implementing but the call stack

# Initialization $\neq$ Assignment!

```
stack s1;
s1.push (1);
s1.push (3);
s1.push (2);
std::cout << s1 << "\n"; // 2 3 1
```

```
stack s2 = s1; // Initialisierung
```

```
s1.pop ();
std::cout << s1 << "\n"; // 3 1
s2.pop (); // ok: Copy Constructor!
```

# Initialization $\neq$ 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!

```
// POST: *this (left operand) becomes a
// copy of s (right operand)
stack& stack::operator= (const stack& s)
```

# It works with an Assignment Operator!

```
// POST: *this (left operand) becomes a
// copy of s (right operand)
stack& stack::operator= (const stack& s){
 if (topn != s.topn){ // no self-assignment
```

# It works with an Assignment Operator!

```
// POST: *this (left operand) becomes a
// copy of s (right operand)
stack& stack::operator= (const stack& s){
 if (topn != s.topn){ // no self-assignment
 stack copy = s; // Copy Construction
```

# It works with an Assignment Operator!

```
// POST: *this (left operand) becomes a
// copy of s (right operand)
stack& stack::operator= (const stack& s){
 if (topn != s.topn){ // no self-assignment
 stack copy = s; // Copy Construction
 std::swap(topn, copy.topn); // now copy has the garbage
```

# It works with an Assignment Operator!

```
// POST: *this (left operand) becomes a
// copy of s (right operand)
stack& stack::operator= (const stack& s){
 if (topn != s.topn){ // no self-assignment
 stack copy = s; // Copy Construction
 std::swap(topn, copy.topn); // now copy has the garbage
 } // copy is cleaned up -> deconstruction
 return *this; // return as L-Value (convention)
}
```

# It works with an Assignment Operator!

```
// POST: *this (left operand) becomes a
// copy of s (right operand)
stack& stack::operator= (const stack& s){
 if (topn != s.topn){ // no self-assignment
 stack copy = s; // Copy Construction
 std::swap(topn, copy.topn); // now copy has the garbage
 } // copy is cleaned up -> deconstruction
 return *this; // return as L-Value (convention)
}
```

Cool trick! 😊

# Done

```
class stack{
public:
 stack(); // constructor
 ~stack(); // destructor
 stack(const stack& s); // copy constructor
 stack& operator=(const stack& s); // assignment operator

 void push(int value);
 void pop();
 int top() const;
 bool empty() const;
 void print(std::ostream& out) const;
private:
 llnode* topn;
}
```



# Dynamic Datatype

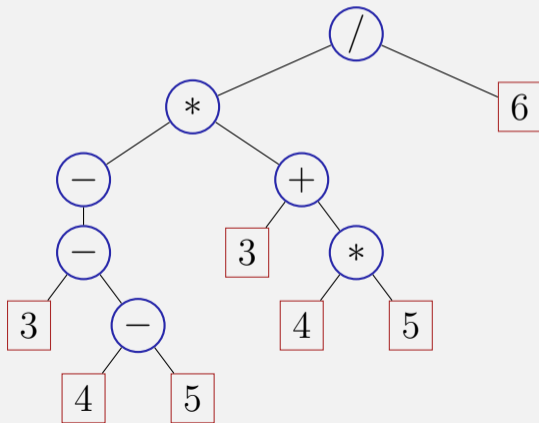
- Type that manages dynamic memory (e.g. our class for a stack)
- Minimal Functionality:
  - Constructors
  - Destructor
  - Copy Constructor
  - Assignment Operator

# Dynamic Datatype

- Type that manages dynamic memory (e.g. our class for a stack)
  - 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

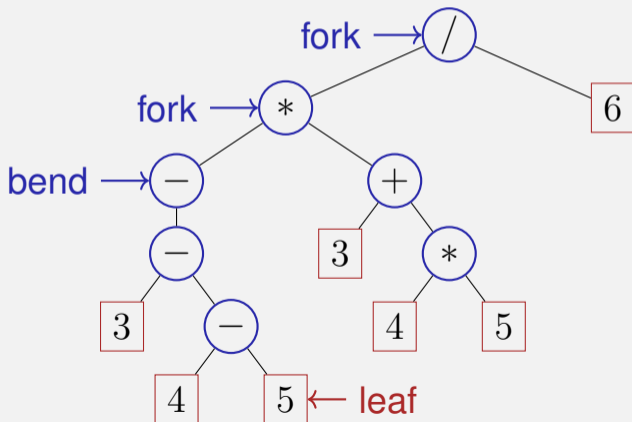
# (Expression) Trees

$$-(3-(4-5))*(3+4*5)/6$$



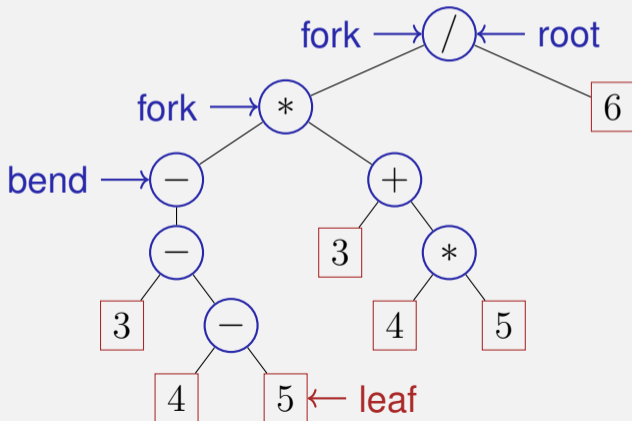
# (Expression) Trees

$$-(3-(4-5))*(3+4*5)/6$$



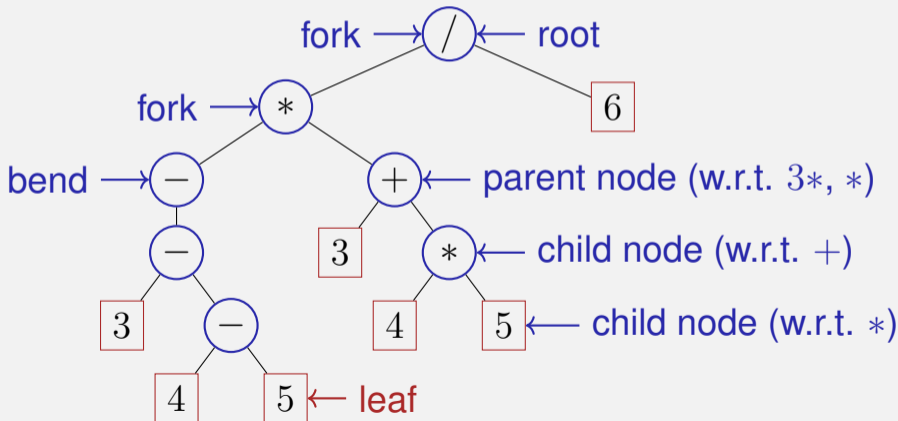
# (Expression) Trees

$$-(3-(4-5))*(3+4*5)/6$$

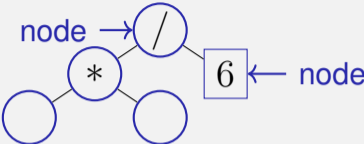


# (Expression) Trees

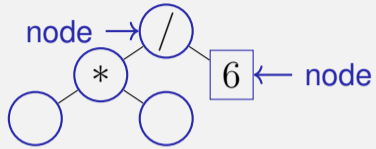
$$-(3-(4-5))*(3+4*5)/6$$



# Nodes: Forks, Bends or Leaves

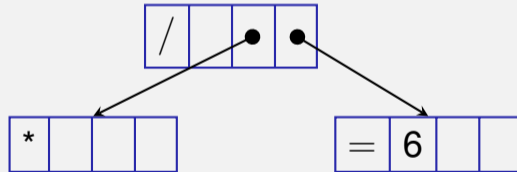
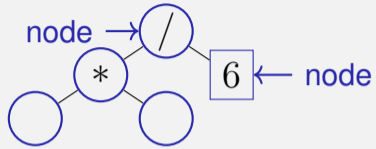


# Nodes: Forks, Bends or Leaves

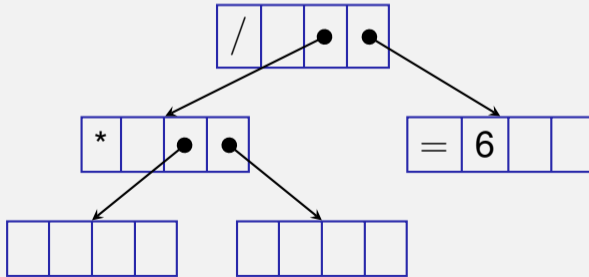
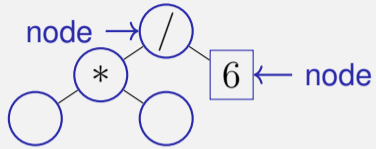




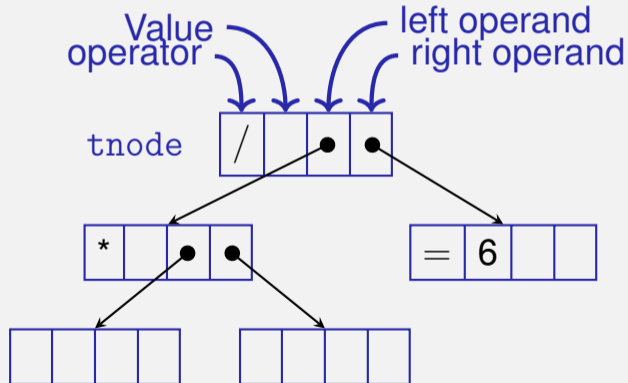
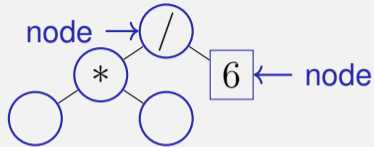
# Nodes: Forks, Bends or Leaves



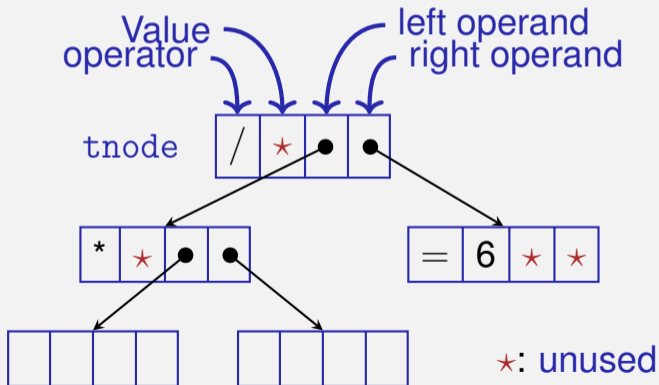
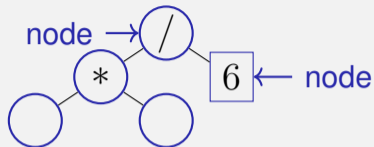
# Nodes: Forks, Bends or Leaves



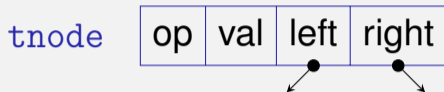
# Nodes: Forks, Bends or Leaves



# Nodes: Forks, Bends or Leaves



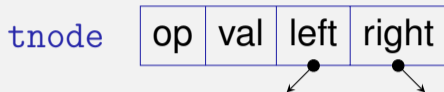
# Nodes (struct tnode)



```
struct tnode {
 char op; // leaf node: op is '='
 // internal node: op is '+', '-', '*', or '/'
 double val;
 tnode* left;
 tnode* right;

 tnode(char o, double v, tnode* l, tnode* r)
 : op(o), val(v), left(l), right(r) {}
};
```

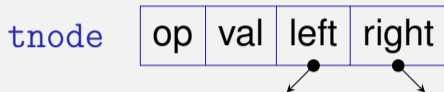
# Nodes (struct tnode)



```
struct tnode {
 char op; // leaf node: op is '='
 // internal node: op is '+', '-', '*', or '/'
 double val;
 tnode* left; // == nullptr for unary minus
 tnode* right;

 tnode(char o, double v, tnode* l, tnode* r)
 : op(o), val(v), left(l), right(r) {}
};
```

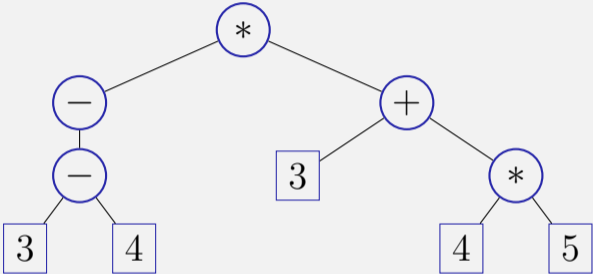
# Nodes (struct tnode)



```
struct tnode {
 char op; // leaf node: op is '='
 // internal node: op is '+', '-', '*', or '/'
 double val;
 tnode* left; // == nullptr for unary minus
 tnode* right;

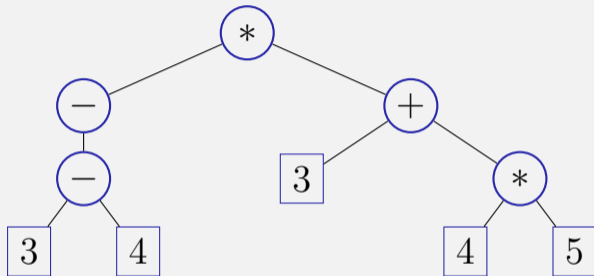
 tnode(char o, double v, tnode* l, tnode* r)
 : op(o), val(v), left(l), right(r) {}
};
```

# Size = Count Nodes in Subtrees



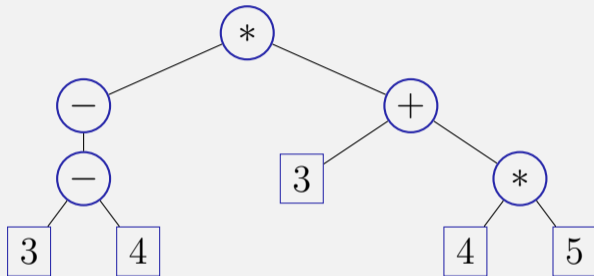


# Size = Count Nodes in Subtrees



- Size of a leaf: 1
- Size of other nodes: 1 + sum of child nodes' size

# Size = Count Nodes in Subtrees



- Size of a leaf: 1
- Size of other nodes: 1 + sum of child nodes' size
- E.g. size of the "+"-node is 5

# Count Nodes in Subtrees

```
// POST: returns the size (number of nodes) of
// the subtree with root n
int size (const tnode* n) {
 if (n){ // shortcut for n != nullptr
 return size(n->left) + size(n->right) + 1;
 }
 return 0;
}
```



# Evaluate Subtrees

```
// POST: evaluates the subtree with root n
```

```
double eval(const tnode* n){
 assert(n);
 if (n->op == '=') return n->val; ← leaf...
 double l = 0; ... or fork:
 if (n->left) l = eval(n->left); ← op unary, or left branch
 double r = eval(n->right); ← right branch
 switch(n->op){
 case '+': return l+r;
 case '-': return l-r;
 case '*': return l*r;
 case '/': return l/r;
 default: return 0;
 }
}
```



# Cloning Subtrees

```
// POST: a copy of the subtree with root n is made
// and a pointer to its root node is returned
tnode* copy (const tnode* n) {
 if (n == nullptr)
 return nullptr;
 return new tnode (n->op, n->val, copy(n->left), copy(n->right));
}
```

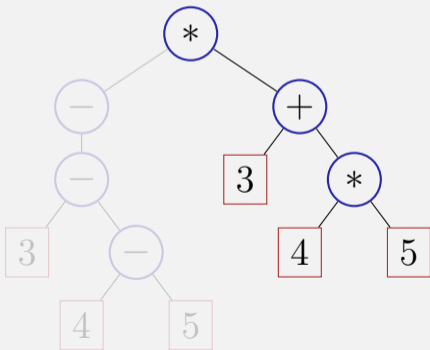




# Felling Subtrees

```
// POST: all nodes in the subtree with root n are deleted
```

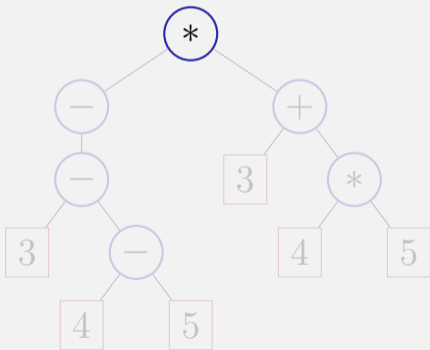
```
void clear(tnode* n) {
 if(n){
 clear(n->left);
 clear(n->right);
 delete n;
 }
}
```



# Felling Subtrees

```
// POST: all nodes in the subtree with root n are deleted
```

```
void clear(tnode* n) {
 if(n){
 clear(n->left);
 clear(n->right);
 delete n;
 }
}
```

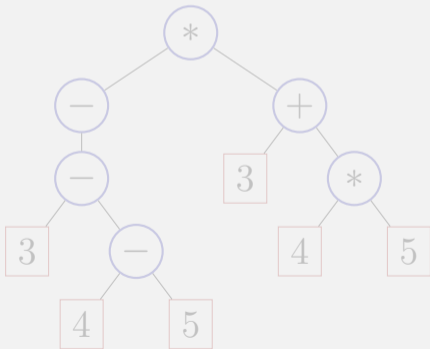




# Felling Subtrees

```
// POST: all nodes in the subtree with root n are deleted
```

```
void clear(tnode* n) {
 if(n){
 clear(n->left);
 clear(n->right);
 delete n;
 }
}
```



# Using Expression Subtrees

```
// Construct a tree for $1 - (-(3 + 7))$
tnode* n1 = new tnode('=', 3, nullptr, nullptr);
tnode* n2 = new tnode('=', 7, nullptr, nullptr);
tnode* n3 = new tnode('+', 0, n1, n2);
tnode* n4 = new tnode('-', 0, nullptr, n3);
tnode* n5 = new tnode('=', 1, nullptr, nullptr);
tnode* root = new tnode('-', 0, n5, n4);

// Evaluate the overall tree
std::cout << "1 - (-(3 + 7)) = " << eval(root) << '\n';


// Evaluate a subtree
std::cout << "3 + 7 = " << eval(n3) << '\n';

clear(root); // free memory
```

# Planting Trees

```
class texpression {
public:
 texpression (double d)
 : root (new tnode ('=', d, 0, 0)) {}
 ...
private:
 tnode* root;
};
```

creates a tree with  
one leaf



# Letting Trees Grow

```
texpression& texpression::operator-= (const texpression& e)
{
 assert (e.root);
 root = new tnode ('-', 0, root, copy(e.root));
 return *this;
}
```



# Letting Trees Grow

```
texpression& texpression::operator-= (const texpression& e)
{
 assert (e.root);
 root = new tnode ('-', 0, root, copy(e.root));
 return *this;
}
```



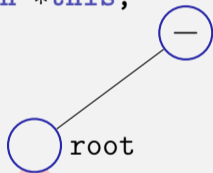
\*this



e

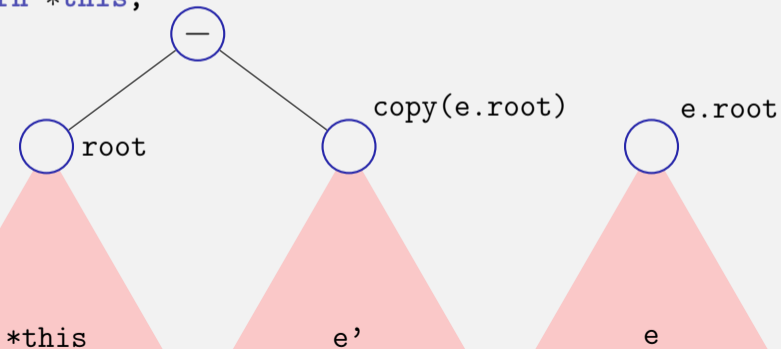
# Letting Trees Grow

```
texpression& texpression::operator-= (const texpression& e)
{
 assert (e.root);
 root = new tnode ('-', 0, root, copy(e.root));
 return *this;
}
```



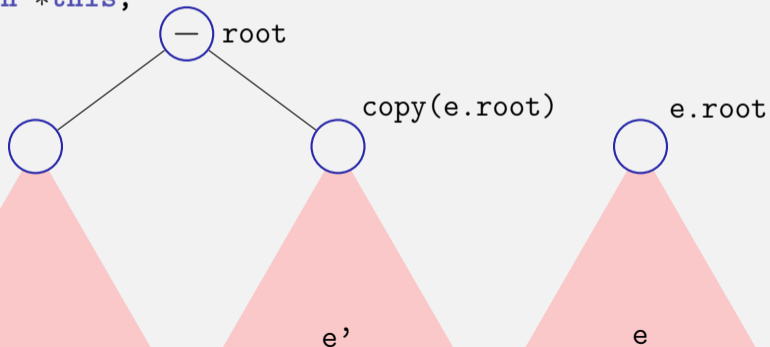
# Letting Trees Grow

```
texpression& texpression::operator-= (const texpression& e)
{
 assert (e.root);
 root = new tnode ('-', 0, root, copy(e.root));
 return *this;
}
```



# Letting Trees Grow

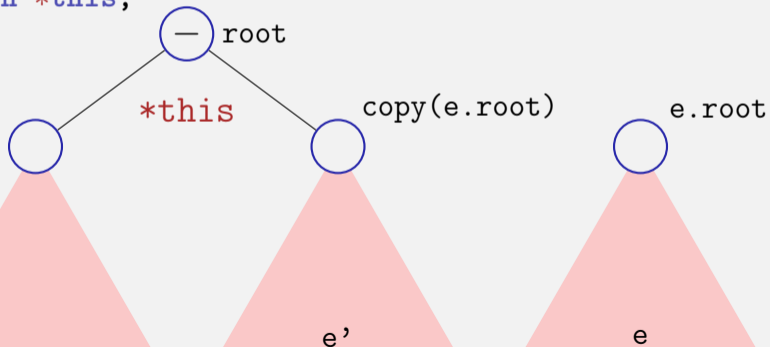
```
texpression& texpression::operator-= (const texpression& e)
{
 assert (e.root);
 root = new tnode ('-', 0, root, copy(e.root));
 return *this;
}
```





# Letting Trees Grow

```
texpression& texpression::operator-= (const texpression& e)
{
 assert (e.root);
 root = new tnode ('-', 0, root, copy(e.root));
 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

```
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;
```

3

# 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;
```

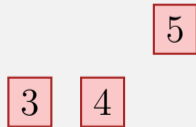
3

4

# Raising Trees

```
texpression operator- (const texpression& l,
 const texpression& r){
 texpression result = l;
 return result -= r;
}
```

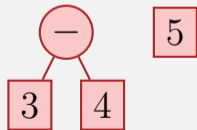
```
texpression a = 3;
texpression b = 4;
texpression c = 5;
texpression d = a-b-c;
```



# Raising Trees

```
expression operator- (const expression& l,
 const expression& r){
 expression result = l;
 return result -= r;
}
```

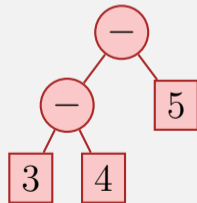
```
expression a = 3;
expression b = 4;
expression c = 5;
expression d = a-b-c;
```



# Raising Trees

```
expression operator- (const expression& l,
 const expression& r){
 expression result = l;
 return result -= r;
}
```

```
expression a = 3;
expression b = 4;
expression c = 5;
expression d = a-b-c;
```



# Rule of three: Clone, reproduce and cut trees

```
expression::~~expression(){
 clear(root);
}
```

```
expression::expression (const expression& e)
 : root(copy(e.root)) { }
```

```
expression::expression& operator=(const expression& e){
 if (root != e.root){
 expression cp = e;
 std::swap(cp.root, root);
 }
 return *this;
}
```



# Concluded

```
class texpression{
public:
 texpression (double d); // constructor
 ~texpression(); // destructor
 texpression (const texpression& e); // copy constructor
 texpression& operator=(const texpression& e); // assignment op
 texpression operator-();
 texpression& operator-=(const texpression& e);
 texpression& operator+=(const texpression& e);
 texpression& operator*=(const texpression& e);
 texpression& operator/=(const texpression& e);
 double evaluate();
private:
 tnode* root;
};
```

# From values to trees!

```
// term = factor { "*" factor | "/" factor }
double term (std::istream& is){
 double value = factor (is);
 while (true) {
 if (consume (is, '*'))
 value *= factor (is);
 else if (consume (is, '/'))
 value /= factor (is);
 }
 return value;
}
```

calculator.cpp  
(expression value)

# From values to trees!

```
using number_type = double;
```

```
// term = factor { "*" factor | "/" factor }
number_type term (std::istream& is){
 number_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)

# From values to trees!

```
using number_type = texpression ;
```

```
// term = factor { "*" factor | "/" factor }
number_type term (std::istream& is){
 number_type value = factor (is);
 while (true) {
 if (consume (is, '*'))
 value *= factor (is);
 else if (consume (is, '/'))
 value /= factor (is);
 }
 return value;
}
```

double\_calculator.cpp  
(expression value)

→

texpression\_calculator.cpp  
(expression tree)

# Concluding Remark

- In this lecture, we have intentionally refrained from implementing member functions in the node classes of the list or tree.<sup>7</sup>
- When there is inheritance and polymorphism used, the implementation of the functionality such as evaluate, print, clear (etc..) is better implemented in member functions.
- In any case it is not a good idea to implement the memory management of the composite data structure list or tree within the nodes.

---

<sup>7</sup>Parts of the implementations are even simpler (because the case `n==nullptr` can be caught more easily

## **22. 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

# Kapitelüberblick

- 1. Introduction
- 2. Integers
- 3. Booleans
- 4. Defensive Programming
- 5./6. Control Statements
- 7./8. Floating Point Numbers
- 9./10. Functions
- 11. Reference Types
- 12./13. Vectors and Strings
- 14./15. Recursion
- 16. Structs and Overloading
- 17. Classes
- 18./19. Dynamic Datastructures
- 20. Containers, Iterators and Algorithms
- 21. Dynamic Datatypes and Memory Management



# 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

- ©
  - Boolean functions, completeness
  - DeMorgan rules
- Ⓛ
  - 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>`
- Ⓔ
  - Div-Mod identity.

## 4. Defensive Programming

- ③ ■ Assertions and Constants
- ④ ■ The assert-statement, `#include <cassert>`
  - `const int speed_of_light=2999792458`
- ⑤ ■ Assertions for the GCD

# 5./6. Control Statements

- linear control flow vs. interesting programs
- selection statements, iteration statements
- (avoiding) endless loops, halting problem
- Visibility and scopes, automatic memory
- equivalence of iteration statement
- 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;`
- Switch statement `switch(grade) {case 6: }`
- sum computation (Gauss), prime number tests, Collatz sequence, Fibonacci numbers, calculator, output grades

# 7./8. Floating Point Numbers

- ① ■ correct computation: Celsius / Fahrenheit
- ② ■ 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*
- ③ ■ types `float`, `double`
  - floating point literals `1.23e-7f`
- ④ ■ Celsius/Fahrenheit, Euler, Harmonic Numbers

# 9./10. Functions

- Ⓜ ■ Computation of Powers
- Ⓒ ■ Encapsulation of Functionality
  - functions, formal arguments, arguments
  - scope, forward declarations
  - procedural programming, modularization, separate compilation
  - *Stepwise Refinement*
- Ⓛ ■ declaration and definition of functions `double pow(double b, int e){ ... }`
  - function call `pow (2.0, -2)`
  - the type `void`
- ⓔ ■ powers, perfect numbers, minimum, calendar

# 11. Reference Types

- ④ ■ Swap
- ③ ■ value- / reference- semantics, pass by value, pass by reference, return by reference
  - lifetime of objects / temporary objects
  - constants
- ② ■ reference type `int& a`
  - call by reference, return by reference `int& increment (int& i)`
  - const guideline, const references, reference guideline
- ① ■ swap, increment



# 12./13. Vectors and Strings

- Ⓜ
  - Iterate over data: sieve of erathosthenes
- Ⓒ
  - vectors, memory layout, random access
  - (missing) bound checks
  - vectors
  - characters: ASCII, UTF8, texts, strings
- Ⓛ
  - vector types `std::vector<int> a {4,3,5,2,1};`
  - characters and texts, the type `char` `char c = 'a';`, Konversion nach `int`
  - vectors of vectors
  - Streams `std::istream`, `std::ostream`
- Ⓔ
  - sieve of Erathosthenes, Caesar-code, shortest paths

# 14./15. Recursion

- Ⓜ recursive math. functions, the n-Queen problem, Lindenmayer systems, a command line calculator
- Ⓒ
  - recursion
  - call stack, memory of recursion
  - correctness, termination,
  - recursion vs. iteration
  - Backtracking, EBNF, formal grammars, parsing
- Ⓔ
  - factorial, GCD, sudoku-solver, command line calculator

# 16. Structs and Overloading

- ④ ■ build your own rational number
- ③ ■ heterogeneous data types  
■ function and operator overloading  
■ encapsulation of data
- ② ■ 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
- ① ■ rational numbers, complex numbers

# 17. Classes

- ④ rational numbers with encapsulation
- ③ Encapsulation, Construction, Member Functions
- ②
  - classes `class rational { ... };`
  - access control `public:/private:`
  - member functions `int rational::denominator () const`
  - The implicit argument of the member functions
- ① finite rings, complex numbers

# 18./19. Dynamic Datastructures

- ④ ■ Our own vector
- ③ ■ linked list, allocation, deallocation, dynamic data type
- ② ■ The `new` statement
  - pointer `int* x;`, Null-pointer `nullptr.`
  - address and dereference operator `int *ip = &i; int j = *ip;`
  - pointer and const `const int *a;`
- ① ■ linked list, stack

## 20. Containers, Iterators and Algorithms

- ① ■ vectors are containers
- ② ■ iteration with pointers
  - containers and iterators
  - algorithms
- ③ ■ Iterators `std::vector<int>::iterator`
  - Algorithms of the standard library `std::fill (a, a+5, 1);`
  - implement an iterator
  - iterators and const
- ④ ■ output a vector, a set

# 21. Dynamic Datatypes and Memory Management

- ①
  - Stack
  - Expression Tree
- ②
  - Guideline "dynamic memory"
  - Pointer sharing
  - Dynamic Datatype
  - Tree-Structure
- ③
  - `new` and `delete`
  - Destructor `stack::~~stack()`
  - Copy-Constructor `stack::stack(const stack& s)`
  - Assignment operator `stack& stack::operator=(const stack& s)`
  - Rule of Three
- ④
  - Binary Search Tree

# The End

End of the Course