

# Organization

# Course Goals

Learn to write **good** C++

- Basic syntax
- Common idioms and best practices

Learn to implement **large systems** with C++

- C++ standard library and Linux ecosystem
- Tools and techniques (building, debugging, etc.)

Learn to write **high-performance** code with C++

- Multithreading and synchronization
- Performance pitfalls

# Formal Prerequisites

Knowledge equivalent to the lectures

- Introduction to Informatics 1 (IN0001)
- Fundamentals of Programming (IN0002)
- Fundamentals of Algorithms and Data Structures (IN0007)

Additional formal prerequisites (B.Sc. Informatics)

- Introduction to Computer Architecture (IN0004)
- Basic Principles: Operating Systems and System Software (IN0009)

Additional formal prerequisites (B.Sc. Games Engineering)

- Operating Systems and Hardware oriented Programming for Games (IN0034)

# Practical Prerequisites

## Practical prerequisites

- **No previous experience with C or C++ required**
- Familiarity with another general-purpose programming language

## Operating System

- Working Linux operating system (e.g. Ubuntu)
- Basic experience with Linux (in particular with shell)
- You are free to use your favorite OS, **we only support Linux**

# Lecture & Tutorial

- Lecture: **Tuesday, 14:00 – 16:00, MI 02.11.018**
- Tutorial: **Friday, 10:00 – 12:00, MI 02.11.018**
  - Discuss assignments and any questions
  - First **two** tutorials are additional lectures
- **Attendance is mandatory**
- Announcements on the website and Mattermost

# Assignments

- Brief non-coding quiz at the beginning of random lectures or tutorials
- Weekly programming assignments published after each lecture
  - No teams
  - Due on Thursday the following week at 23:59 (i.e. 9 days per assignment)
  - Managed through our GitLab (more details in first tutorial)
  - Deadline is enforced automatically (no exceptions)
- Final (larger) project at end of the semester
  - No teams
  - Published mid-June
  - Due 11.08.2019 at 23:59 (two weeks after last lecture)
  - Managed through our GitLab (more details in first tutorial)
  - Deadline is enforced automatically (no exceptions)

# Grading

## Grading system

- Quizzes: Varying number of points
- Weekly assignments: Varying number of points depending on workload
- Final project

## Final grade consists of

- $\approx 60\%$  programming assignments
- $\approx 30\%$  final project
- $\approx 10\%$  quizzes

# Literature

## Primary

- C++ Reference Documentation. (<https://en.cppreference.com/>)
- Lippman, 2013. *C++ Primer (5th edition)*. Only covers C++11.
- Stroustrup, 2013. *The C++ Programming Language (4th edition)*. Only covers C++11.
- Meyers, 2015. *Effective Modern C++*. 42 specific ways to improve your use of C++11 and C++14..

## Supplementary

- Aho, Lam, Sethi & Ullman, 2007. *Compilers. Principles, Techniques & Tools (2nd edition)*.
- Tanenbaum, 2006. *Structured Computer Organization (5th edition)*.

# Contact

## Important links

- Website: <http://db.in.tum.de/teaching/ss19/c++praktikum>
- E-Mail: [freitagm@in.tum.de](mailto:freitagm@in.tum.de), [sichert@in.tum.de](mailto:sichert@in.tum.de)
- GitLab: <https://gitlab.db.in.tum.de/cpplab19>
- Mattermost: <https://mattermost.db.in.tum.de/cpplab19>

# Introduction

# What is C++?

Multi-paradigm general-purpose programming language

- Imperative programming
- Object-oriented programming
- Generic programming
- Functional programming

Key characteristics

- Compiled language
- Statically typed language
- Facilities for low-level programming

# A Brief History of C++

## Initial development

- Bjarne Stroustrup at Bell Labs (since 1979)
- In large parts based on C
- Inspirations from Simula67 (classes) and Algol68 (operator overloading)

## First ISO standardization in 1998 (C++98)

- Further amendments in following years (C++03, C++11, C++14)
- Current standard: C++17
- Next planned standard: C++20

# Why Use C++?

## Performance

- Flexible level of abstraction (very low-level to very high-level)
- High-performance even for user-defined types
- Direct mapping of hardware capabilities
- Zero-overhead rule: “What you don’t use, you don’t pay for.” (Bjarne Stroustrup)

## Flexibility

- Choose suitable programming paradigm
- Comprehensive ecosystem (tool chains & libraries)
- Scales easily to very large systems (with some discipline)
- Interoperability with other programming languages (especially C)

# Background

# The Central Processing Unit (1)

"Brains" of the computer

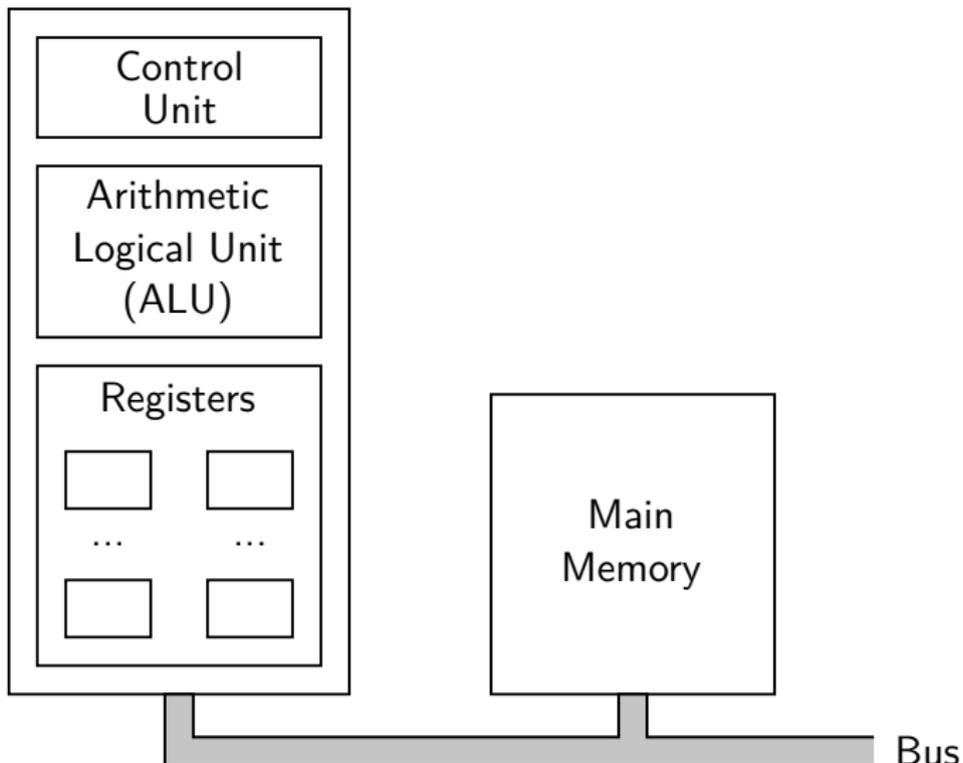
- Execute programs stored in main memory
- Fetch, examine and execute instructions

Connected to other components by a **bus**

- Collection of parallel wires for transmitting signals
- External (inter-device) and internal (intra-device) buses

# The Central Processing Unit (2)

## Central Processing Unit



# Components of a CPU

## Control Unit

- Fetch instructions from memory and determine their type
- Orchestrate other components

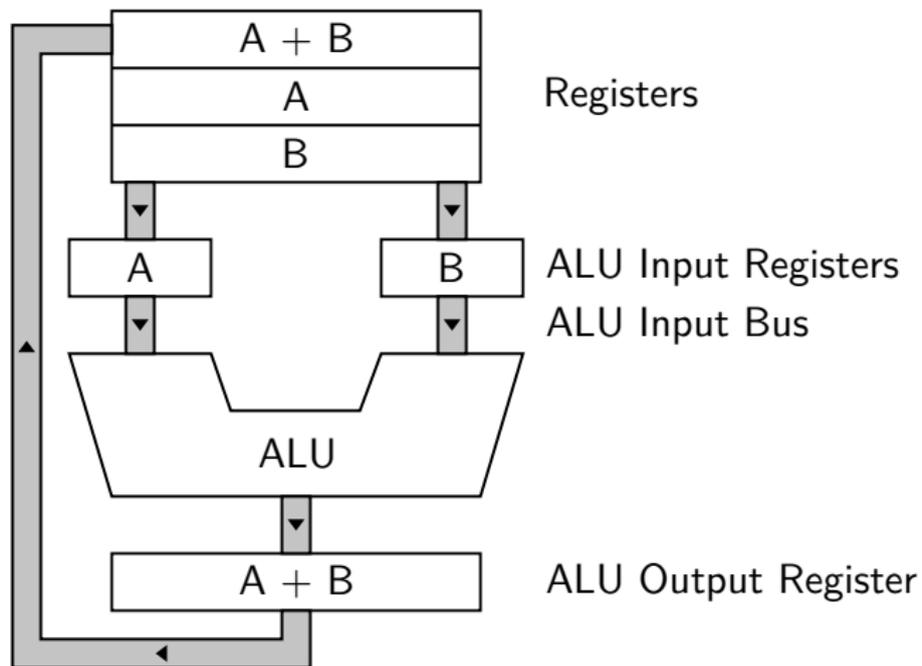
## Arithmetic Logical Unit (ALU)

- Perform operations (e.g. addition, logical AND, ...)
- "Workhorse" of the CPU

## Registers

- Small, high-speed memory with fixed size and function
- Temporary results and control information (one number / register)
- Program Counter (PC): Next instruction to be fetched
- Instruction Register (IR): Instruction currently being executed

# Data Path (1)



## Data Path (2)

Internal organization of a typical von Neumann CPU

- Registers feed two ALU input registers
- ALU input registers hold data while ALU performs operations
- ALU stores result in output register
- ALU output register can be stored back in register

⇒ **Data Path Cycle**

- Central to most CPUs (in particular x86)
- Fundamentally determines capabilities and speed of a CPU

# Instruction Categories

## Register-register instructions

- Fetch two operands from registers into ALU input registers
- Perform some computation on values
- Store result back into one of the registers
- Low latency, high throughput

## Register-memory instructions

- Fetch memory words into registers
- Store registers into memory words
- Potentially incur high latency and stall the CPU

# Fetch-Decode-Execute Cycle

Rough steps to execute an instruction

1. Load the next instruction from memory into the instruction register
2. Update the program counter to point to the next instruction
3. Determine the type of the current instruction
4. Determine the location of memory words accessed by the instruction
5. If required, load the memory words into CPU registers
6. Execute the instruction
7. Continue at step 1

Central to the operation of all computers

# Execution vs. Interpretation

We do not have to implement the fetch-decode-execute cycle in hardware

- Easy to write an interpreter in software (or some hybrid)
- Break each instruction into small steps (**microoperations**, or **μops**)
- Microoperations can be executed in hardware

Major implications for computer organization and design

- Interpreter requires much simpler hardware
- Easy to maintain backward compatibility
- Historically led to interpreter-based microprocessors with very large instruction sets

# RISC vs. CISC

## Complex Instruction Set Computer (CISC)

- Large instruction set
- Large overhead due to interpretation

## Reduced Instruction Set Computer (RISC)

- Small instruction set executed in hardware
- Much faster than CISC architectures

## CISC architectures still dominate the market

- Backward compatibility is paramount for commercial customers
- Modern Intel CPUs: RISC core for most common instructions

# Instruction-Level Parallelism

Just increasing CPU clock speed is not enough

- Fetching instructions from memory becomes a major bottleneck
- Increase instruction throughput by parallel execution

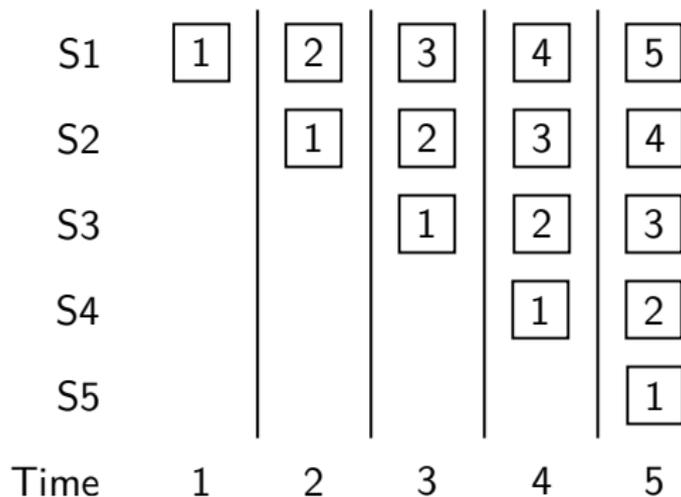
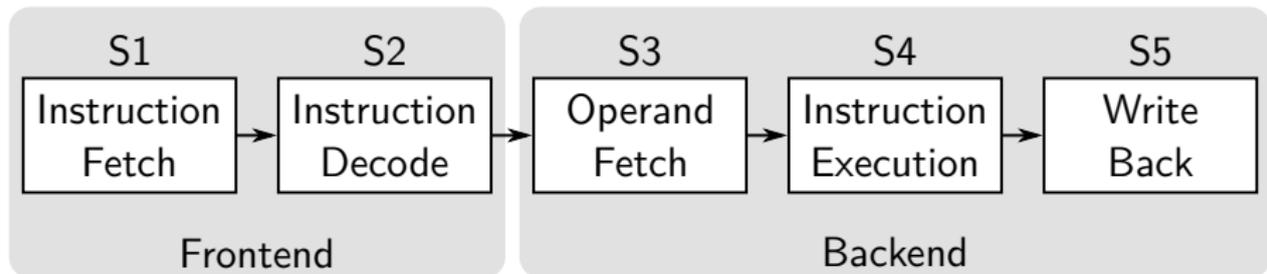
Instruction Prefetching

- Fetch instructions from memory in advance
- Hold prefetched instructions in buffer for fast access
- Breaks instruction execution into fetching and actual execution

**Pipelining**

- Divide instruction execution into many steps
- Each step handled in parallel by dedicated piece of hardware
- Central to modern CPUs

# Pipelining (1)



## Pipelining (2)

### Pipeline frontend (x86)

- Fetch instructions from memory in-order
- Decode assembly instructions to microoperations
- Provide stream of work to pipeline backend (Skylake: 6  $\mu$ ops / cycle)
- Requires branch prediction (implemented in special hardware)

### Pipeline backend (x86)

- Execute microoperations out-of-order as soon as possible
- Complex bookkeeping required
- Microoperations are run on execution units (e.g. ALU, FPU)

# Superscalar Architectures

Multiple pipelines could execute instructions even faster

- Parallel instructions must not conflict over resources
- Parallel instructions must be independent
- Incurs hardware replication

Superscalar architectures

- S3 stage is typically much faster than S4
- Issue multiple instructions per clock cycle in a single pipeline
- Replicate (some) execution units in S4 to keep up with S3

# Multiprocessors

Include multiple CPUs in a system

- Shared access to main memory over common bus
- Requires coordination in software to avoid conflicts
- CPU-local caches to reduce bus contention

# Main Memory

Main memory provides storage for data and programs

- Information is stored in binary units (**bits**)
- Bits are represented by values of a measurable quantity (e.g. voltage)
- More complex data types are translated into suitable binary representation (e.g. two's complement for integers, IEEE 754 for floating point numbers, ...)
- Main memory is (much) slower but (much) larger than registers

# Memory Addresses (1)

Memory consists of a number of cells

- All cells contain the same number of bits
- Each cell is assigned a unique number (its **address**)
- Logically adjacent cells have consecutive addresses
- De-facto standard: 1 byte per cell  $\Rightarrow$  **byte-addressable** memory (with some caveats, more details later)
- Usually 1 byte is defined to consist of 8 bits

Instructions typically operate on entire groups of bytes (**memory words**)

- 32-bit architecture: 4 bytes / word
- 64-bit architecture: 8 bytes / word
- Memory accesses commonly need to be aligned to word boundaries

Addresses are memory words themselves

- Addresses can be stored in memory or registers just like data
- Word size determines the maximum amount of addressable memory

## Memory Addresses (2)

Example: two-byte addresses, one-byte cells

Hexadecimal

		Address							
		00	01	02	03	04	05	06	07
Address	0000	48	65	6c	6c	6f	20	57	6f
	0008	72	6c	64	21	20	49	20	6c
	0010	69	6b	65	20	43	2b	2b	21

ASCII

		Address							
		00	01	02	03	04	05	06	07
Address	0000	H	e	l	l	o		W	o
	0008	r	l	d	!		l		l
	0010	i	k	e		C	+	+	!

# Byte Ordering (1)

ASCII requires just one byte per character

- Fits into a single memory cell
- What about data spanning multiple cells (e.g. 32-bit integers)?

Bytes of wider data types can be ordered differently (**endianness**)

- Most significant byte first  $\Rightarrow$  **big-endian**
- Least significant byte first  $\Rightarrow$  **little-endian**

Most current architectures are little-endian

- But big-endian architectures such as ARM still exist (although many support little-endian mode)
- Has to be taken into account for low-level memory manipulation

## Byte Ordering (2)

Big-endian byte ordering can lead to unexpected results

- Conversions between word-sizes need care and address calculations

00	01	02	03
00	00	00	2a

32-bit integer  
at address 00:  
**42<sub>10</sub>**

16-bit integer  
at address 00:  
**0<sub>10</sub>**

00	01	02	03
00	2a	00	00

16-bit integer  
at address 00:  
**42<sub>10</sub>**

32-bit integer  
at address 00:  
**2,752,512<sub>10</sub>**

## Byte Ordering (3)

Little-endian byte ordering can lead to unexpected results

- Mainly because we are used to reading from left to right

4-byte words in  
byte-wise lexicographical  
order

interpreted as  
little-endian  
32-bit integers

00 01 02 03		
00 00 00 00	→	$0_{10}$
00 01 00 00	→	$256_{10}$
00 02 00 00	→	$512_{10}$
01 00 00 00	→	$1_{10}$
01 01 00 00	→	$257_{10}$
01 02 00 00	→	$513_{10}$

# Cache Memory (1)

Main memory has substantial latency

- Usually 10s of nanoseconds
- Memory accesses cause CPU to **stall** for multiple cycles

Memory accesses very commonly exhibit spatial and temporal locality

- When a memory load is issued adjacent words are likely accessed too
- The same memory word is likely to be accessed multiple times within a small number of instructions
- Locality can be exploited to hide main memory latency

## Cache Memory (2)

Introduce small but fast **cache** between CPU and main memory

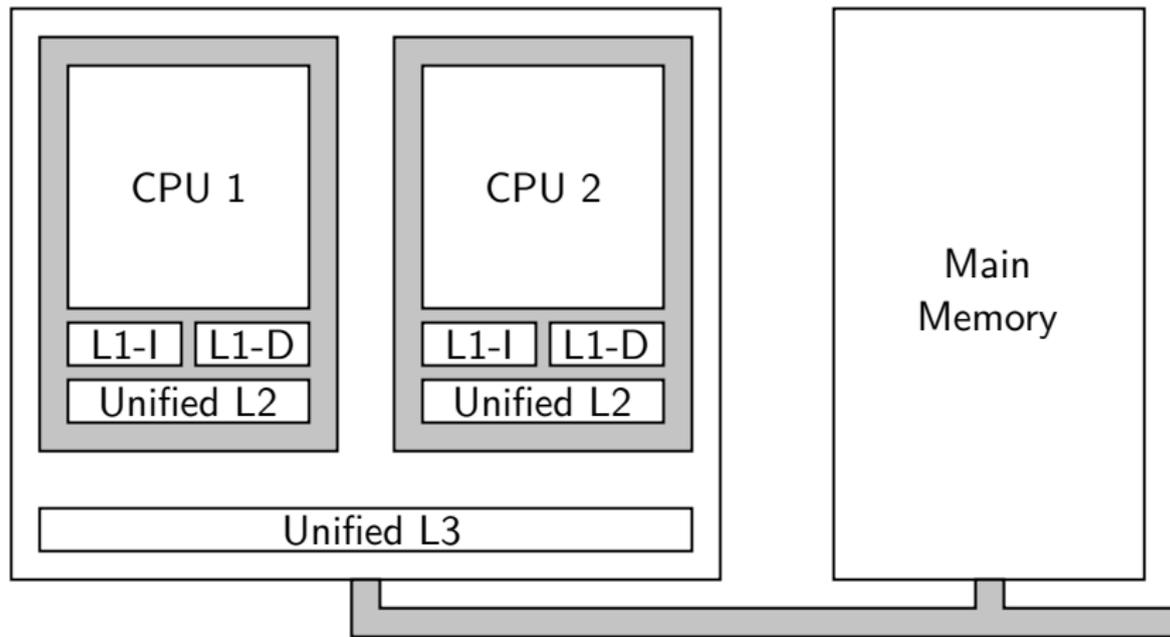
- CPU transparently keeps recently accessed data in cache (temporal locality)
- Memory is divided into blocks (**cache lines**)
- Whenever a memory cell is referenced, load the entire corresponding cache line into the cache (spatial locality)
- Requires specialized eviction strategy

Intel CPUs

- 3 caches (L1, L2, L3) with increasing size and latency
- Caches are inclusive (i.e. L1 is replicated within L2, and L2 within L3)
- 64 byte cache lines

## Cache Memory (3)

Typical cache hierarchy on Intel CPUs



## Cache Memory (4)

Cache memory interacts with byte-addressability

- On Intel, we can access each byte individually
- However, on each access, the entire corresponding cache line is loaded
- Can lead to read amplification (e.g. if we read every 64th byte)

Designing cache-efficient data structures is a major challenge

- A programmer has to take care that data is kept in caches as long as possible
- However, there is no direct control over caches
- Must be ensured through suitable programming techniques